PROYECTO HARDWARE

----------MEMORIA DEL PROYECTO----------

CURSO 2024 - 2025

Grado de Ingeniería Informática

Universidad de Zaragoza

Ariana Porroche Llorén - 874055

Irene Pascual Albericio - 871627

1. ÍNDICE

2. INTRODUCCIÓN

En nuestro curso de ingeniería informática, nos hemos enfrentado en la asignatura de Proyecto Hardware al reto de aprender a interactuar con hardware complejo y desarrollar software, que no solo sea funcional, sino también eficiente y adaptativo a diferentes plataformas.

Nuestro proyecto incluye tres componentes principales de programación:

1) **Blinks**: son una serie de ejercicios que incrementan progresivamente en complejidad y que tienen como objetivo manejar el parpadeo de LEDs.

2) **Bit Counter Strike**: un juego interactivo que utiliza LEDs y botones, donde el jugador debe responder rápidamente a secuencias aleatorias para incrementar la dificultad del juego. Este componente del proyecto permite integrar gestión de eventos y temporizadores, enfocándose en la interacción del usuario-máquina.

3) **El juego del Simón**: es una versión avanzada que integra todos los conocimientos previos en un juego de memoria y habilidad. Los jugadores deben observar y replicar secuencias de luces, incrementando en longitud y complejidad con cada nivel superado. Los errores en la secuencia o el exceder el tiempo de respuesta, llevan a una secuencia de error y al fin del juego.

Este documento tiene como objetivo mostrar el proceso y los resultados de nuestro proyecto, donde nos centramos en manipular entornos de hardware específicos como el LPC y la placa nRF, y en desarrollar un juego que funcionase efectivamente en ambos.

La organización de este documento sigue una estructura clara para facilitar la comprensión de nuestro trabajo. Comenzamos estableciendo los objetivos que guían el proyecto, seguido de una metodología detallada que incluye las herramientas y técnicas utilizadas. Posteriormente, observamos y analizamos los resultados obtenidos y se concluye con las lecciones aprendidas y las referencias que nos han ayudado en nuestro desarrollo.

En cada sección presentaremos no solo lo que hicimos, sino también por qué lo hicimos. En esta memoria reflejamos tanto nuestro trabajo como nuestro aprendizaje.

3. OBJETIVOS

A continuación vamos a explicar los diferentes objetivos que hemos planteado a lo largo de este curso, enfocados principalmente en maximizar la eficacia y usabilidad del proyecto final, el juego del Simón:

1. **Interacción con plataformas hardware:** aprender a manejar y configurar el entorno LPC y la placa nRF teniendo en cuenta sus capacidades y limitaciones técnicas. Esto incluye la adaptación del código para que sea compatible en ambas plataformas, haciendo que únicamente dependa de aspectos hardware de cada uno.
2. **Desarrollo de programas eficientes energéticamente:** analizar y optimizar el consumo energético, identificando las zonas de código que requieren más energía y tratando de mejorar en cada versión.
3. **Optimización del tiempo de uso de la placa:** mejorar la organización y planificación del tiempo, para maximizar la eficiencia durante las horas de acceso a la placa en el laboratorio.
4. **Dominio de timers en programación:** desarrollar la capacidad para programar y administrar timers de forma efectiva para controlar el flujo de eventos dentro del código.
5. **Manejo de múltiples mains en un solo programa:** aprender a configurar y utilizar diferentes modos de compilación para manejar varios archivos main en un único proyecto, permitiendo pruebas más flexibles y una organización del código más clara.
6. **Implementación de pruebas de código:** desarrollar un conjunto de pruebas que aseguren el correcto funcionamiento de algunos componentes del código de forma independiente.
7. **Identificación y resolución de problemas de condiciones de carrera:** saber detectar y solucionar problemas de condiciones de carrera y otros errores de sincronización que puedan surgir durante el desarrollo del juego.
8. **Sincronización de entrada y salida:** asegurar que el pulsado de botones esté perfectamente sincronizado con la salida de los LEDs para proporcionar una respuesta inmediata durante el programa.
9. **Implementación de un sistema de Watchdog:** programar un sistema de watchdog capaz de reiniciar el juego automáticamente después de un tiempo determinado en caso de fallos o posibles bloqueos, asegurando la continuidad del juego sin que el usuario jugador tenga que modificar nada.
10. **Diseño centrado en el usuario:** comprender detalladamente lo que los usuarios jugadores buscan y prefieren, con el objetivo de diseñar características de juego que sean atractivas y mejoren su experiencia, haciéndola satisfactoria y amena.

