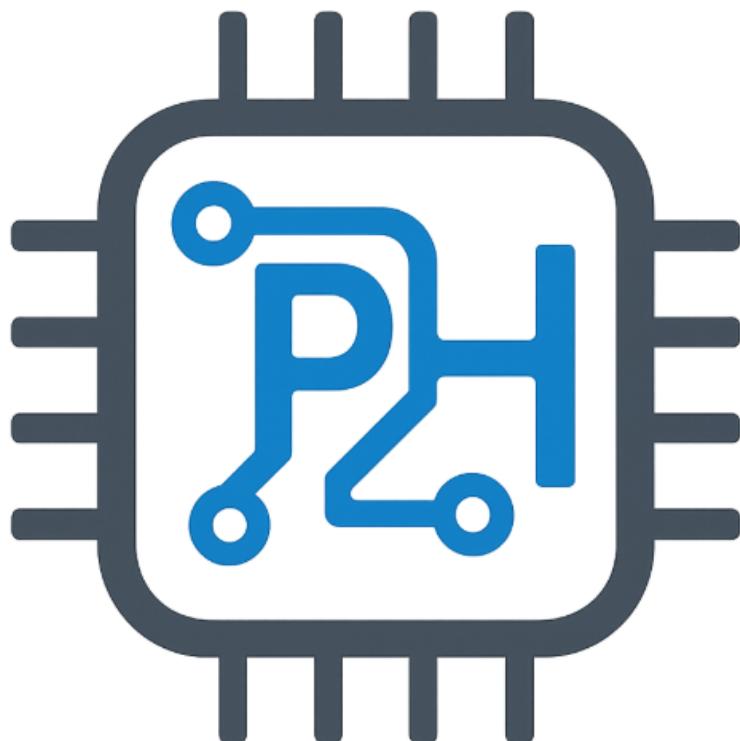

Final Report

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

Executive Summary	4
1 Introduction	5
1.1 Project Context	5
1.2 Target Platforms	5
1.3 Functional Scope	5
2 Objectives	5
3 Methodology	6
3.1 Overall System Architecture	6
3.1.1 Layered View	6
3.1.2 Layer Description	6
3.1.3 Rationale for the Event-Driven Model	7
3.1.4 Hardware/Software Separation	7
3.2 Development Environment	7
3.2.1 Keil µVision	7
3.2.2 Build Configurations	8
3.3 Time Management	8
3.3.1 64-bit Monotonic Clock	8
3.3.2 Resolution and Range	9
3.3.3 Tick-to-Time Conversion	9
3.3.4 Periodic Timing	9
3.4 Peripheral Management	9
3.4.1 LEDs	9
3.4.2 Buttons and External Interrupts	10
3.4.3 ISR Implementation Differences	11
3.4.4 GPIO Monitors	11
3.5 Watchdog	11
3.6 Concurrency and Event Model	11
3.6.1 Event Definition	11
3.6.2 FIFO Event Queue	12
3.6.3 Overflow Handling	12
3.6.4 Critical Sections	12
3.7 Power Management	12
3.7.1 Power Modes	12
3.7.2 Busy-Wait vs. Low Power vs. Deep Sleep	12
3.7.3 Wake-Up by Button	13
3.7.4 Inactivity Timeout	13
3.8 System Services	13
3.8.1 Alarm Service	13
3.8.2 Event Manager (Dispatcher)	14
3.8.3 Subscription and Cancellation	14
4 Results	15
4.1 Bit Counter Strike	15
4.1.1 Functional Description	15
4.1.2 Game state diagram	15
4.1.3 Configuration Parameters	16

4.1.4	Difficulty Progression	16
4.1.5	System Integration	16
4.2	Final Game: Beat Hero	16
4.2.1	Functional Description	16
4.2.2	Game state diagram	17
4.2.3	Scoring System	17
4.2.4	Difficulty Progression	18
4.2.5	Metrics and Statistics	18
4.2.6	Reset via Long Press	18
4.3	Validation and Testing	19
4.3.1	Module Tests	19
4.3.2	Queue Overflow Test	19
4.3.3	Watchdog Verification	19
4.3.4	Power Consumption Measurements	20
5	Conclusions	23
5.1	Degree of Objective Fulfillment	23
5.2	Conclusions and Lessons Learned	23
5.3	Self-Assessment Notes	24
References		25
Appendices		26
Appendix A: Statistics		26
Appendix B: Beat Hero Game		27
Appendix C: Event Manager		34
Appendix D: FIFO Queue		36
Appendix E: Random (LPC)		38
Appendix F: Random (nRF)		38
Appendix G: Critical Sections (LPC)		39
Appendix H: Critical Sections (nRF)		39
Appendix I: Button Driver		40
Appendix J: Alarm Service		44

List of Figures

1	Button driver state machine with debouncing	10
2	State diagram of the Bit Counter Strike game	15
3	State diagram of the Beat Hero game	17
4	Power measurement — Blink v2 (busy-wait)	20
5	Power measurement — Blink v3 (low power with WFI)	20
6	Power measurement — Blink v3 bis (awake mode)	21
7	Power measurement — Blink v3 bis (sleep/System OFF)	21
8	Power measurement — Beat Hero during gameplay	22
9	Power measurement — Beat Hero with the CPU sleeping	22

List of Tables

1	Platform feature comparison: LPC2105 vs. nRF52840 DK	5
2	Layered architecture of the embedded system	6
3	Build configurations in Keil uVision	8
4	Resolution and range of the 64-bit monotonic clock	9
5	System power modes	12
6	Configuration parameters of the Bit Counter Strike game	16
7	Beat Hero scoring system	17
8	Per-session module test results	19
9	Power consumption comparison across system states	23
10	Objective fulfillment status	23
11	Authors' self-assessment	24

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 (GPIOE)
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 `SESSION` 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):

```

1 static volatile uint32_t s_overflows_t1 = 0;
2
3 void T1_ISR(void) __irq {
4     T1IR = 1;
5     s_overflows_t1++;
6     VICVectAddr = 0;
7 }
8
9 uint64_t hal_tiempo_actual_tick64(void) {
10    uint32_t hi1, lo1, hi2, lo2;
11    lo1 = T1TC;
12    hi1 = s_overflows_t1;
13    lo2 = T1TC;
14    if (lo2 < lo1) {
15        hi2 = s_overflows_t1;
16        return ((uint64_t)hi2 * 0x10000000ULL) + lo2;
17    }
18    return ((uint64_t)hi1 * 0x10000000ULL) + lo2;
19 }
```

nRF52840 (hal_tiempo_nrf.c):

```

1 static volatile uint64_t tiempo_64b = 0;
2
3 void TIMERO0_IRQHandler(void) {
4     if (NRF_TIMER0->EVENTS_COMPARE[0]) {
5         NRF_TIMER0->EVENTS_COMPARE[0] = 0;
6         tiempo_64b += (1ULL << 32);
7     }
8 }
9
10 uint64_t hal_tiempo_actual_tick64(void) {
11     uint64_t t1, t2;
12     uint32_t ahora;
13     do {
14         t1 = tiempo_64b;
15         NRF_TIMER0->TASKS_CAPTURE[3] = 1;
16         ahora = NRF_TIMER0->CC[3];
17         t2 = tiempo_64b;
18     } while (t1 != t2);
19     return t1 + (uint64_t)ahora;
```

20 }

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:

```

1 Tiempo_us_t drv_tiempo_actual_us(void) {
2     uint64_t ticks = hal_tiempo_actual_tick64();
3     return (Tiempo_us_t)(ticks / s_info.ticks_per_us);
4 }
```

3.3.4. Periodic Timing

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

```

1 void drv_tiempo_periodico_ms(Tiempo_ms_t ms,
2                               void(*callback)(uint32_t, uint32_t),
3                               uint32_t ID_evento) {
4     s_cb_app = callback;
5     s_id = ID_evento;
6     uint32_t ticks = ms * 1000 * s_info.ticks_per_us;
7     hal_tiempo_reloj_periodico_tick(ticks, drv_tiempo_RSI);
8 }
```

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_comutar(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`)

```

1 static inline uint32_t hw_level_from_status(LED_status_t st) {
2     return (LEDS_ACTIVE_STATE ? (st == LED_ON) : (st == LED_OFF));
3 }
```

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:

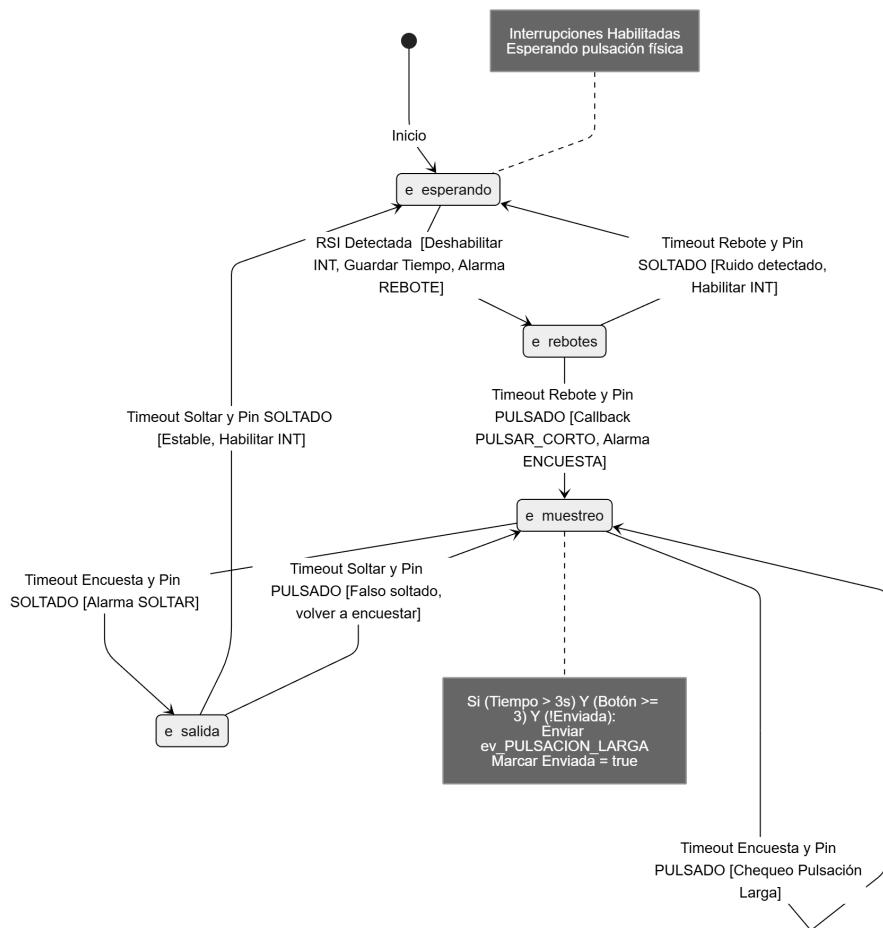


Figure 1: Button driver state machine with debouncing

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:

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

nRF52840: Use of the GPIOTE SENSE/LATCH mechanism:

```
1 void hal_ext_int_habilitar(uint8_t id_boton) {
2     // SENSE_Low -> generates PORT event when pin=0
3     cnf |= (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos);
4 }
```

3.4.4. GPIO Monitors

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

```
1 void drv_monitor_marcar(uint32_t id);      // pin -> HIGH
2 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:

```
1 void rt_GE_lanzador() {
2     while(1) {
3         drv_WDT_alimentar(); // prevents reset
4         // ... event processing
5     }
6 }
```

3.6. Concurrency and Event Model

3.6.1. Event Definition

Events are defined in `rt_evento_t.h`:

```
1 typedef enum {
2     ev_VOID = 0,           // null event
3     ev_T_PERIODICO = 1,    // timer tick
4     ev_BOTON_PULSAR = 2,   // button pressed
5     ev_INACTIVIDAD = 3,    // inactivity timeout
6     ev_BOTON_RETARDO = 4,   // debounce handling
7     ev_JUEGO_TICK = 5,     // game timing
8     ev_PULSACION_LARGA = 6 // press >=3s
9 } EVENTO_T;
10
11 #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:

```
1 if (cola.contador == RT_FIFO_CAPACITY) {
2     while(true) { /* overflow -> diagnostics */ }
3 }
```