Estos objetivos nos van a servir para guiarnos durante el desarrollo del código, para que finalmente, el juego del Simón, no solo sea más eficiente técnica como energéticamente, sino que también sea entretenido y atractivo para los usuarios.

4. METODOLOGÍA

4.1. HERRAMIENTAS

Para desarrollar el código en C y poder ejecutar, compilar y depurar, hemos utilizado en entorno de desarrollo **Keil uVision5**, en la versión 5.26. Esta herramienta tiene soporte para múltiples microcontroladores, de los que hemos usado el LPC2105 desarrollado por NXP y en nRF52840 DK, con arquitectura Arm Cortex-M4F. Además, hemos trabajado con distintos proyectos a la vez, y dentro de estos, con distintos modos de compilación para poder probar el juego o el test que nos interesara.

La placa que hemos utilizado en los laboratorios para probar nuestros juegos se trata de la **nRF52840 DK** desarrollada por Nordic, la cual cuenta con 4 LEDs y 4 botones que hemos utilizado para desarrollar los programas de cada práctica.

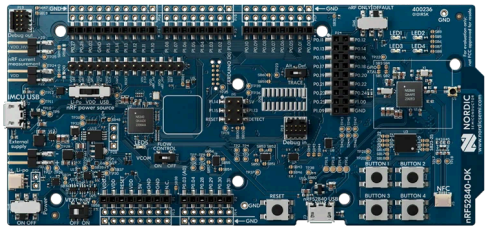

Figura 1: Placa nRF52840 DK de Nordic

Para medir el consumo de los distintos programas, utilizamos la aplicación de escritorio **nRF Connect**, dentro de la cual utilizamos la herramienta **Power Profiler**. Con esta herramienta, analizamos las trazas que habíamos obtenido previamente en los laboratorios utilizando el dispositivo hardware **Power Profiler Kit II (PPK2)**.

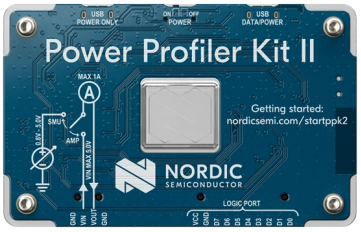

Figura 2: Dispositivo Power Profiler Kit II (PPK2) de Nordic

4.2. DISEÑO DE LOS JUEGOS / PROGRAMAS

Durante el curso hemos llevado a cabo diversas tareas, las cuales iban incrementando de nivel poco a poco, para que mediante el paso del tiempo, aprendiésemos a manejar con soltura tanto los conceptos básicos como las herramientas más complejas.

Hemos empezado con ejercicios simples, como programar un LED para que parpadee, y más adelante, hemos realizado tareas que implican un uso más avanzado del hardware y software.

En las primeras prácticas, por ejemplo con el ***blink_v1***, nos centramos en entender cómo funcionan los pines de entrada y salida controlando el parpadeo de un LED periódicamente

gracias a un bucle con espera activa. Aunque este método no es el más eficiente, nos ayudó a familiarizarnos con la programación a nivel de hardware de forma directa y sencilla.

Aspectos clave:

- Control directo de GPIO
- Espera activa: usamos bucles de espera activa que consumen ciclos de CPU para mantener el tiempo entre los estados de encendido y apagado del LED.

```
void blink_v1(uint32_t id){
	while (1) {
		uint32_t volatile tmo;
		tmo = 10000000;
		while (tmo--);
		drv_led_conmutar(id);
	}
}
```

Código 1: Main de blink_v1

Luego avanzamos a ***blink_v2***, donde introdujimos temporizadores hardware para controlar el parpadeo del LED. Esto no solo ayudó a que el LED parpadeara con mayor precisión, sino que también nos ayudó a entender cómo funcionaban los timers en ambos entornos. En este código inicializamos el controlador de consumo energético, y nos permite empezar a medir y controlar el consumo de este código.

Aspectos clave:

- Uso de temporizadores: implementamos temporizadores para evitar la espera activa que realizamos en el anterior código, y con esto logramos ser más eficientes energéticamente.
- Control de consumo: introducimos este módulo para medir y optimizar el uso de energía.

```
void blink_v2(uint32_t id){
	drv_tiempo_iniciar();
	drv_consumo_iniciar(3);
	Tiempo_ms_t siguiente_activacion;
	drv_led_encender(id);
	siguiente_activacion = drv_tiempo_actual_ms();
	while (true) {
		siguiente_activacion += RETARDO_MS;
		drv_tiempo_esperar_hasta_ms(siguiente_activacion);
		drv_led_conmutar(id);
	}
}
```

Código 2: Main de blink_v2

Continuando con la evolución de las prácticas de parpadeo de LEDs, llegamos a ***blink_v3***, que representa un avance muy significativo ya que empezamos a gestionar las tareas mediante interrupciones.

Aspectos clave:

- Gestión de tiempo e interrupciones: configuramos interrupciones periódicas que permiten evitar que el procesador esté constantemente activo, despertándose sólo para conmutar el estado del LED.

- Modo de bajo consumo: vamos a mandar al procesador al modo espera, del cual despertará una vez llegue una interrupción.

```
void leds_c(uint32_t id, Tiempo_ms_t ms) {
	drv_led_conmutar(id);
}

void blink_v3(uint32_t id){
	drv_tiempo_iniciar();
	drv_monitor_iniciar_unico(3);
	drv_led_encender(id);
	drv_tiempo_periodico_ms(RETARDO_MS, leds_c, id);
	while (1) {
		drv_consumo_esperar();
	}
}
```

Código 3: Main y función auxiliar de blink_v3

Respecto al siguiente código, ***blink_v3_bis***, este va a provocar el parpadeo del LED 20 veces (es decir, 10 encendidos y 10 apagados), manteniendo el modo de bajo consumo entre cada ciclo. Una vez finalice esta secuencia, procederá a entrar en un nuevo modo incorporado, llamado modo sueño profundo, que sólo despertará al sistema hasta que, tanto en el nRF como el LPC, pulsemos cualquier botón (evento externo).

Aspectos clave:

- Modo sueño profundo: reducimos el consumo de energía al mínimo y solo puede ser interrumpido por eventos externos.
- Comportamiento al despertar: realizaremos un reinicio completo, por lo que volverá a repetir la secuencia que se realizó anteriormente.

```
void leds_c(uint32_t id, Tiempo_ms_t ms) {
	drv_led_conmutar(id);
}

void blink_v3_bis(uint32_t id){
	drv_tiempo_iniciar();
	drv_monitor_iniciar_unico(3);
	drv_botones_iniciar(rt_FIFO_encolar);
	drv_led_encender(id);
	drv_tiempo_periodico_ms(RETARDO_MS, leds_c, id);
	uint32_t contador = 20;
	while (contador-- > 0) {
		drv_consumo_esperar();
	}
	drv_led_apagar(id);
	drv_consumo_dormir();
}
```

Código 4: Main y función auxiliar de blink_v3_bis

El siguiente código que realizamos fue el ***blink_v4***, que activa un temporizador, que cuando salté, realizará la función de callback de *rt_FIFO_encolar* del evento en cuestión. Mientras tanto, si el procesador ha intentado extraer, sin éxito, un evento de la cola, irá a modo de bajo consumo. El procesador se reanudará cuando se haya encolado un evento en la fifo, y si al extraerlo es un evento periódico, conmutará la LED.

Aspectos clave:

- Temporizadores programados: establecemos un temporizador en intervalos regulares, que ejecuta una función de callback cuando este salta. Esto permite desacoplar la gestión del tiempo del resto de la lógica del programa.
- Cola de eventos FIFO: implementamos una cola FIFO para manejar los eventos y asegurar que todos los eventos se procesan en el orden en que se encolan.
- Procesamiento basado en eventos: el sistema verifica continuamente la cola para asegurar que todas las tareas necesarias se realizan a tiempo.

```
void leds_c(uint32_t id, Tiempo_ms_t ms) {
        drv_led_conmutar(id);
}

void blink_v4(uint32_t id) {
        drv_tiempo_iniciar();
        rt_FIFO_inicializar(MONITOR4);
        drv_led_encender(id);
        drv_tiempo_periodico_ms(RETARDO_MS, rt_FIFO_encolar, ev_T_PERIODICO);
        while (1) {
                EVENTO_T EV_ID_evento;
                uint32_t EV_auxData;
                Tiempo_us_t EV_TS;
                if (rt_FIFO_extraer(&EV_ID_evento, & EV_auxData, & EV_TS)){
                        if (EV_ID_evento == ev_T_PERIODICO){
                                drv_led_conmutar(id);
                        }
                }
                else {
                        drv_consumo_esperar();
                }
        }
}
```

Código 5: Main y función auxiliar de blink_v4

En cuanto al código del ***Bit Counter Strike***, el objetivo va a ser crear un juego interactivo que utiliza LEDs y botones en donde el jugador debe responder correctamente si quiere aumentar la velocidad (y por tanto la dificultad). Para ello se muestra una LED aleatoria y el jugador debe pulsar el botón relacionado con esa misma LED. Al responder correctamente, se disminuye el tiempo que tarda en mostrarse la LED (osea, se aumenta la velocidad de pulsado). Si falla, la velocidad permanece igual.