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`):

```
1 rt_sc_estado_t rt_sc_entrar(void); // disables IRQs, saves state
2 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	<code>drv_consumo_esperar()</code>	WFI, wakes on IRQ
Sleep	<code>drv_consumo_dormir()</code>	System OFF, requires reset

Table 5: System power modes

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

Busy-wait:

```
1 delay = 1000000;
2
3 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:

```

1 void hal_consumo_dormir(void) {
2     NRF_POWER->SYSTEMOFF = POWER_SYSTEMOFF_SYSTEMOFF_Enter;
3     __WFE();
4     while (1) { } // does not return
5 }
```

LPC2105:

```

1 void hal_consumo_dormir(void) {
2     EXTWAKE = 7;           // EINT0,1,2 as wake-up sources
3     PCON |= 0x02;         // Power-down
4     Reset_Handler();     // restart after wake-up
5 }
```

3.7.3. Wake-Up by Button

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

```

1 static void preparar_despertar_por boton(void) {
2     hal_ext_int_iniciar();
3     for (uint8_t i = 1; i <= BUTTONS_NUMBER; ++i) {
4         hal_ext_int_habilitar(i); // SENSE=LOW
5     }
6 }
```

3.7.4. Inactivity Timeout

The system schedules a 20-second inactivity alarm:

```

1 void rt_GE_lanzador() {
2     svc_alarma_activar(
3         svc_alarma_codificar(0, 20*1000, 0),
4         ev_INACTIVIDAD, 0
5     );
6     // ... event loop
7 }
```

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:

```

1 void svc_alarma_iniciar(uint32_t monitor_overflow,
2                           void (*cb_encolar)(uint32_t, uint32_t),
3                           EVENTO_T ev_tick);
4
5 void svc_alarma_activar(svc_alarma_flags_t flags,
6                         EVENTO_T id_evento,
7                         uint32_t auxData);
8
9 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:

```

1 void rt_GE_lanzador() {
2     while(1) {
3         drv_WDT_alimentar();
4
5         if (rt_FIFO_extraer(&id_ev, &aux, &ts) != (uint8_t)-1) {
6             // Latency statistics
7             Tiempo_us_t delta = drv_tiempo_actual_us() - ts;
8             // ...
9
10            // Invoke subscribed callbacks
11            for (uint8_t i = 0; i < ge_tabla[id_ev].num_suscritos; i++) {
12                ge_tabla[id_ev].lista_callbacks[i](id_ev, aux);
13            }
14        } else {
15            drv_consumo_esperar(); // Empty queue -> WFI
16        }
17    }
18}

```

3.8.3. Subscription and Cancellation

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

```

1 void svc_GE_suscribir(EVENTO_T id_evento,
2                       uint8_t prioridad,
3                       rt_GE_callback_t callback);
4
5 void svc_GE_cancelar(EVENTO_T id_evento,
6                      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

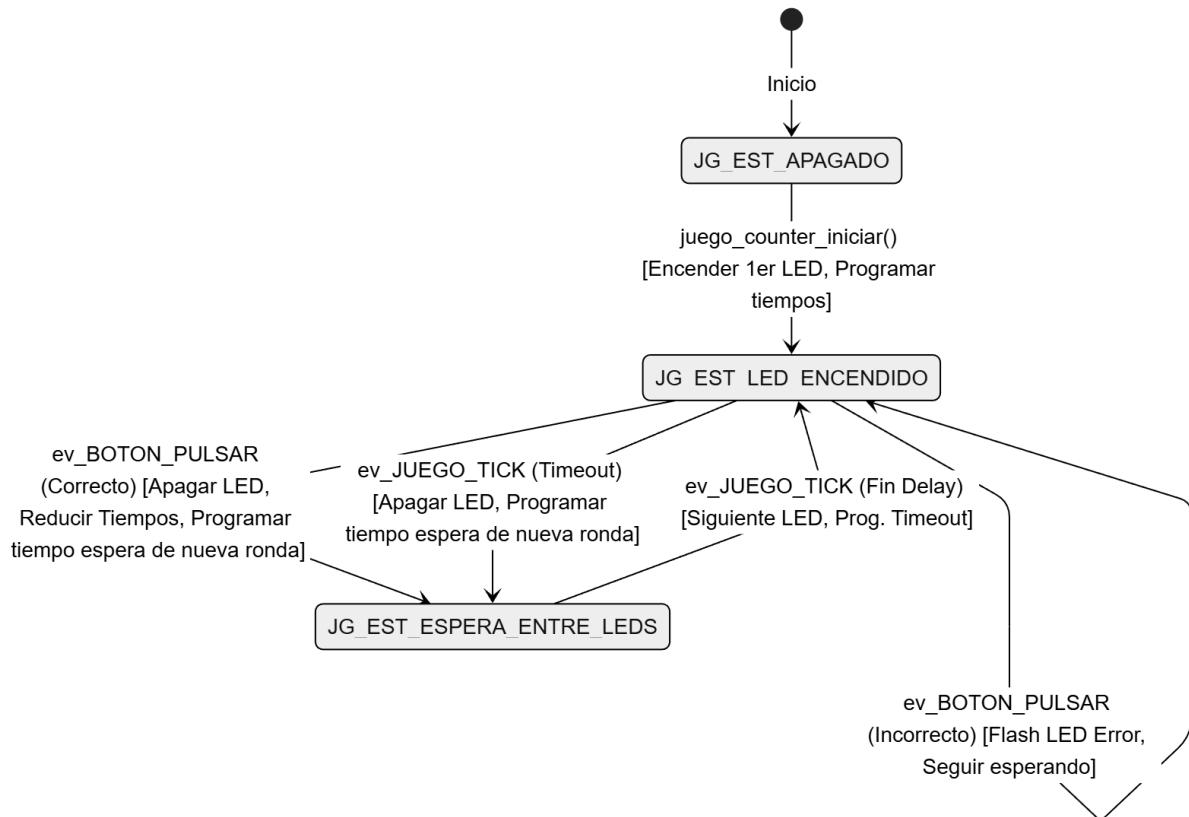


Figure 2: State diagram of the Bit Counter Strike game

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:

```

1 svc_GE_suscribir(ev_BOTON_PULSAR, 2, juego_counter_actualizar);
2 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

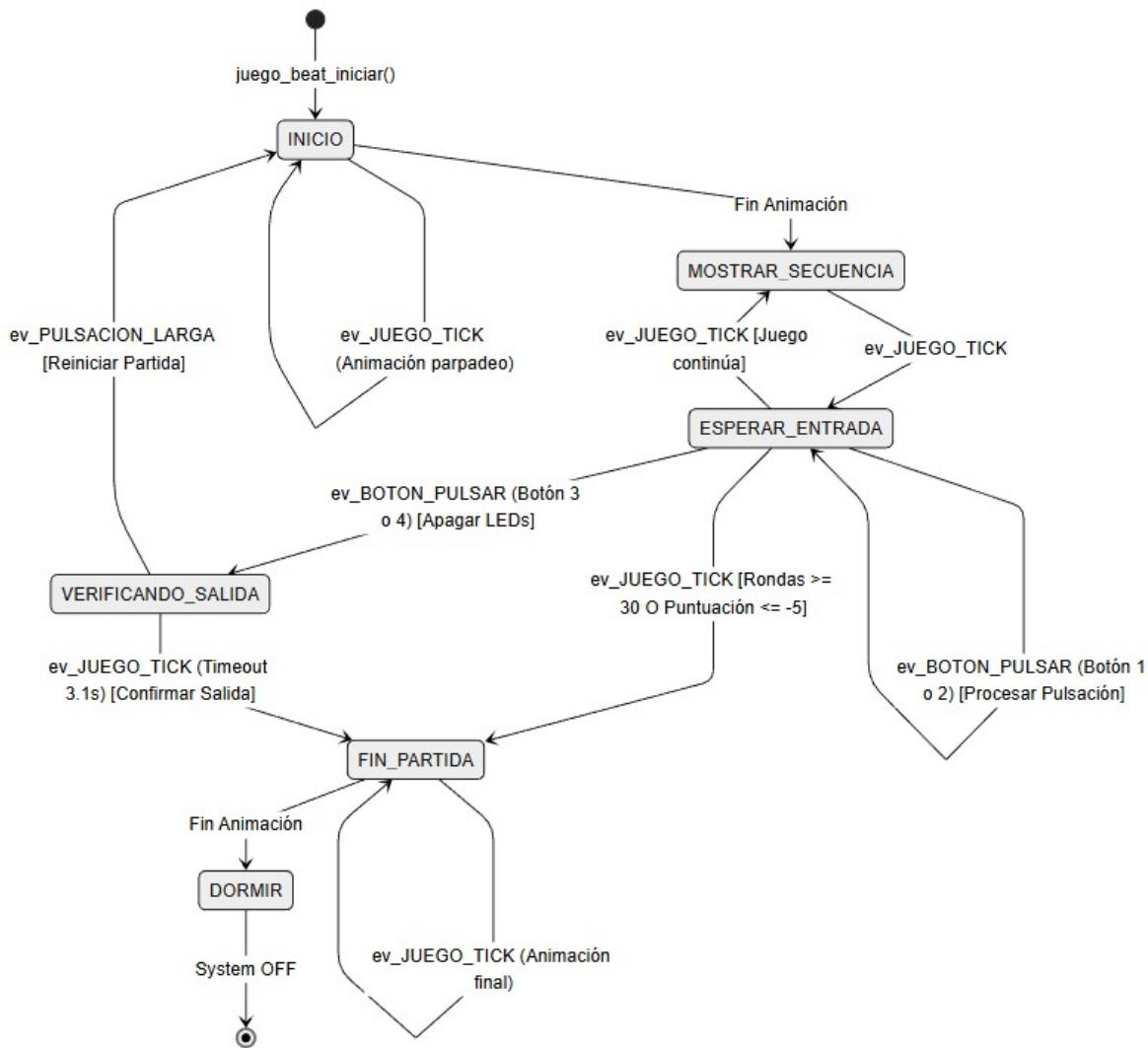


Figure 3: State diagram of the Beat Hero game

4.2.3. Scoring System

The score depends on button press timing accuracy:

Accuracy	Points	Condition
Excellent	+2	$\leq 10\%$ of the bar period
Good	+1	$\leq 20\%$ of the period
Acceptable	0	$\leq 40\%$ (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

```

1 static void actualizar_nivel_y_bpm(void) {
2     if ((j.compases_jugados % JUEGO_COMPASES_POR_NIVEL) == 0
3         && j.nivel < JUEGO_NIVEL_MAX)
4         j.nivel++;
5
6     if (j.nivel == JUEGO_NIVEL_MAX)
7         j.bpm += 2;
8
9     j.t_compas_ms = 60000 / j.bpm;
10    j.ventana_ms = (j.t_compas_ms * JUEGO_VENTANA_PORC) / 100;
11 }
```

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 (≥ 3 s) on button 3 or 4 resets the session:

```

1 if (evento == ev_PULSACION_LARGA && j.estado != DORMIR) {
2     juego_beat_iniciar();
3     return;
4 }
```

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 SESSION set to 5, an explicit test can be executed:

```

1 static void test_cola_overflow() {
2     rt_FIFO_inicializar(4);
3
4     __disable_irq();
5     for (int i = 0; i < 32; ++i) rt_FIFO_encolar(ev_VOID, 0);
6     __enable_irq();
7
8     int fin = 1; // must be reached (32 OK, no overflow).
9
10    // Intentional overflow
11    rt_FIFO_encolar(ev_VOID, 1); // must enter infinite loop due to overflow
12
13    fin = -1;      // must never reach here
14 }
```

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:

```

1 case VERIFICANDO_SALIDA:
2     // 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).