Hacemos usos de diversos tipos de eventos para controlar la lógica e interactividad:

- *ev_VOID*: lo utilizamos para contabilizar el número de eventos totales.
- *ev_T_PERIODICO*: lo utilizamos para activar acciones repetitivas dentro del juego. En vez de usar una programación estrictamente periódica, este evento se reprograma cada vez que ocurre, permitiendo ajustar dinámicamente el intervalo basado en cómo está jugando el jugador.
  La razón detrás de esta metodología se debe a las limitaciones en el número de timers disponibles, ya que solo contabamos con dos en el LPC. Mientras uno de ellos se reserva permanentemente para el contador de máxima precisión y detectar inactividad, el otro se libera para ser reutilizado para otros tipos de eventos. Esta estrategia nos optimiza nuestros recursos de hardware y permite más flexibilidad en la gestión del juego.

- *ev_PULSAR_BOTON*: lo utilizamos en el hal_ext_int para poder indicar que la pulsación del botón ha llegado y ha sido manejado por el IRQ_handler.
- *ev_INACTIVIDAD*: lo utilizamos para detectar la ausencia de interacción del usuario durante un periodo que hemos determinado previamente.
- *ev_BOTON_RETARDO*: lo utilizamos para gestionar los rebotes de los botones al ser presionados, evitando múltiples registros de una sola pulsación.
- *ev_SOLTAR_BOTON*: lo utilizamos para detectar cuando un botón es liberado.
- *ev_T_ESPORADICO*: lo utilizamos para activar acciones dentro del juego que no siguen una secuencia temporal fija.

Hemos incorporado un servicio de alarmas que está configurado para activar temporizadores para eventos con el tiempo que se le indica.

Todos estos eventos los tratamos en el archivo *rt_GE*, pero estos solo pueden ser extraídos si han sido previamente colocados en la cola (lo cual se hace con una función de callback que llama a *rt_FIFO_encolar* cuando salta el temporizador programado), sino, mientras tanto el programa irá a modo de bajo consumo. Si no hay eventos para extraer, el programa entra en modo de bajo consumo hasta que se extrae un evento de la cola, momento en el que se ejecutan las funciones de callback asociadas a cada evento.

```
void bit_counter_strike(){
	drv_tiempo_iniciar();
	hal_WDG_iniciar(7);
	rt_FIFO_inicializar(MONITOR4);
	rt_GE_iniciar(MONITOR1);
	drv_led_encender(1);
	svc_alarma_iniciar(MONITOR2, rt_FIFO_encolar, ev_T_PERIODICO, 5000);
	uint8_t num_botones=drv_botones_iniciar(rt_FIFO_encolar);
	iniciar_secuencia_aleatoria(num_botones, 1);
	rt_GE_lanzador();
}

int main(void){
	uint32_t Num_Leds;
	hal_gpio_iniciar();
	Num_Leds = drv_leds_iniciar();
	if (Num_Leds > 0) {
		bit_counter_strike();
	}
}
```

Código 6: Main y función del Bit Counter Strike

Por último, **el juego del Simón** inicia con una secuencia de luces que indica que se puede elegir un modo de juego (apartado opcional que hemos pensado que es buena idea para dar más jugabilidad, el cual comentamos más adelante). Una vez ya se ha elegido el modo, se muestra la secuencia de luces que indica el comienzo de la partida. Luego, una de las luces se ilumina aleatoriamente y el jugador debe observar y recordar la secuencia de luces, ya que debe reproducir esa misma secuencia presionando los botones en el mismo orden que han sido mostrados. Si el jugador reproduce correctamente la secuencia, el juego avanza al siguiente nivel donde se añade otro elemento a la secuencia y se aumenta la velocidad de mostrado de LEDs. Si no, se termina el juego y se muestra una secuencia diferente de terminar.

Para este nuevo juego, hemos realizado varios cambios y añadidos significativos comparados con el proyecto comentado anteriormente. A continuación vamos a explicarlos:

1) Modos de compilación: hemos configurado y establecido distintos modos de compilación para evitar tener diferentes mains en diferentes archivos o tener que comentar código para poder ejecutar. Estos modos son, el juego en sí, tests de algunos módulos y también un modelo de depuración (el juego junto con sus estadísticas).

```
int main(void) {
    #if defined(modo_simon_lpc)||defined(modo_simon_nrf)||defined(modo_depuracion_lpc)||defined(modo_depuracion_nrf)
        hal_gpio_iniciar();
        uint32_t num_Leds = drv_leds_iniciar();
        if (num_Leds > 0){
            drv_tiempo_iniciar();
            hal_WDG_iniciar(6);
            rt_FIFO_inicializar(MONITOR4);
            rt_GE_iniciar(MONITOR1, maq_estados_simon);
            svc_alarma_iniciar(MONITOR2,rt_FIFO_encolar,ev_T_PERIODICO,svc_alarma_codificar(1,VELOCIDAD_SECUENCIA_INICIO));
            #if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
                actualizar_estadisticas_empezando_a_ejecutar();
            #endif
            iniciar_simon();
        }

    #elif defined(modo_test_fifo)
        run_tests_fifo(MONITOR4);

    #elif defined(modo_test_WDG)
        hal_WDG_iniciar(10);
        hal_WDG_feed();
        drv_consumo_esperar();

    #elif defined(modo_test_inactividad_lpc) || defined(modo_test_inactividad_nrf)
        uint8_t num_botones = drv_botones_iniciar(rt_FIFO_encolar);
        drv_consumo_dormir();
    #endif
}
```

Código 7: Main con diferentes modos de compilación del juego del Simon

2) Archivo dedicado a la lógica del juego: hemos trasladado la lógica del juego desde el archivo *rt_GE* a un módulo independiente: *simon*. De esta forma, concentramos toda la gestión del juego en un solo lugar específico para ello.

3) Manejo de temporizadores y alarmas: ahora cuando salta una alarma que hemos programado previamente, se dirige a su función de callback que es *svc_alarma_tratar*, donde añadirá el evento del cual ha saltado su alarma a la cola fifo mediante la función de callback que tiene almacenada que es *rt_FIFO_encolar*. Con esto, evitamos que el módulo de temporizadores tengan que entender de tipos de eventos, y sí que se encargue un módulo que esté destinado a ello.

4) Creación de archivo para la sección crítica: hemos incluido un archivo específico para realizar la sección crítica, ya que anteriormente la realizamos en el *hal_ext_int*.

5) Creación de archivo para obtener estadísticas: hemos incluido un archivo para medir las estadísticas y mostrarlas cuando se termina el juego, solo si estamos ejecutando en modo depuración.

Con estos cambios, además de los propios en sí para que funcione de forma diferente el juego, logramos centralizar y organizar la lógica del juego, ya que en el Bit Counter Strike la gestión estaba más dispersa entre varios módulos.

Cabe destacar que hemos incluido de forma opcional al inicio del juego, una selección de dificultad:

1) Nivel normal: el juego muestra cada vez una LED extra y funciona a una velocidad moderada.
2) Nivel rápido: el juego difiere en que aumenta la velocidad.
3) Nivel doble: el juego difiere en que muestra cada vez dos LEDs seguidas extras.

El diseño del juego del simón que hemos realizado consta de 5 estados: *s_elegir_modo_juego*, *s_secuencia_inicio*, *s_secuencia*, *s_esperar_a_pulsar* y *s_secuencia_terminar*. El primer estado, *s_elegir_modo_juego*, muestra la secuencia de elección del modo de juego y enciende las 3 LEDs para que el usuario elija modo de juego. Una vez el usuario ha seleccionado el modo de juego presionando el botón correspondiente, nos iríamos al estado *s_secuencia_inicio*. El segundo estado, *s_secuencia_inicio*, muestra la secuencia de inicio de la partida y va a *s_secuencia*. El estado *s_secuencia*, es el que muestra la secuencia de LEDs que el usuario debe pulsar. Una vez mostrada la secuencia a pulsar, va a *s_esperar_a_pulsar*. Este estado comprueba que el usuario haya pulsado los botones correctos en el orden correcto. Si alguno de ellos es incorrecto, va a *s_secuencia_terminar*, que mostraría la secuencia de fin de la partida. En caso contrario, volvemos a *s_secuencia* para mostrar la siguiente secuencia de LEDs a pulsar, incrementándose en número y decrementando su velocidad de aparición.