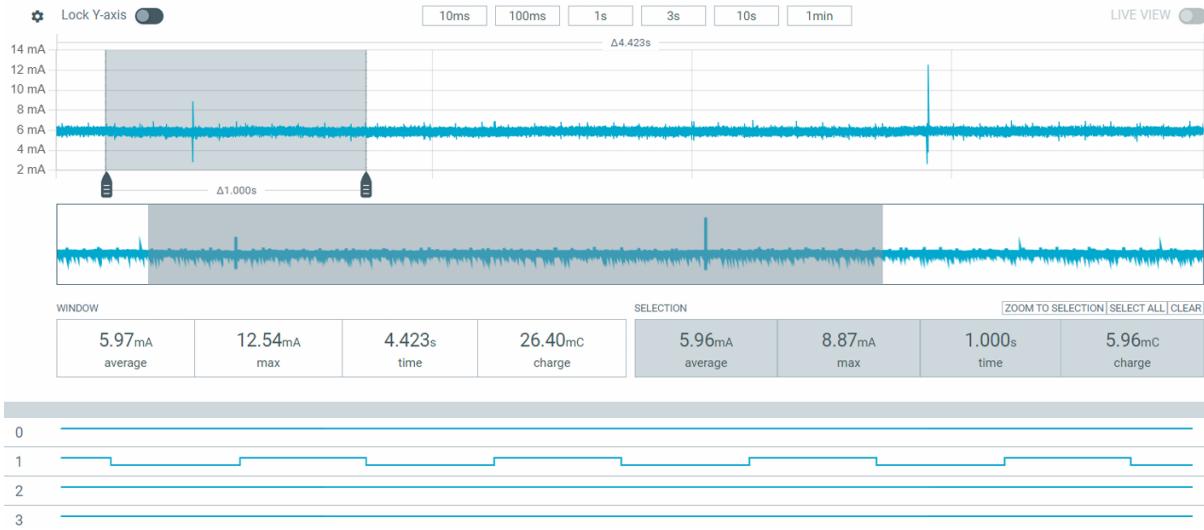


Figure 4: Power measurement — Blink v2 (busy-wait)

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

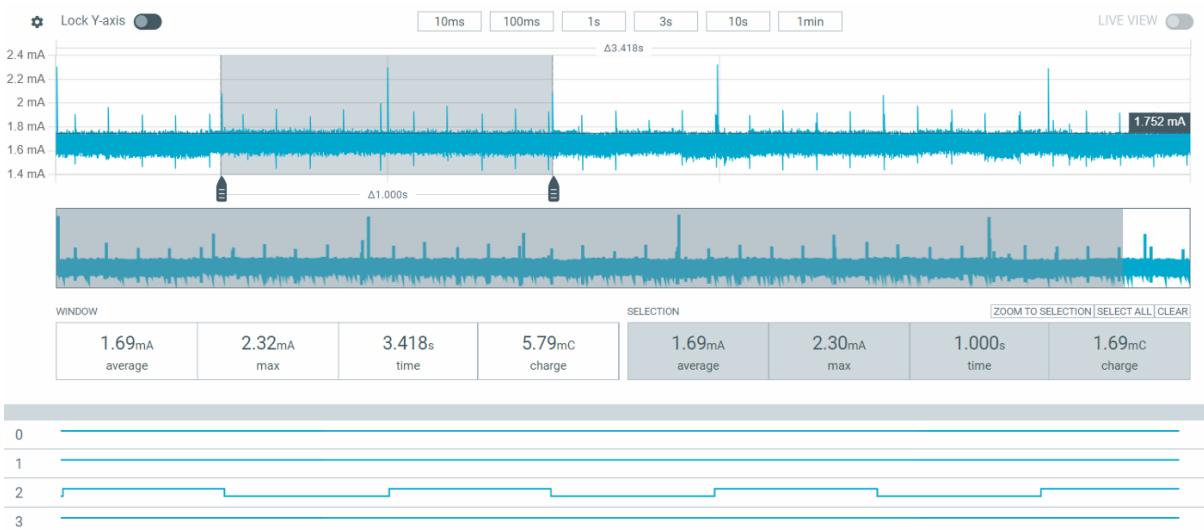


Figure 5: Power measurement — Blink v3 (low power with WFI)

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

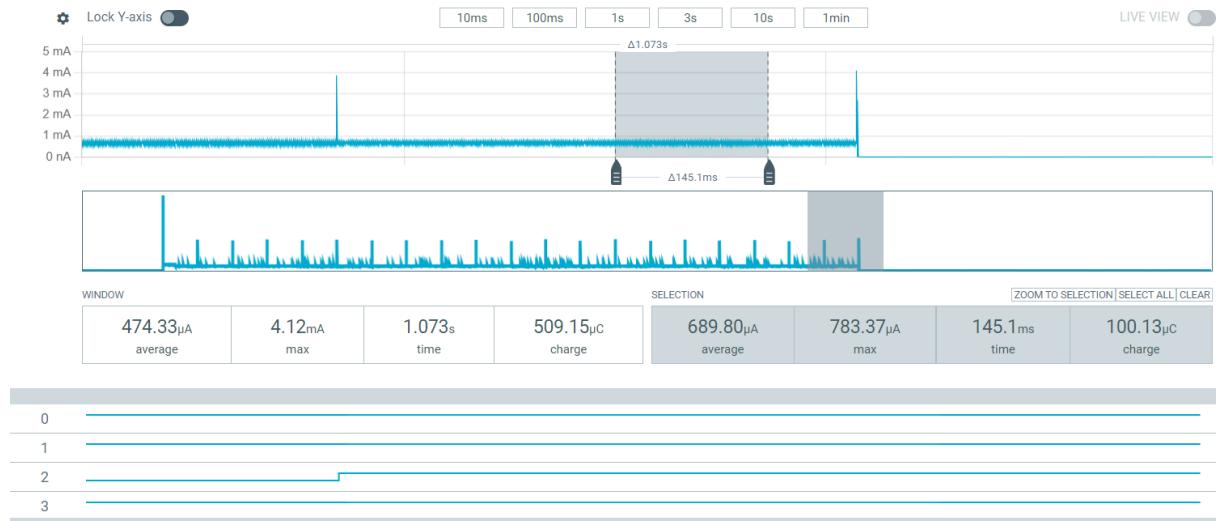


Figure 6: Power measurement — Blink v3 bis (awake mode)

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

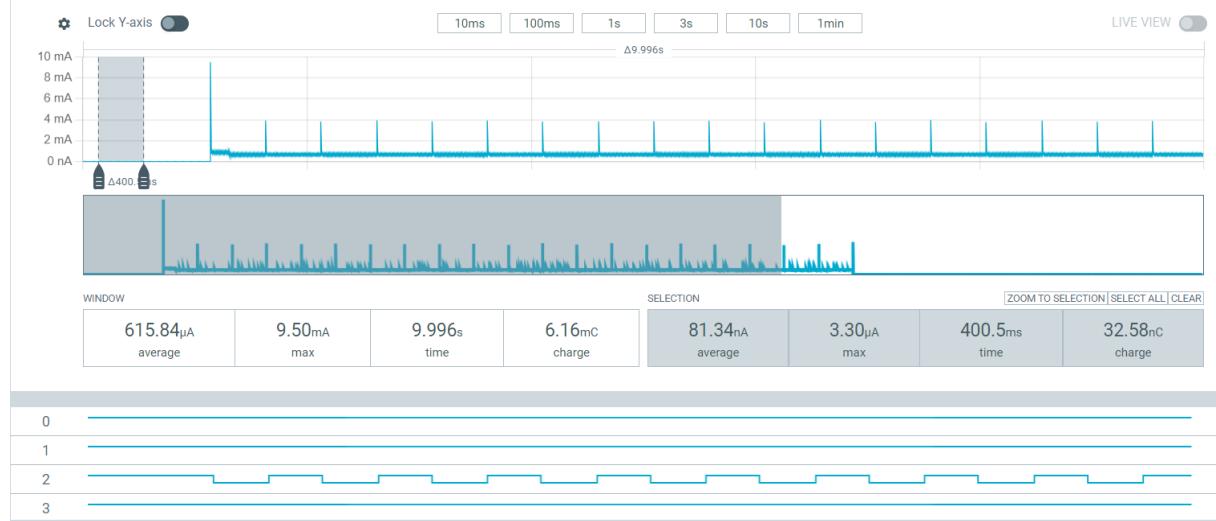


Figure 7: Power measurement — Blink v3 bis (sleep/System OFF)

- Beat Hero:

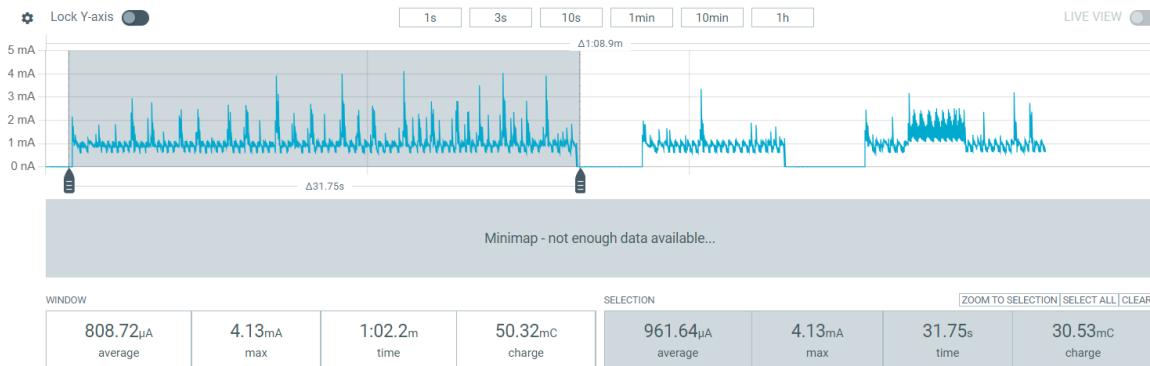


Figure 8: Power measurement — Beat Hero during gameplay

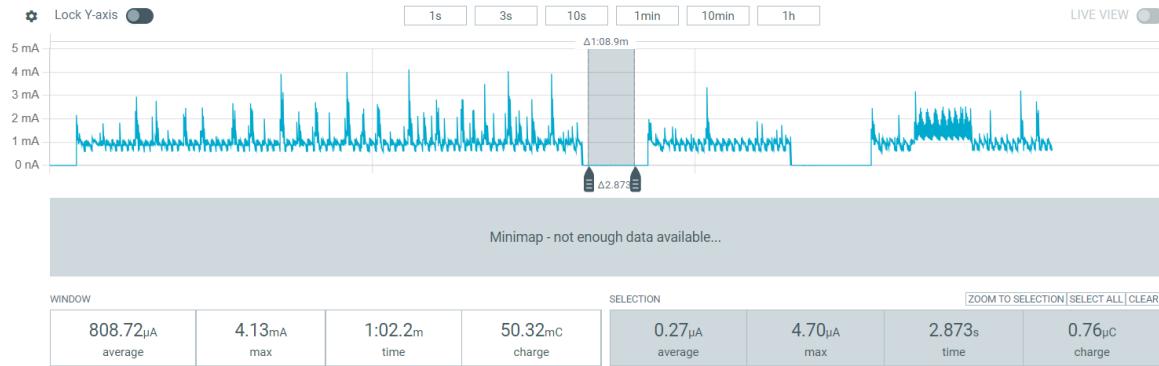


Figure 9: Power measurement — Beat Hero with the CPU sleeping

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	$\sim 962 \mu\text{A}$
CPU sleeping	$\sim 0.27 \mu\text{A}$

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/datasheets/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:

```
1 uint32_t total_eventos = rt_FIFO_estadisticas(ev_VOID);
2 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)

```

1 #include "juego_beat.h"
2 #include "drv_leds.h"
3 #include "svc_alarmas.h"
4 #include "drv_tiempo.h"
5 #include "board.h"
6 #include "drv_consumo.h"
7 #include "drv_random.h"
8 #include <stdint.h>
9 #include <stdbool.h>
10 #include <stdlib.h>
11
12 // =====
13 // CONSTANTS
14 // =====
15 #define JUEGO_BPM_INICIAL 60
16 #define JUEGO_COMPASES_MAX 30
17 #define JUEGO_PUNTOS_MIN -5
18 #define JUEGO_VENTANA_PORC 40
19 #define JUEGO_PRECISION_10 10
20 #define JUEGO_PRECISION_20 20
21 #define JUEGO_COMPASES POR_NIVEL 7
22 #define JUEGO_TIMEOUT_INACTIVIDAD_MS 10000
23 #define JUEGO_NIVEL_MAX 4
24
25 #define ANIM_PARPADEO_MS 500
26 #define ANIM_FIN_MS 300
27
28 // ----- Internal state -----
29 typedef enum {
30     INICIO = 0,
31     MOSTRAR_SECUENCIA = 1,
32     ESPERAR_ENTRADA = 2,
33     VERIFICANDO_SALIDA = 3,
34     FIN_PARTIDA = 4,
35     DORMIR = 5
36 } juego_estado_t;
37
38 typedef struct {
39     juego_estado_t estado;
40     uint8_t compas[3];
41     uint8_t nivel;
42     uint8_t bpm;
43     uint32_t t_compas_ms;
44     uint32_t ventana_ms;
45
46     int8_t compases_jugados;
47
48     uint32_t timestamp_inicio;
49     uint8_t patron_recibido;
50     bool primera_pulsacion;
51     uint32_t timestamp_primera_pulsacion;
52
53     uint8_t paso_animacion;
54     bool anim_exito;
55
56     //---- Statistics ----
57     int32_t puntuacion;
58     uint8_t puntuacion_por_compas;
59     uint8_t puntuacion_por_nivel[4];
60     uint32_t tiempo_reaccion_total;
61     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,
71         LED_OFF);
72 }
73
74 static void juego_encender.todos(void){
75     for (uint8_t i = 1; i <= LEDS_NUMBER; ++i) drv_led_establecer((LED_id_t)i,
76         LED_ON);
77 }
78
79 static void juego_reprogramar(uint32_t retardo_ms){
80     svc_alarma_activar(svc_alarma_codificar(0, retardo_ms, 0), ev_JUEGO_TICK, 0)
81     ;
82 }
83
84 static void juego_resetear_inactividad(void) {
85     svc_alarma_activar(svc_alarma_codificar(0, JUEGO_TIMEOUT_INACTIVIDAD_MS, 0),
86     ev_INACTIVIDAD, 0);
87 }
88
89 // ----- Logic -----
90
91 static uint8_t generar_compas(void){
92     static uint8_t ultimo_generado = 0;
93     uint32_t r;
94     uint8_t resultado = 0;
95
96     do {
97         r = drv_random_generar_rango(4); // Generates 0, 1, 2, 3
98
99         switch (j.nivel) {
100             case 1:
101                 // Level 1: ONLY one LED ON (1 or 2).
102                 // No empty bars (0) and no double bars (3) are allowed.
103                 if (r == 0) resultado = 1;
104                 else if (r == 3) resultado = 2;
105                 else resultado = (uint8_t)r;
106                 break;
107
108             case 2:
109                 // Level 2: empty bars (0) are allowed.
110                 // Double bars (3) are NOT allowed.
111                 if (r == 3) resultado = 0;
112                 else resultado = (uint8_t)r;
113                 break;
114
115         default:
116             // Levels 3 and 4: allow two LEDs (3).
117             resultado = (uint8_t)r;
118             break;
119         }
120
121         // In level 1, enforce alternation for visual clarity.
122         if (j.nivel == 1 && ultimo_generado != 0 && resultado == ultimo_generado)
123             {
124                 // If repeated in level 1, force the opposite pattern.
125             }
126     }
127 }
```

```

120         resultado = (resultado == 1) ? 2 : 1;
121     }
122
123 } while (resultado == ultimo_generado && j.nivel > 1); // Avoid exact
   repetition at higher levels.
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     // TOP (LEDs 1 and 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     // BOTTOM (LEDs 3 and 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) { // If the player did not press anything...
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     // Outside the window (>40%): -1 point.
166     if (tiempo > j.ventana_ms) {
167         j.puntuacion -= 1;
168         return;
169     }
170
171     // Incorrect pattern: -1 point.
172     uint8_t esperado = j.compas[0];
173     if (j.patron_recibido != esperado) {
174         j.puntuacion -= 1;
175         return;
176     }
177
178     // Compute accuracy.
179     uint32_t porcentaje = (tiempo * 100) / j.t_compas_ms;
180
181     // Point assignment:

```

```

182     if (porcentaje <= JUEGO_PRECISION_10) {
183         j.puntuacion += 2; // +2 if within 10%
184     }
185     else if (porcentaje <= JUEGO_PRECISION_20) {
186         j.puntuacion += 1; // +1 if within 20%
187     }
188     // 0 points if between 20% and 40% (no change).
189
190     // Update statistics
191     j.puntuacion_por_nivel[j.nivel-1] += j.puntuacion;
192     if (j.compases_jugados > 0) j.puntuacion_por_compas = j.puntuacion / j.
193         compases_jugados;
194     j.tiempo_reaccion_total += tiempo;
195     if (j.compases_jugados > 0) j.tiempo_reaccion_medio_ms = j.
196         tiempo_reaccion_total / j.compases_jugados;
197     if (tiempo < j.tiempo_reaccion_mas_rapido_ms) j.
198         tiempo_reaccion_mas_rapido_ms = tiempo;
199 }
200
201 static void actualizar_nivel_y_bpm(void){
202     // If all bars for the current level have been completed, advance to the
203     // next level (if any remain).
204     if ((j.compases_jugados % JUEGO_COMPASES_POR_NIVEL) == 0 && j.nivel <
205         JUEGO_NIVEL_MAX)
206         j.nivel++;
207
208     if (j.nivel == JUEGO_NIVEL_MAX) j.bpm += 2; // Increase difficulty.
209
210     j.t_compas_ms = 60000 / j.bpm;
211     j.ventana_ms = (j.t_compas_ms * JUEGO_VENTANA_PORC) / 100;
212 }
213
214 // ----- Public interface -----
215
216 void juego_beat_iniciar(void){
217     juego_apagar_todos();
218     j.nivel = 1;
219     j.bpm = JUEGO_BPM_INICIAL;
220     j.puntuacion = 0;
221     j.puntuacion_por_compas = 0;
222     for (uint8_t i=0; i<JUEGO_NIVEL_MAX; i++) j.puntuacion_por_nivel[i] = 0;
223     j.tiempo_reaccion_total = 0;
224     j.tiempo_reaccion_medio_ms = 0;
225     // The reaction time cannot be larger than the window; otherwise a point
226     // would be lost.
227     j.tiempo_reaccion_mas_rapido_ms = j.ventana_ms;
228     // Starts at -1 because the first "pixel" must drop to the bottom before the
229     // game begins.
230     j.compases_jugados = -1;
231     j.t_compas_ms = 60000/ j.bpm;
232     j.patron_recibido = 0;
233     j.primera_pulsacion = false;
234     j.timestamp_primera_pulsacion = 0;
235
236     j.compas[0] = 0;
237     j.compas[1] = 0;
238     j.compas[2] = generar_compas();
239
240     juego_resetear_inactividad();
241
242     j.paso_animacion = 0;
243     j.estado = INICIO;

```

```

238     juego_encender.todos();
239     juego_reprogramar(ANIM_PARPADEO_MS);
240 }
241
242 void juego_beat_actualizar(EVENTO_T evento, uint32_t auxData){
243
244     // If the inactivity period expires, enter sleep.
245     if (evento == ev_INACTIVIDAD && j.estado != DORMIR) {
246         juego_apagar.todos();
247         j.estado = DORMIR;
248         juego_reprogramar(1);
249         return;
250     }
251     // If a long press is detected, start a new session.
252     if (evento == ev_PULSACION_LARGA && j.estado != DORMIR) {
253         juego_beat_iniciar();
254         return;
255     }
256
257     switch (j.estado) {
258         case INICIO:
259             if (evento == ev_JUEGO_TICK) {
260                 // Play start-of-game animation.
261                 j.paso_animacion++;
262                 switch(j.paso_animacion) {
263                     case 1: juego_apagar.todos(); juego_reprogramar(
264                             ANIM_PARPADEO_MS); break;
265                     case 2: juego_encender.todos(); juego_reprogramar(
266                             ANIM_PARPADEO_MS); break;
267                     case 3: juego_apagar.todos(); juego_reprogramar(200);
268                         break;
269                     default:
270                         // Once the animation ends, transition to
271                         // MOSTRAR_SECUENCIA.
272                         j.estado = MOSTRAR_SECUENCIA;
273                         juego_reprogramar(1);
274                         break;
275                 }
276             }
277             break;
278
279         case MOSTRAR_SECUENCIA:
280             if (evento == ev_JUEGO_TICK) {
281                 // Show the LED sequence to the player and then wait for input.
282                 mostrar_compases();
283                 j.timestamp_inicio = drv_tiempo_actual_ms();
284                 j.patron_recibido = 0;
285                 j.primera_pulsacion = false;
286                 juego_reprogramar(j.t_compas_ms());
287                 j.estado = ESPERAR_ENTRADA;
288             }
289             break;
290
291         case ESPERAR_ENTRADA:
292             if (evento == ev_BOTON_PULSAR) {
293                 // Reset inactivity timer on any press.
294                 juego_resetear_inactividad();
295                 // If button 3 or 4 is pressed, the player is requesting exit (
296                 // or restart),
297                 // so transition to VERIFICANDO_SALIDA.
298                 if (auxData >= 3) { // Exit
299                     juego_apagar.todos();
300                     j.estado = VERIFICANDO_SALIDA;
301                 }
302             }
303         }
304     }
305 }
```

```

296         // >3 seconds so that there is time to trigger the restart
297         long press.
298         juego_reprogramar(3100);
299         return;
300     }
301     // Otherwise, record the input pattern for later evaluation.
302     if (auxData == 1 || auxData == 2) procesar_pulsacion(auxData);
303
304 } else if (evento == ev_JUEGO_TICK) {
305     // When the bar period expires, compute score and statistics.
306     j.compases_jugados++;
307     evaluar_ronda();
308     avanzar_compas();
309     // Update level if needed.
310     actualizar_nivel_y_bpm();
311
312     // End the game if max bars reached or score too low.
313     // (Comment out "|| j.puntuacion <= JUEGO_PUNTOS_MIN" to avoid
314     // losing by score and
315     // observe inactivity-based sleep instead.)
316     if (j.compases_jugados >= JUEGO_COMPASES_MAX || j.puntuacion <=
317         JUEGO_PUNTOS_MIN) {
318         j.estado = FIN_PARTIDA;
319
320         if (j.compases_jugados >= JUEGO_COMPASES_MAX) {
321             j.anim_exito = true; // Win.
322         } else {
323             j.anim_exito = false; // Lose.
324         }
325         // Prepare end-of-game animation.
326         j.paso_animacion = 0;
327         juego_reprogramar(1);
328     } else {
329         // Otherwise proceed to the next bar/round.
330         j.estado = MOSTRAR_SECUENCIA;
331         juego_reprogramar(1);
332     }
333     break;
334
335 case VERIFICANDO_SALIDA:
336     // while(1); // uncomment to verify watchdog
337
338     // If a long-press event arrives: RESTART.
339     if (evento == ev_PULSACION_LARGA) {
340         juego_beat_iniciar();
341     }
342     // If the 3100 ms timeout expires and no long press occurred: end
343     // the game.
344     else if (evento == ev_JUEGO_TICK) {
345         j.estado = FIN_PARTIDA;
346         j.paso_animacion = 0;
347         juego_reprogramar(1);
348     }
349     break;
350
351 case FIN_PARTIDA:
352     if (evento == ev_JUEGO_TICK) {
353         // Show animation.
354         j.paso_animacion++;
355         if (j.paso_animacion < 8) {
356             if (j.paso_animacion % 2 != 0) {
357                 // Select animation depending on outcome.

```

```
355         if (j.anim_exito) {
356             juego_encender_todos(); // VICTORY: all LEDs on.
357         } else {
358             // DEFEAT: cross using LEDs 1 and 4.
359             juego_apagar_todos();
360             drv_led_establecer(1, LED_ON);
361             drv_led_establecer(4, LED_ON);
362         }
363     }
364     else juego_apagar_todos();
365
366     juego_reprogramar(ANIM_FIN_MS);
367 } else {
368     // When the animation ends, transition to sleep.
369     j.estado = DORMIR;
370     juego_reprogramar(1);
371 }
372 }
373 break;
374
375 case DORMIR:
376     juego_apagar_todos();
377     drv_consumo_dormir();
378     break;
379 }
380 }
```

Appendix C: Event Manager (rt_GE.c)