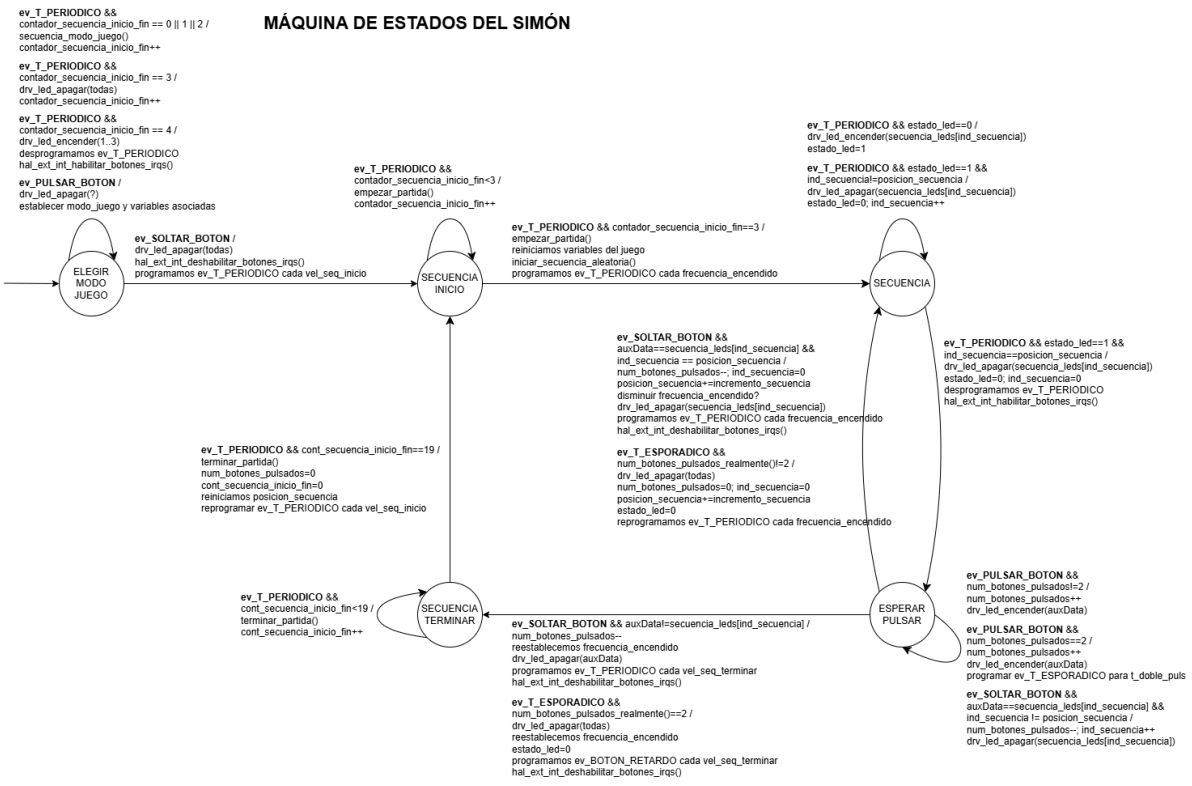

Figura 3: Diagrama de estados del juego del simón

4.3. DISEÑO DE LOS MÓDULOS

TIMERS Y ALARMAS

En cuanto al diseño del módulo del tiempo y las alarmas (*svc_alarmas*, *drv_tiempo* y *hal_tiempo*) hemos decidido incorporar 2 timers: ***Timer0*** y ***Timer1***. El *Timer0* es el que programará la alarma del evento *ev_INACTIVIDAD*, por lo que siempre estará funcionando y es el que utilizaremos para el contador de máxima precisión. El *Timer1* es el que se encarga de los demás eventos y el que vamos programando y desprogramando a lo largo del juego según los tiempos que nos interesan en cada momento. Como los módulos *drv_tiempo* y *hal_tiempo* no entienden de eventos, es la función *svc_alarma_activar* [1] la que se encarga de indicarles cuál timer de los dos deben programar.

Para ver el tiempo actual, podemos utilizar las funciones *drv_tiempo_actual_us* y *drv_tiempo_actual_ms*, las cuales obtienen el número de ticks del *Timer0* y hacen la conversión a us o ms respectivamente.

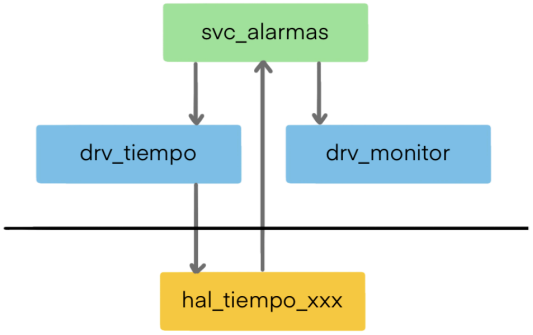

Figura 4: Diagrama de interacción entre módulos para las alarmas y el tiempo

RT_FIFO

El módulo **rt_FIFO** implementa una cola FIFO circular, de tal manera que los elementos encolados se extraen en el mismo orden en el que se encolaron. La cola es un vector de FIFO_TAM componentes, en nuestro caso 64 componentes. Pero es una cola circular para que cuando ya se hayan encolado *FIFO_TAM* elementos y se pueda seguir dando servicio, volveríamos al principio de la cola. Ofrece funciones como rt_FIFO_inicializar, rt_FIFO_encolar y rt_FIFO_extraer, además de una función *rt_FIFO_estadisticas* que dado un evento, nos devuelve el número de veces que fue encolado.