```

1 #include "rt_fifo.h"
2 #include "drv_tiempo.h"
3 #include "drv_monitor.h"
4 #include "drv_consumo.h"
5 #include "rt_GE.h"
6 #include "svc_GE.h"
7 #include "rt_evento_t.h"
8 #include "svc_alarmas.h"
9 #include "board.h"
10 #include "hal_ext_int.h"
11 #include "drv_WDT.h"

12
13
14 // Data structure to store subscriptions
15 uint32_t rt_monitor_overflow;
16 GE_suscripcion_t ge_tabla[EVENT_TYPES];
17 Tiempo_us_t tiempo_max_sin_tratar;
18 uint32_t eventos_tratados;
19 Tiempo_us_t tiempo_sin_tratar;
20 Tiempo_us_t media_tiempo_sin_tratar;
21
22
23
24 void rt_GE_iniciar(uint32_t monitor_overflow){
25     tiempo_max_sin_tratar = 0;
26     eventos_tratados = 0;
27     media_tiempo_sin_tratar = 0;
28     tiempo_sin_tratar = 0;
29     rt_monitor_overflow = monitor_overflow;
30     for (uint8_t i = 0; i < EVENT_TYPES; i++) {
31         ge_tabla[i].num_suscritos = 0;
32         for (uint8_t j = 0; j < rt_GE_MAX_SUSCRITOS; j++) {
33             ge_tabla[i].lista_callbacks[j] = NULL;
34         }
35     }
36 }
37
38
39
40 void rt_GE_lanzador(){
41
42     // Inactivity alarm after 20 seconds with no activity.
43     svc_alarma_activar(svc_alarma_codificar(0,20*1000,0), ev_INACTIVIDAD, 0);
44     EVENTO_T id_ev;
45     uint32_t aux;
46     Tiempo_us_t ts;
47     while(1){

48         drv_WDT_alimentar(); // Feed the watchdog
49
50         // If there are events in the FIFO...
51         if(rt_FIFO_extraer(&id_ev, &aux, &ts) != (uint8_t)-1{
52             // Update statistics
53             Tiempo_us_t ahora = drv_tiempo_actual_us();
54             Tiempo_us_t delta = ahora - ts;
55             eventos_tratados++;
56             tiempo_sin_tratar += delta;
57             media_tiempo_sin_tratar = tiempo_sin_tratar / eventos_tratados;
58             if (delta > tiempo_max_sin_tratar) tiempo_max_sin_tratar = delta;
59
60             // Walk the subscription list and invoke callbacks
61         }
62     }
63 }
```

```

62     if ((uint8_t)id_ev < EVENT_TYPES) {
63         for (uint8_t i = 0; i < ge_tabla[(uint8_t)id_ev].num_suscritos;
64             i++) {
65             if (ge_tabla[(uint8_t)id_ev].lista_callbacks[i] != NULL) {
66                 ge_tabla[(uint8_t)id_ev].lista_callbacks[i](id_ev, aux);
67             }
68         }
69     } else{
70         // Otherwise, wait
71         drv_consumo_esperar();
72     }
73 }
74
75 }
76
77 // Helper
78 static void preparar_despertar_inactividad(void){
79 #if BUTTONS_NUMBER > 0
80     for (uint8_t i = 1; i <= BUTTONS_NUMBER; ++i) {
81         hal_ext_int_habilitar(i);    // SENSE=LOW on ALL buttons
82     }
83 #endif
84 }
85
86
87 static uint32_t contador_alarma = 0;
88
89 // Re-arm
90 void rt_GE_actualizar(EVENTO_T ID_EVENTO, uint32_t auxData){
91
92     switch((uint8_t)ID_EVENTO){
93         case ev_INACTIVIDAD:
94             preparar_despertar_inactividad();
95             drv_consumo_dormir();
96             break;
97
98         case ev_BOTON_PULSAR:
99             contador_alarma++;
100            svc_alarma_activar(
101                svc_alarma_codificar(0, 20000, 0),
102                ev_INACTIVIDAD,
103                contador_alarma
104            );
105            break;
106
107        default:
108            break;
109    }
110 }
```

Appendix D: FIFO Queue (rt_fifo.c)

```

1 #include "rt_fifo.h"
2 #include "drv_tiempo.h"
3 #include "drv_monitor.h"
4 #include "rt_sc.h"
5
6 // Queue parameters.
7 #ifndef RT_FIFO_CAPACITY
8 #define RT_FIFO_CAPACITY 32
9 #endif
10
11
12 // Internal state.
13 typedef struct {
14     EVENTO eventos_encolados[RT_FIFO_CAPACITY];
15     volatile uint32_t head;           // next free position to write
16     volatile uint32_t tail;          // next position with data to read (
17         main)
18     uint32_t estadisticas[EVENT_TYPES]; // per-event-type counters
19     uint32_t total;                // historical total enqueue
20     uint32_t mon_overflow;         // monitor to assert on overflow (0 =>
21         none)
22     volatile uint32_t contador;      // current number of enqueued elements
23         : 0..RT_FIFO_CAPACITY
24 } fifo_t;
25
26
27 static fifo_t cola;
28
29
30 void rt_FIFO_inicializar(uint32_t monitor_overflow) {
31     cola.head = 0;
32     cola.tail = 0;
33     cola.contador = 0;
34     for (uint32_t i = 0; i < EVENT_TYPES; ++i) cola.estadisticas[i] = 0;
35     cola.total = 0;
36     cola.mon_overflow = monitor_overflow;
37 }
38
39
40 void rt_FIFO_encolar(uint32_t ID_evento, uint32_t auxData) {
41     // Enter critical section
42     rt_sc_estado_t sc = rt_sc_entrar();
43
44     if (cola.contador == RT_FIFO_CAPACITY) {
45         if (cola.mon_overflow) drv_monitor_marcar(cola.mon_overflow);
46         while(true){ /* overflow -> infinite loop for diagnostics */ }
47     }
48
49     uint32_t pos = cola.head;
50     cola.eventos_encolados[pos].ID_EVENTO = (EVENTO_T)ID_evento;
51     cola.eventos_encolados[pos].auxData = auxData;
52     cola.eventos_encolados[pos].TS = drv_tiempo_actual_us();
53
54     cola.head = (pos + 1) % RT_FIFO_CAPACITY;
55     cola.contador++;
56
57     // Statistics
58     if (ID_evento < EVENT_TYPES) cola.estadisticas[ID_evento]++;
59     cola.total++;
60 }
```

```
59     // Exit critical section
60     rt_sc_salir(sc);
61 }
62
63
64 uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t *auxData, Tiempo_us_t *TS)
65 {
66     // Enter critical section
67     rt_sc_estado_t sc = rt_sc_entrar();
68
69     uint32_t t = cola.tail;
70     if (cola.contador == 0){
71         rt_sc_salir(sc);
72         return (uint8_t)-1;
73     }
74
75     if (ID_evento) *ID_evento = cola.eventos_encolados[t].ID_EVENTO;
76     if (auxData) *auxData = cola.eventos_encolados[t].auxData;
77     if (TS) *TS = cola.eventos_encolados[t].TS;
78
79     cola.tail = (t + 1) % RT_FIFO_CAPACITY;
80     cola.contador--;
81
82     // Return number of events still pending after extraction.
83     uint32_t pendientes = cola.contador;
84     if (pendientes > 254) pendientes = 254;
85
86     // Exit critical section
87     rt_sc_salir(sc);
88
89     return (uint8_t)pendientes;
90 }
91
92
93 uint32_t rt_FIFO_estadisticas(EVENTO_T ID_evento) {
94     if (ID_evento == ev_VOID) return cola.total;
95     if ((uint32_t)ID_evento < EVENT_TYPES) return cola.estadisticas[(uint32_t)
96         ID_evento];
97     return 0;
}
```

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

```

1 #include "hal_random.h"
2 #include <LPC210x.H>
3
4 /*
5  * We need access to a time counter to "simulate" entropy.
6  * It is assumed that hal_tiempo has initialized Timer 1.
7  * Otherwise, a fixed value will be returned and the system will rely on manual
8  * seeding in main.
9 */
10
11 void hal_random_iniciar(void) {
12     // The LPC2105 provides no specific hardware RNG that needs initialization.
13 }
14
15 uint8_t hal_random_generar_u8(void) {
16
17     // Use the current Timer 1 value (T1TC) as a source of "noise".
18     uint32_t entropia_simulada = T1TC;
19
20     // Take the low byte.
21     return (uint8_t)(entropia_simulada & 0xFF);
22 }
```

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

```

1 #include "hal_random.h"
2 #include <nrf.h>
3
4 void hal_random_iniciar(void) {
5     /*
6      * Configure the random number generator.
7      * Enable bias correction.
8      */
9     NRF_RNG->CONFIG = RNG_CONFIG_DERCEN_Enabled;
10
11    // Start the task.
12    NRF_RNG->TASKS_START = 1;
13 }
14
15 uint8_t hal_random_generar_u8(void) {
16
17     // Clear the previous event for safety.
18     NRF_RNG->EVENTS_VALRDY = 0;
19
20     // Wait until a value is ready. Latency is nondeterministic but fast for 1
21     // byte.
22     while (NRF_RNG->EVENTS_VALRDY == 0);
23
24     // Clear the event for the next read.
25     NRF_RNG->EVENTS_VALRDY = 0;
26
27     // Read from the VALUE register.
28     return (uint8_t)(NRF_RNG->VALUE);
29 }
```

Appendix G: Critical Sections (LPC) (rt_sc_lpc.c)

```

1 #include <LPC210x.H>
2 #include <stdint.h>
3 #include "rt_sc.h"
4
5
6 rt_sc_estado_t rt_sc_entrar(void){
7     __disable_irq();
8
9     // For this lab we do not use the previous state value, so we return 0.
10    return 0;
11 }
12
13 void rt_sc_salir(rt_sc_estado_t estado_prev){
14     (void)estado_prev; // Unused in this simplified version.
15     __enable_irq();
16 }
```

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

```

1 #include <stdint.h>
2 #include <nrf.h>
3 #include "rt_sc.h"
4
5 /*
6  * - PRIMASK = 1 => IRQs disabled
7  * - PRIMASK = 0 => IRQs enabled
8 */
9
10 rt_sc_estado_t rt_sc_entrar(void)
11 {
12     // Save the previous state.
13     uint32_t primask = __get_PRIMASK();
14     // Disable interrupts.
15     __disable_irq();
16     return primask;
17 }
18
19 void rt_sc_salir(rt_sc_estado_t estado_prev)
20 {
21     // Restore the exact previous state.
22     __set_PRIMASK(estado_prev);
23 }
```

Appendix I: Button Driver (drv_botones.c)

```

1 #include "drv_botones.h"
2 #include "svc_GE.h"
3 #include "board.h"
4 #include "hal_ext_int.h"
5 #include "svc_alarmas.h"
6 #include "drv_tiempo.h"
7
8 #define RETARDO_REBOTE_MS 50      // Press debounce time
9 #define RETARDO_SOLTAR_MS 50      // Release debounce time
10 #define RETARDO_ENCUESTA_MS 20    // Periodic polling interval (check if still
11     pressed)
12 #define PULSACION_LARGA 3000     // 3 seconds as long-press threshold
13
14 typedef enum {
15     e_esperando,
16     e_salida,
17     e_rebotes,
18     e_muestreo,
19 } boton_estado_t;
20
21 typedef struct {
22     boton_estado_t estado;
23     uint8_t pin;
24     uint32_t inicio_pulsacion; // Timestamp when the button was pressed
25     bool larga_enviada;
26 } boton_t;
27
28 static boton_t botones[BUTTONS_NUMBER] = {
29     {e_salida, BUTTON_1},
30     {e_salida, BUTTON_2},
31     {e_salida, BUTTON_3}
32     #if BUTTON_4
33     ,
34     {e_salida, BUTTON_4}
35     #endif
36 };
37
38 // Upper-layer callback.
39 static drv_botones_callback_t drv_botones_callback;
40
41 void drv_botones_iniciar(drv_botones_callback_t botones_callback, EVENTO_T ev1,
42                           EVENTO_T ev2){
43     drv_botones_callback = botones_callback;
44     hal_ext_int_registrar_callback(drv_botones_pulsar_RSI);
45     hal_ext_int_iniciar();
46
47     // Initial state: waiting for presses.
48     for (uint8_t i = 0; i < BUTTONS_NUMBER; ++i) {
49         botones[i].estado = e_esperando;
50     }
51
52     // Enable SENSE (LOW) on all buttons.
53     for (uint8_t i = 1; i <= BUTTONS_NUMBER; ++i) {
54         hal_ext_int_habilitar(i);
55     }
56
57     svc_GE_suscribir(ev1, 1, drv_botones_actualizar);
58     if (ev2 != ev_VOID) {
59         svc_GE_suscribir(ev2, 1, drv_botones_actualizar);
60     }
61 }
```

```

60
61 // Called from the HAL ISR upon press detection.
62 void drv_botones_pulsar_RSI(uint8_t id_boton){
63
64     if (id_boton == 0 || id_boton > BUTTONS_NUMBER) return;
65
66     // Only process if we are waiting (prevents creating multiple alarms).
67     if (botones[id_boton - 1].estado != e_esperando) {
68         return; // This button is already being processed; ignore.
69     }
70
71     // Button pressed: store timestamp.
72     botones[id_boton - 1].inicio_pulsacion = drv_tiempo_actual_ms();
73
74     // Reset long-press flag.
75     botones[id_boton - 1].larga_enviada = false;
76
77
78     // NOTE: For LPC with level-triggered interrupts, the ISR already disables
79     // the interrupt.
80     // For NRF with edge-triggered interrupts, we must disable it here.
81     // Since this function runs in ISR context, LPC already has it disabled.
82     hal_ext_int_deshabilitar(id_boton);
83
84     svc_alarma_activar(svc_alarma_codificar(0, RETARDO_REBOTE_MS, 0),
85                         ev_BOTON_RETARDO, id_boton);
86
87     botones[id_boton - 1].estado = e_rebotes;
88 }
89
90 void drv_botones_actualizar(EVENTO_T evento, uint32_t auxiliar){
91     uint32_t ahora; // used to measure how long the button has been pressed
92     uint8_t id_boton = (uint8_t)auxiliar;
93
94     // Out-of-range button? Return...
95     if (id_boton == 0 || id_boton > BUTTONS_NUMBER) {
96         return;
97     }
98
99     switch (evento) {
100
101         // Internal event: timeout for debounce/release alarms.
102         case ev_BOTON_RETARDO:
103             switch (botones[id_boton - 1].estado){
104
105                 // We just detected the ISR and are in debounce.
106                 case e_rebotes:
107                     if (hal_ext_int_obtener_estado(id_boton)) {
108                         // Still pressed after delay: valid press.
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                         // Schedule alarm for periodic checking.
116                         svc_alarma_activar(svc_alarma_codificar(0,
117                                         RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton);
118                     };
119                 } else {
120                     // Noise/bounce: return to waiting and re-enable.
121                 }
122             }
123         }
124     }
125 }
```