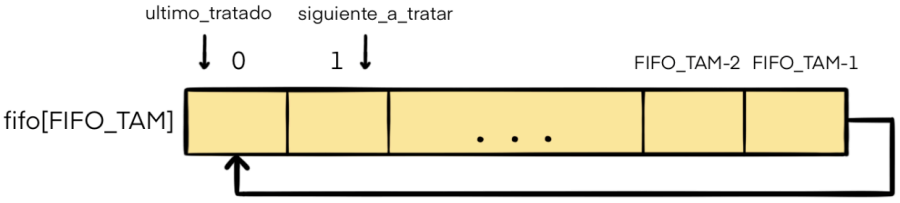

Figura 5: Dibujo de la cola FIFO circular del módulo rt_FIFO

[1]: ANEXO I: código de svc_alarma_activar

CONSUMO

El módulo del consumo proporciona funciones para reducir el consumo del procesador, poniéndolo en uno de los dos siguientes modos: Idle o Power-down. La función esperar, pone al procesador en modo Idle, es decir, para la ejecución de las instrucciones hasta que haya una interrupción o un reset, pero los periféricos siguen y pueden generar interrupciones. Por otro lado, tenemos la función dormir, que pone al procesador en modo Power-down, del que sólo sale por un reset o interrupciones externas. Este modo para todos los relojes internos además de la ejecución de instrucciones, pero mantiene el estado, y lo hemos configurado para que se despierte con la pulsación de cualquiera de los botones.

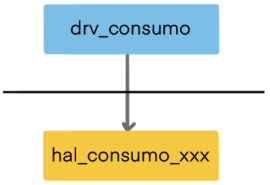

Figura 6: Diagrama de interacción entre módulos del consumo

MÓDULOS GPIO (LEDS, BOTONES Y MONITORES)

Los módulos *drv_leds*, *drv_botones* y *drv_monitor* utilizan todos el módulo ***hal_gpio_xxx***, ya que interactúan con los GPIOs del LPC2105 o del nRF52840 DK, según el proyecto que estemos probando. El módulo *hal_gpio_xxx* ofrece funciones para escribir o leer en los pines indicados, es decir, marcar un pin (encender una LED) o desmarcarlo (apagar una LED) o comprobar si está marcado o no (encendida o no). Hemos añadido funciones para los monitores, que también marcan o desmarcan el pin indicado por el nivel superior. Y ya por último, hay funciones para inicializar tanto los pines de las LEDs como los botones o los monitores.

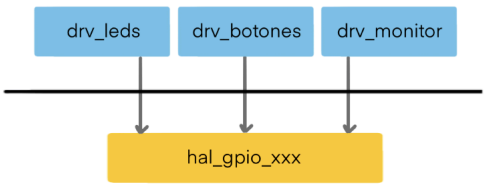

Figura 7: Diagrama de interacción entre módulos para los GPIOs

Más concretamente, el módulo ***drv_leds*** tiene las funciones *drv_leds_iniciar*, *drv_led_encender*, *drv_led_apagar* y *drv_led_conmutar*, las cuales llaman a la correspondiente función del *hal_gpio_xxx* para realizar la acción indicada.

El módulo ***drv_monitor*** es similar, ya que sus funciones *drv_monitor_iniciar_unico*, *drv_monitor_marcar* y *drv_monitor_desmarcar* también llaman a las funciones de monitores disponibles en el *hal_gpio_xxx*.

BOTONES

El módulo ***drv_botones*** es distinto, ya que únicamente llama al *hal_gpio_xxx* al iniciar en *drv_botones_iniciar*, para fijar el estado de los pines de los botones inicialmente. Esta función también invoca a la función *hal_ext_int_botones_iniciar* del módulo *hal_ext_int_xxx*,

la cual configura los pines de los botones para que generen interrupciones externas. Inicializa el estado de todos los botones a reposo y suscribe los eventos de botón a *drv_botones_tratar*.

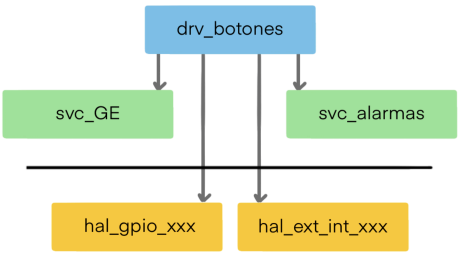

Figura 8: Diagrama de interacción entre módulos para los botones

La función *drv_botones_tratar* es la máquina de estados de los botones, la cual tiene 4 estados: *e_reposo*, *e_entrando*, *e_esperando* y *e_soltado*. Todos los botones están en *e_reposo* desde el inicio. Cuando el programa entra en *drv_botones_tratar*, y el botón está en reposo, programamos una alarma con el retardo de presión TRP y el siguiente estado será *e_entrando*. Dentro de *e_entrando*, comprobamos si se ha soltado el botón, si no se ha soltado, programamos una alarma con el retardo periódico TEP e iremos a *e_esperando*. Si se había soltado, activamos la alarma con el retardo de depresión TRD y vamos a *e_soltado*. Dentro de *e_esperando*, volvemos a hacer la misma comprobación que en *e_entrando*. Por último, vuelve a habilitar las interrupciones de ese botón, llama a la función de callback indicada al iniciar y vuelve a *e_reposo*.