```

119         botones[id_boton - 1].estado = e_esperando;
120         hal_ext_int_habilitar(id_boton);
121     }
122     break;
123
124     // Waiting for the button to be released.
125     case e_muestreo:
126         ahora = drv_tiempo_actual_ms();
127
128         // Check whether it is still physically pressed.
129         if(hal_ext_int_obtener_estado(id_boton)) {
130
131             // Compute press duration.
132             uint32_t duracion = ahora - botones[id_boton - 1].
133                 inicio_pulsacion;
134
135             // If pressed for >= 3 seconds AND not yet notified AND
136             // from button 3 or 4 (id_boton >= 3).
137             if (duracion >= PULSACION_LARGA && !botones[id_boton - 1].larga_enviada && id_boton >= 3) {
138
139                 // Emit the event to the game.
140                 if (drv_botones_callback) {
141                     drv_botones_callback(ev_PULSACION_LARGA,
142                         id_boton);
143                 }
144
145                 // Mark as sent to avoid repeated notifications.
146                 botones[id_boton - 1].larga_enviada = true;
147             }
148
149             // Keep polling periodically while pressed.
150             svc_alarma_activar(
151                 svc_alarma_codificar(0, RETARDO_ENCUESTA_MS, 0),
152                 ev_BOTON_RETARDO,
153                 id_boton
154             );
155
156         } else {
157             // No longer pressed: transition to release state.
158             botones[id_boton - 1].estado = e_salida;
159
160             // Release debounce delay.
161             svc_alarma_activar(svc_alarma_codificar(0,
162                 RETARDO_SOLTAR_MS, 0), ev_BOTON_RETARDO, id_boton);
163         }
164         break;
165
166         // After release, wait for stable idle before re-arming SENSE.
167         case e_salida:
168             if (!hal_ext_int_obtener_estado(id_boton)) {
169                 // Ready for a new press.
170                 botones[id_boton - 1].estado = e_esperando;
171                 hal_ext_int_habilitar(id_boton);
172             } else {
173                 // If still pressed, return to sampling.
174                 botones[id_boton - 1].estado = e_muestreo;
175                 svc_alarma_activar(svc_alarma_codificar(0,
176                     RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton)
177                     ;
178             }
179         break;
180
181     }
182
183     break;
184
185     // If still pressed, return to sampling.
186     case e_muestreo:
187         botones[id_boton - 1].estado = e_muestreo;
188         svc_alarma_activar(svc_alarma_codificar(0,
189             RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton)
190             ;
191
192         break;
193
194     }
195
196     break;
197
198     // If still pressed, return to sampling.
199     case e_esperando:
200         botones[id_boton - 1].estado = e_esperando;
201         svc_alarma_activar(svc_alarma_codificar(0,
202             RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton)
203             ;
204
205         break;
206
207     }
208
209     break;
210
211     // If still pressed, return to sampling.
212     case e_larga_pulsacion:
213         botones[id_boton - 1].estado = e_larga_pulsacion;
214         svc_alarma_activar(svc_alarma_codificar(0,
215             RETARDO_ENCUESTA_MS, 0), ev_BOTON_RETARDO, id_boton)
216             ;
217
218         break;
219
220     }
221
222     break;
223
224     // If still pressed, return to sampling.
225     case e_larga_enviada:
226         botones[id_boton - 1].larga_enviada = false;
227         svc_alarma_desactivar(id_boton);
228
229         break;
230
231     }
232
233     break;
234
235     // If still pressed, return to sampling.
236     case e_no_pulsacion:
237         botones[id_boton - 1].estado = e_no_pulsacion;
238         svc_alarma_desactivar(id_boton);
239
240         break;
241
242     }
243
244     break;
245
246     // If still pressed, return to sampling.
247     case e_no_soltar:
248         botones[id_boton - 1].estado = e_no_soltar;
249         svc_alarma_desactivar(id_boton);
250
251         break;
252
253     }
254
255     break;
256
257     // If still pressed, return to sampling.
258     case e_no_retardo:
259         botones[id_boton - 1].estado = e_no_retardo;
260         svc_alarma_desactivar(id_boton);
261
262         break;
263
264     }
265
266     break;
267
268     // If still pressed, return to sampling.
269     case e_no_encuesta:
270         botones[id_boton - 1].estado = e_no_encuesta;
271         svc_alarma_desactivar(id_boton);
272
273         break;
274
275     }
276
277     break;
278
279     // If still pressed, return to sampling.
280     case e_no_soltar:
281         botones[id_boton - 1].estado = e_no_soltar;
282         svc_alarma_desactivar(id_boton);
283
284         break;
285
286     }
287
288     break;
289
290     // If still pressed, return to sampling.
291     case e_no_retardo:
292         botones[id_boton - 1].estado = e_no_retardo;
293         svc_alarma_desactivar(id_boton);
294
295         break;
296
297     }
298
299     break;
300
301     // If still pressed, return to sampling.
302     case e_no_encuesta:
303         botones[id_boton - 1].estado = e_no_encuesta;
304         svc_alarma_desactivar(id_boton);
305
306         break;
307
308     }
309
310     break;
311
312     // If still pressed, return to sampling.
313     case e_no_soltar:
314         botones[id_boton - 1].estado = e_no_soltar;
315         svc_alarma_desactivar(id_boton);
316
317         break;
318
319     }
320
321     break;
322
323     // If still pressed, return to sampling.
324     case e_no_retardo:
325         botones[id_boton - 1].estado = e_no_retardo;
326         svc_alarma_desactivar(id_boton);
327
328         break;
329
330     }
331
332     break;
333
334     // If still pressed, return to sampling.
335     case e_no_encuesta:
336         botones[id_boton - 1].estado = e_no_encuesta;
337         svc_alarma_desactivar(id_boton);
338
339         break;
340
341     }
342
343     break;
344
345     // If still pressed, return to sampling.
346     case e_no_soltar:
347         botones[id_boton - 1].estado = e_no_soltar;
348         svc_alarma_desactivar(id_boton);
349
350         break;
351
352     }
353
354     break;
355
356     // If still pressed, return to sampling.
357     case e_no_retardo:
358         botones[id_boton - 1].estado = e_no_retardo;
359         svc_alarma_desactivar(id_boton);
360
361         break;
362
363     }
364
365     break;
366
367     // If still pressed, return to sampling.
368     case e_no_encuesta:
369         botones[id_boton - 1].estado = e_no_encuesta;
370         svc_alarma_desactivar(id_boton);
371
372         break;
373
374     }
375
376     break;
377
378     // If still pressed, return to sampling.
379     case e_no_soltar:
380         botones[id_boton - 1].estado = e_no_soltar;
381         svc_alarma_desactivar(id_boton);
382
383         break;
384
385     }
386
387     break;
388
389     // If still pressed, return to sampling.
390     case e_no_retardo:
391         botones[id_boton - 1].estado = e_no_retardo;
392         svc_alarma_desactivar(id_boton);
393
394         break;
395
396     }
397
398     break;
399
400     // If still pressed, return to sampling.
401     case e_no_encuesta:
402         botones[id_boton - 1].estado = e_no_encuesta;
403         svc_alarma_desactivar(id_boton);
404
405         break;
406
407     }
408
409     break;
410
411     // If still pressed, return to sampling.
412     case e_no_soltar:
413         botones[id_boton - 1].estado = e_no_soltar;
414         svc_alarma_desactivar(id_boton);
415
416         break;
417
418     }
419
420     break;
421
422     // If still pressed, return to sampling.
423     case e_no_retardo:
424         botones[id_boton - 1].estado = e_no_retardo;
425         svc_alarma_desactivar(id_boton);
426
427         break;
428
429     }
430
431     break;
432
433     // If still pressed, return to sampling.
434     case e_no_encuesta:
435         botones[id_boton - 1].estado = e_no_encuesta;
436         svc_alarma_desactivar(id_boton);
437
438         break;
439
440     }
441
442     break;
443
444     // If still pressed, return to sampling.
445     case e_no_soltar:
446         botones[id_boton - 1].estado = e_no_soltar;
447         svc_alarma_desactivar(id_boton);
448
449         break;
450
451     }
452
453     break;
454
455     // If still pressed, return to sampling.
456     case e_no_retardo:
457         botones[id_boton - 1].estado = e_no_retardo;
458         svc_alarma_desactivar(id_boton);
459
460         break;
461
462     }
463
464     break;
465
466     // If still pressed, return to sampling.
467     case e_no_encuesta:
468         botones[id_boton - 1].estado = e_no_encuesta;
469         svc_alarma_desactivar(id_boton);
470
471         break;
472
473     }
474
475     break;
476
477     // If still pressed, return to sampling.
478     case e_no_soltar:
479         botones[id_boton - 1].estado = e_no_soltar;
480         svc_alarma_desactivar(id_boton);
481
482         break;
483
484     }
485
486     break;
487
488     // If still pressed, return to sampling.
489     case e_no_retardo:
490         botones[id_boton - 1].estado = e_no_retardo;
491         svc_alarma_desactivar(id_boton);
492
493         break;
494
495     }
496
497     break;
498
499     // If still pressed, return to sampling.
500     case e_no_encuesta:
501         botones[id_boton - 1].estado = e_no_encuesta;
502         svc_alarma_desactivar(id_boton);
503
504         break;
505
506     }
507
508     break;
509
510     // If still pressed, return to sampling.
511     case e_no_soltar:
512         botones[id_boton - 1].estado = e_no_soltar;
513         svc_alarma_desactivar(id_boton);
514
515         break;
516
517     }
518
519     break;
520
521     // If still pressed, return to sampling.
522     case e_no_retardo:
523         botones[id_boton - 1].estado = e_no_retardo;
524         svc_alarma_desactivar(id_boton);
525
526         break;
527
528     }
529
530     break;
531
532     // If still pressed, return to sampling.
533     case e_no_encuesta:
534         botones[id_boton - 1].estado = e_no_encuesta;
535         svc_alarma_desactivar(id_boton);
536
537         break;
538
539     }
540
541     break;
542
543     // If still pressed, return to sampling.
544     case e_no_soltar:
545         botones[id_boton - 1].estado = e_no_soltar;
546         svc_alarma_desactivar(id_boton);
547
548         break;
549
550     }
551
552     break;
553
554     // If still pressed, return to sampling.
555     case e_no_retardo:
556         botones[id_boton - 1].estado = e_no_retardo;
557         svc_alarma_desactivar(id_boton);
558
559         break;
560
561     }
562
563     break;
564
565     // If still pressed, return to sampling.
566     case e_no_encuesta:
567         botones[id_boton - 1].estado = e_no_encuesta;
568         svc_alarma_desactivar(id_boton);
569
570         break;
571
572     }
573
574     break;
575
576     // If still pressed, return to sampling.
577     case e_no_soltar:
578         botones[id_boton - 1].estado = e_no_soltar;
579         svc_alarma_desactivar(id_boton);
580
581         break;
582
583     }
584
585     break;
586
587     // If still pressed, return to sampling.
588     case e_no_retardo:
589         botones[id_boton - 1].estado = e_no_retardo;
590         svc_alarma_desactivar(id_boton);
591
592         break;
593
594     }
595
596     break;
597
598     // If still pressed, return to sampling.
599     case e_no_encuesta:
600         botones[id_boton - 1].estado = e_no_encuesta;
601         svc_alarma_desactivar(id_boton);
602
603         break;
604
605     }
606
607     break;
608
609     // If still pressed, return to sampling.
610     case e_no_soltar:
611         botones[id_boton - 1].estado = e_no_soltar;
612         svc_alarma_desactivar(id_boton);
613
614         break;
615
616     }
617
618     break;
619
620     // If still pressed, return to sampling.
621     case e_no_retardo:
622         botones[id_boton - 1].estado = e_no_retardo;
623         svc_alarma_desactivar(id_boton);
624
625         break;
626
627     }
628
629     break;
630
631     // If still pressed, return to sampling.
632     case e_no_encuesta:
633         botones[id_boton - 1].estado = e_no_encuesta;
634         svc_alarma_desactivar(id_boton);
635
636         break;
637
638     }
639
640     break;
641
642     // If still pressed, return to sampling.
643     case e_no_soltar:
644         botones[id_boton - 1].estado = e_no_soltar;
645         svc_alarma_desactivar(id_boton);
646
647         break;
648
649     }
650
651     break;
652
653     // If still pressed, return to sampling.
654     case e_no_retardo:
655         botones[id_boton - 1].estado = e_no_retardo;
656         svc_alarma_desactivar(id_boton);
657
658         break;
659
660     }
661
662     break;
663
664     // If still pressed, return to sampling.
665     case e_no_encuesta:
666         botones[id_boton - 1].estado = e_no_encuesta;
667         svc_alarma_desactivar(id_boton);
668
669         break;
670
671     }
672
673     break;
674
675     // If still pressed, return to sampling.
676     case e_no_soltar:
677         botones[id_boton - 1].estado = e_no_soltar;
678         svc_alarma_desactivar(id_boton);
679
680         break;
681
682     }
683
684     break;
685
686     // If still pressed, return to sampling.
687     case e_no_retardo:
688         botones[id_boton - 1].estado = e_no_retardo;
689         svc_alarma_desactivar(id_boton);
690
691         break;
692
693     }
694
695     break;
696
697     // If still pressed, return to sampling.
698     case e_no_encuesta:
699         botones[id_boton - 1].estado = e_no_encuesta;
700         svc_alarma_desactivar(id_boton);
701
702         break;
703
704     }
705
706     break;
707
708     // If still pressed, return to sampling.
709     case e_no_soltar:
710         botones[id_boton - 1].estado = e_no_soltar;
711         svc_alarma_desactivar(id_boton);
712
713         break;
714
715     }
716
717     break;
718
719     // If still pressed, return to sampling.
720     case e_no_retardo:
721         botones[id_boton - 1].estado = e_no_retardo;
722         svc_alarma_desactivar(id_boton);
723
724         break;
725
726     }
727
728     break;
729
730     // If still pressed, return to sampling.
731     case e_no_encuesta:
732         botones[id_boton - 1].estado = e_no_encuesta;
733         svc_alarma_desactivar(id_boton);
734
735         break;
736
737     }
738
739     break;
740
741     // If still pressed, return to sampling.
742     case e_no_soltar:
743         botones[id_boton - 1].estado = e_no_soltar;
744         svc_alarma_desactivar(id_boton);
745
746         break;
747
748     }
749
750     break;
751
752     // If still pressed, return to sampling.
753     case e_no_retardo:
754         botones[id_boton - 1].estado = e_no_retardo;
755         svc_alarma_desactivar(id_boton);
756
757         break;
758
759     }
760
761     break;
762
763     // If still pressed, return to sampling.
764     case e_no_encuesta:
765         botones[id_boton - 1].estado = e_no_encuesta;
766         svc_alarma_desactivar(id_boton);
767
768         break;
769
770     }
771
772     break;
773
774     // If still pressed, return to sampling.
775     case e_no_soltar:
776         botones[id_boton - 1].estado = e_no_soltar;
777         svc_alarma_desactivar(id_boton);
778
779         break;
780
781     }
782
783     break;
784
785     // If still pressed, return to sampling.
786     case e_no_retardo:
787         botones[id_boton - 1].estado = e_no_retardo;
788         svc_alarma_desactivar(id_boton);
789
790         break;
791
792     }
793
794     break;
795
796     // If still pressed, return to sampling.
797     case e_no_encuesta:
798         botones[id_boton - 1].estado = e_no_encuesta;
799         svc_alarma_desactivar(id_boton);
800
801         break;
802
803     }
804
805     break;
806
807     // If still pressed, return to sampling.
808     case e_no_soltar:
809         botones[id_boton - 1].estado = e_no_soltar;
810         svc_alarma_desactivar(id_boton);
811
812         break;
813
814     }
815
816     break;
817
818     // If still pressed, return to sampling.
819     case e_no_retardo:
820         botones[id_boton - 1].estado = e_no_retardo;
821         svc_alarma_desactivar(id_boton);
822
823         break;
824
825     }
826
827     break;
828
829     // If still pressed, return to sampling.
830     case e_no_encuesta:
831         botones[id_boton - 1].estado = e_no_encuesta;
832         svc_alarma_desactivar(id_boton);
833
834         break;
835
836     }
837
838     break;
839
840     // If still pressed, return to sampling.
841     case e_no_soltar:
842         botones[id_boton - 1].estado = e_no_soltar;
843         svc_alarma_desactivar(id_boton);
844
845         break;
846
847     }
848
849     break;
850
851     // If still pressed, return to sampling.
852     case e_no_retardo:
853         botones[id_boton - 1].estado = e_no_retardo;
854         svc_alarma_desactivar(id_boton);
855
856         break;
857
858     }
859
860     break;
861
862     // If still pressed, return to sampling.
863     case e_no_encuesta:
864         botones[id_boton - 1].estado = e_no_encuesta;
865         svc_alarma_desactivar(id_boton);
866
867         break;
868
869     }
870
871     break;
872
873     // If still pressed, return to sampling.
874     case e_no_soltar:
875         botones[id_boton - 1].estado = e_no_soltar;
876         svc_alarma_desactivar(id_boton);
877
878         break;
879
880     }
881
882     break;
883
884     // If still pressed, return to sampling.
885     case e_no_retardo:
886         botones[id_boton - 1].estado = e_no_retardo;
887         svc_alarma_desactivar(id_boton);
888
889         break;
890
891     }
892
893     break;
894
895     // If still pressed, return to sampling.
896     case e_no_encuesta:
897         botones[id_boton - 1].estado = e_no_encuesta;
898         svc_alarma_desactivar(id_boton);
899
900         break;
901
902     }
903
904     break;
905
906     // If still pressed, return to sampling.
907     case e_no_soltar:
908         botones[id_boton - 1].estado = e_no_soltar;
909         svc_alarma_desactivar(id_boton);
910
911         break;
912
913     }
914
915     break;
916
917     // If still pressed, return to sampling.
918     case e_no_retardo:
919         botones[id_boton - 1].estado = e_no_retardo;
920         svc_alarma_desactivar(id_boton);
921
922         break;
923
924     }
925
926     break;
927
928     // If still pressed, return to sampling.
929     case e_no_encuesta:
930         botones[id_boton - 1].estado = e_no_encuesta;
931         svc_alarma_desactivar(id_boton);
932
933         break;
934
935     }
936
937     break;
938
939     // If still pressed, return to sampling.
940     case e_no_soltar:
941         botones[id_boton - 1].estado = e_no_soltar;
942         svc_alarma_desactivar(id_boton);
943
944         break;
945
946     }
947
948     break;
949
950     // If still pressed, return to sampling.
951     case e_no_retardo:
952         botones[id_boton - 1].estado = e_no_retardo;
953         svc_alarma_desactivar(id_boton);
954
955         break;
956
957     }
958
959     break;
960
961     // If still pressed, return to sampling.
962     case e_no_encuesta:
963         botones[id_boton - 1].estado = e_no_encuesta;
964         svc_alarma_desactivar(id_boton);
965
966         break;
967
968     }
969
970     break;
971
972     // If still pressed, return to sampling.
973     case e_no_soltar:
974         botones[id_boton - 1].estado = e_no_soltar;
975         svc_alarma_desactivar(id_boton);
976
977         break;
978
979     }
980
981     break;
982
983     // If still pressed, return to sampling.
984     case e_no_retardo:
985         botones[id_boton - 1].estado = e_no_retardo;
986         svc_alarma_desactivar(id_boton);
987
988         break;
989
990     }
991
992     break;
993
994     // If still pressed, return to sampling.
995     case e_no_encuesta:
996         botones[id_boton - 1].estado = e_no_encuesta;
997         svc_alarma_desactivar(id_boton);
998
999         break;
1000
1001     }
1002
1003     break;
1004
1005     // If still pressed, return to sampling.
1006     case e_no_soltar:
1007         botones[id_boton - 1].estado = e_no_soltar;
1008         svc_alarma_desactivar(id_boton);
1009
1010         break;
1011
1012     }
1013
1014     break;
1015
1016     // If still pressed, return to sampling.
1017     case e_no_retardo:
1018         botones[id_boton - 1].estado = e_no_retardo;
1019         svc_alarma_desactivar(id_boton);
1020
1021         break;
1022
1023     }
1024
1025     break;
1026
1027     // If still pressed, return to sampling.
1028     case e_no_encuesta:
1029         botones[id_boton - 1].estado = e_no_encuesta;
1030         svc_alarma_desactivar(id_boton);
1031
1032         break;
1033
1034     }
1035
1036     break;
1037
1038     // If still pressed, return to sampling.
1039     case e_no_soltar:
1040         botones[id_boton - 1].estado = e_no_soltar;
1041         svc_alarma_desactivar(id_boton);
1042
1043         break;
1044
1045     }
1046
1047     break;
1048
1049     // If still pressed, return to sampling.
1050     case e_no_retardo:
1051         botones[id_boton - 1].estado = e_no_retardo;
1052         svc_alarma_desactivar(id_boton);
1053
1054         break;
1055
1056     }
1057
1058     break;
1059
1060     // If still pressed, return to sampling.
1061     case e_no_encuesta:
1062         botones[id_boton - 1].estado = e_no_encuesta;
1063         svc_alarma_desactivar(id_boton);
1064
1065         break;
1066
1067     }
1068
1069     break;
1070
1071     // If still pressed, return to sampling.
1072     case e_no_soltar:
1073         botones[id_boton - 1].estado = e_no_soltar;
1074         svc_alarma_desactivar(id_boton);
1075
1076         break;
1077
1078     }
1079
1080     break;
1081
1082     // If still pressed, return to sampling.
1083     case e_no_retardo:
1084         botones[id_boton - 1].estado = e_no_retardo;
1085         svc_alarma_desactivar(id_boton);
1086
1087         break;
1088
1089     }
1090
1091     break;
1092
1093     // If still pressed, return to sampling.
1094     case e_no_encuesta:
1095         botones[id_boton - 1].estado = e_no_encuesta;
1096         svc_alarma_desactivar(id_boton);
1097
1098         break;
1099
1100     }
1101
1102     break;
1103
1104     // If still pressed, return to sampling.
1105     case e_no_soltar:
1106         botones[id_boton - 1].estado = e_no_soltar;
1107         svc_alarma_desactivar(id_boton);
1108
1109         break;
1110
1111     }
1112
1113     break;
1114
1115     // If still pressed, return to sampling.
1116     case e_no_retardo:
1117         botones[id_boton - 1].estado = e_no_retardo;
1118         svc_alarma_desactivar(id_boton);
1119
1120         break;
1121
1122     }
1123
1124     break;
1125
1126     // If still pressed, return to sampling.
1127     case e_no_encuesta:
1128         botones[id_boton - 1].estado = e_no_encuesta;
1129         svc_alarma_desactivar(id_boton);
1130
1131         break;
1132
1133     }
1134
1135     break;
1136
1137     // If still pressed, return to sampling.
1138     case e_no_soltar:
1139         botones[id_boton - 1].estado = e_no_soltar;
1140         svc_alarma_desactivar(id_boton);
1141
1142         break;
1143
1144     }
1145
1146     break;
1147
1148     // If still pressed, return to sampling.
1149     case e_no_retardo:
1150         botones[id_boton - 1].estado = e_no_retardo;
1151         svc_alarma_desactivar(id_boton);
1152
1153         break;
1154
1155     }
1156
1157     break;
1158
1159     // If still pressed, return to sampling.
1160     case e_no_encuesta:
1161         botones[id_boton - 1].estado = e_no_encuesta;
1162         svc_alarma_desactivar(id_boton);
1163
1164         break;
1165
1166     }
1167
1168     break;
1169
1170     // If still pressed, return to sampling.
1171     case e_no_soltar:
1172         botones[id_boton - 1].estado = e_no_soltar;
1173         svc_alarma_desactivar(id_boton);
1174
1175         break;
1176
1177     }
1178
1179     break;
1180
1181     // If still pressed, return to sampling.
1182     case e_no_retardo:
1183         botones[id_boton - 1].estado = e_no_retardo;
1184         svc_alarma_desactivar(id_boton);
1185
1186         break;
1187
1188     }
1189
1190     break;
1191
1192     // If still pressed, return to sampling.
1193     case e_no_encuesta:
1194         botones[id_boton - 1].estado = e_no_encuesta;
1195         svc_alarma_desactivar(id_boton);
1196
1197         break;
1198
1199     }
1200
1201     break;
1202
1203     // If still pressed, return to sampling.
1204     case e_no_soltar:
1205         botones[id_boton - 1].estado = e_no_soltar;
1206         svc_alarma_desactivar(id_boton);
1207
1208         break;
1209
1210     }
1211
1212     break;
1213
1214     // If still pressed, return to sampling.
1215     case e_no_retardo:
1216         botones[id_boton - 1].estado = e_no_retardo;
1217         svc_alarma_desactivar(id_boton);
1218
1219         break;
1220
1221     }
1222
1223     break;
1224
1225     // If still pressed, return to sampling.
1226     case e_no_encuesta:
1227         botones[id_boton - 1].estado = e_no_encuesta;
1228         svc_alarma_desactivar(id_boton);
1229
1230         break;
1231
1232     }
1233
1234     break;
1235
1236     // If still pressed, return to sampling.
1237     case e_no_soltar:
1238         botones[id_boton - 1].estado = e_no_soltar;
1239         svc_alarma_desactivar(id_boton);
1240
1241         break;
1242
1243     }
1244
1245     break;
1246
1247     // If still pressed, return to sampling.
1248     case e_no_retardo:
1249         botones[id_boton - 1].estado = e_no_retardo;
1250         svc_alarma_desactivar(id_boton);
1251
1252         break;
1253
1254     }
1255
1256     break;
1257
1258     // If still pressed, return to sampling.
1259     case e_no_encuesta:
1260         botones[id_boton - 1].estado = e_no_encuesta;
1261         svc_alarma_desactivar(id_boton);
1262
1263         break;
1264
1265     }
1266
1267     break;
1268
1269     // If still pressed, return to sampling.
1270     case e_no_soltar:
1271         botones[id_boton - 1].estado = e_no_soltar;
1272         svc_alarma_desactivar(id_boton);
1273
1274         break;
1275
1276     }
1277
1278     break;
1279
1280     // If still pressed, return to sampling.
1281     case e_no_retardo:
1282         botones[id_boton - 1].estado = e_no_retardo;
1283         svc_alarma_desactivar(id_boton);
1284
1285         break;
1286
1287     }
1288
1289     break;
1290
1291     // If still pressed, return to sampling.
1292     case e_no_encuesta:
1293         botones[id_boton - 1].estado = e_no_encuesta;
1294         svc_alarma_desactivar(id_boton);
1295
1296         break;
1297
1298     }
1299
1300     break;
1301
1302     // If still pressed, return to sampling.
1303     case e_no_soltar:
1304         botones[id_boton - 1].estado = e_no_soltar;
1305         svc_alarma_desactivar(id_boton);
1306
1307         break;
1308
1309     }
1310
1311     break;
1312
1313     // If still pressed, return to sampling.
1314     case e_no_retardo:
1315         botones[id_boton - 1].estado = e_no_retardo;
1316         svc_alarma_desactivar(id_boton);
1317
1318         break;
1319
1320     }
1321
1322     break;
1323
1324     // If still pressed, return to sampling.
1325     case e_no_encuesta:
1326         botones[id_boton - 1].estado = e_no_encuesta;
1327         svc_alarma_desactivar(id_boton);
1328
1329         break;
1330
1331     }
1332
1333     break;
1334
1335     // If still pressed, return to sampling.
1336     case e_no_soltar:
1337         botones[id_boton - 1].estado = e_no_soltar;
1338         svc_alarma_desactivar(id_boton);
1339
1340         break;
1341
1342     }
1343
1344     break;
1345
1346     // If still pressed, return to sampling.
1347     case e_no_retardo:
1348         botones[id_boton - 1].estado = e_no_retardo;
1349         svc_alarma_desactivar(id_boton);
1350
1351         break;
1352
1353     }
1354
1355     break;
1356
1357     // If still pressed, return to sampling.
1358     case e_no_encuesta:
1359         botones[id_boton - 1].estado = e_no_encuesta;
1360         svc_alarma_desactivar(id_boton);
1361
1362         break;
1363
1364     }
1365
1366     break;
1367
1368     // If still pressed, return to sampling.
1369     case
```

```
175         case e Esperando:
176             default:
177                 break;
178         }
179         break;
180
181     case ev_BOTON_PULSAR:
182         break;
183
184     default:
185         break;
186     }
187 }
```

Appendix J: Alarm Service (svc_alarmas.c)

```

1 #include "svc_alarmas.h"
2 #include "drv_monitor.h"
3 #include "rt_sc.h"
4
5 // Internal structure
6 typedef struct{
7     bool activa;
8     svc_alarma_flags_t flags;           // Activation flags encoding.
9     uint32_t restante_ms;              // Remaining countdown in ms.
10    EVENTO_T id_evento;               // Event to enqueue upon expiration.
11    uint32_t aux;                     // auxData carried with the event.
12} alarma_t;
13
14 typedef struct{
15     alarma_t alarmas[svc_ALARMAS_MAX];          // All alarms in the
16                                         system.
16     MONITOR_id_t monitor_overflow;             // Monitor asserted
17                                         when no alarm slots remain.
17     void (*cb_encolar)(uint32_t id_ev, uint32_t aux); // Enqueue function
18     EVENTO_T ev_tick;                         // Tick event that
19                                         triggers updates.
20 } sistema;
21
22 static sistema s;
23
24 // Auxiliary (internal) functions.
25 static inline bool es_periodica(svc_alarma_flags_t f) {
26     return (f & SVC_ALARMA_FLAG_PERIODICA) != 0;
27 }
28
29 static inline uint32_t periodo_ms(svc_alarma_flags_t f) {
30     return (f & SVC_ALARMA_RETARDO_MASK_MS);
31 }
32
33 static int buscar_por_evento(EVENTO_T ev, uint32_t aux)
34 {
35     for (int i = 0; i < (int)svc_ALARMAS_MAX; ++i) {
36         if (!s.alarmas[i].activa) continue;
37         if (s.alarmas[i].id_evento != ev) continue;
38
39         // For button debounce, also distinguish by aux (button id).
40         if (ev == ev_BOTON_RETARDO) {
41             if (s.alarmas[i].aux == aux) return i;
42         } else {
43             // For other events, keep the original semantics.
44             return i;
45         }
46     }
47     return -1;
48 }
49
50 static int buscar_libre(void) {
51     for (int i = 0; i < (int)svc_ALARMAS_MAX; ++i) {
52         if (!s.alarmas[i].activa) return i;
53     }
54     return -1;
55 }
56
57 static void disparar(const alarma_t* a) {
58     if (s.cb_encolar) {

```

```

59         s.cb_encolar((uint32_t)a->id_evento, a->aux);
60     }
61 }
62
63 // Public operations
64 void svc_alarma_iniciar(uint32_t monitor_overflow, void (*cb_encolar)(uint32_t
65 id_ev, uint32_t aux), EVENTO_T ev_a_notificar){
66
67     for (int i = 0; i < (int)svc_ALARMAS_MAX; ++i) {
68         s.alarmas[i].activa = false;
69         s.alarmas[i].flags = 0;
70         s.alarmas[i].restante_ms = 0;
71         s.alarmas[i].id_evento = ev_VOID;
72         s.alarmas[i].aux = 0;
73     }
74
75     s.monitor_overflow = monitor_overflow;
76     s.cb_encolar = cb_encolar;
77     s.ev_tick = ev_a_notificar;
78 }
79
80 void svc_alarma_activar(svc_alarma_flags_t flags, EVENTO_T id_evento, uint32_t
81 auxData){
82     rt_sc_estado_t sc = rt_sc_entrar();
83     // Check whether the alarm for the event already exists and is active.
84     int id_alarma = buscar_por_evento(id_evento, auxData);
85
86     // If retardo_ms == 0 => CANCEL the alarm (if it exists).
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     // If we are not re-arming, find the next free alarm slot.
96     if (id_alarma < 0) id_alarma = buscar_libre();
97
98     // If none are free, assert the monitor.
99     if (id_alarma < 0) {
100        if (s.monitor_overflow) drv_monitor_marcar(s.monitor_overflow);
101        while(true){ /* No alarm slots available */ }
102    }
103
104    // If re-arming or creating, fill in the new data.
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     // Treat auxData as ms elapsed since the last tick (if 0, use at least 1).
118     uint32_t transcurrido = (auxData == 0) ? 1 : auxData;
119

```

```
120     for(int i = 0; i < (int)svc_ALARMAS_MAX; ++i){  
121         if(!s.alarmas[i].activa) continue;  
122  
123         if(transcurrido >= s.alarmas[i].restante_ms){  
124             // EXPIRES  
125             disparar(&s.alarmas[i]);  
126  
127             if (es_periodica(s.alarmas[i].flags)) {  
128                 // Periodic re-arming.  
129                 uint32_t periodo = periodo_ms(s.alarmas[i].flags);  
130                 uint32_t overshoot = transcurrido - s.alarmas[i].restante_ms;  
131  
132                 if (periodo == 0) periodo = 1;  
133                 // If overshoot is a multiple of the period, fire once and keep  
134                 // a full period remaining.  
135                 s.alarmas[i].restante_ms = periodo - (overshoot % periodo);  
136                 if (s.alarmas[i].restante_ms == 0) s.alarmas[i].restante_ms =  
137                     periodo;  
138             } else {  
139                 // One-shot: deactivate.  
140                 s.alarmas[i].activa = false;  
141             }  
142         } else{  
143             // Not yet expired: decrement.  
144             s.alarmas[i].restante_ms -= transcurrido;  
145         }  
146     }  
147     rt_sc_salir(sc);  
148 }
```