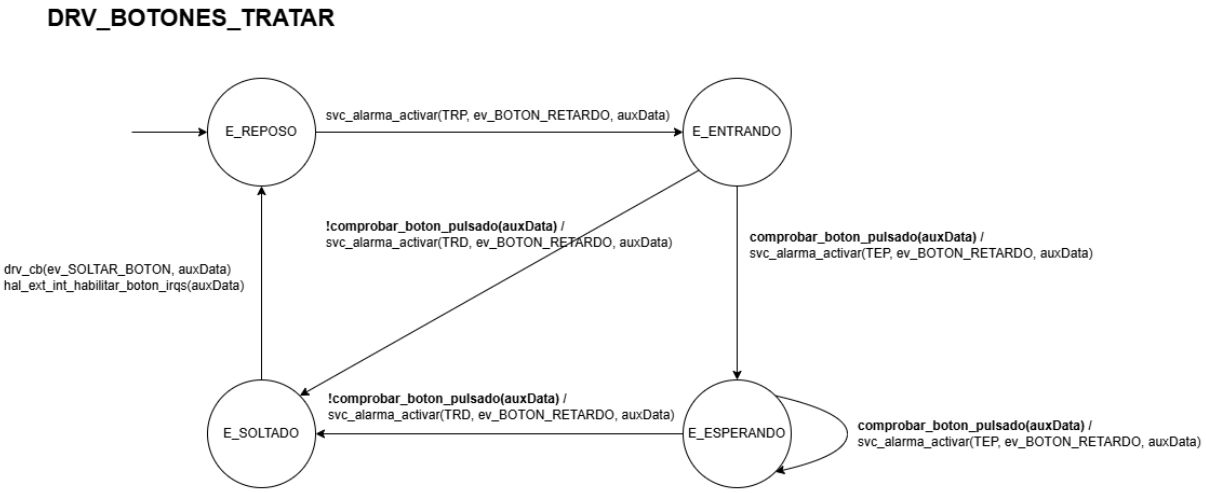

Figura 9: Diagrama de estados de los botones

RT_EVENTO_T

En nuestro diseño tenemos 7 tipos de eventos distintos: *ev_VOID* (que representa cualquiera de los siguientes eventos), *ev_T_PERIODICO* (se utiliza para alarmas periódicas), *ev_PULSAR_BOTON* (significa que el usuario ha pulsado un botón), *ev_INACTIVIDAD* (controla el tiempo de inactividad del usuario), *ev_BOTON_RETARDO* (se utiliza para programar alarmas para los botones), *ev_SOLTAR_BOTON* (significa que el usuario ha dejado de pulsar el botón) y *ev_T_ESPORADICO* (se utiliza para alarmas esporádicas).

Como eventos de usuario, hemos definido 2: *ev_PULSAR_BOTON* y *ev_SOLTAR_BOTON*, que son los eventos en los que intervienen directamente sus acciones.

```
typedef enum {
        ev_VOID = 0,
        ev_T_PERIODICO = 1,
        ev_PULSAR_BOTON = 2,
        ev_INACTIVIDAD = 3,
        ev_BOTON_RETARDO = 4,
        ev_SOLTAR_BOTON = 5,
        ev_T_ESPORADICO = 6,
} EVENTO_T;
#define EVENT_TYPES 7

#define ev_NUM_EV_USUARIO 2
#define ev_USUARIO {ev_PULSAR_BOTON,
ev_SOLTAR_BOTON}
```

Código 8: Código del archivo rt_evento_t.h

SC

También tenemos un módulo para la sección crítica, *rt_sc*, el cual proporciona las funciones *entrar_SC* y *salir_SC*, y garantiza que las instrucciones se ejecuten en exclusión mutua.

GESTOR DE EVENTOS (rt_GE y svc_GE)

El gestor de eventos es un módulo que guarda las funciones de callback de cada evento del programa en una lista de suscripciones. Si hay algún evento que tratar, este se encarga de llamar a las funciones correspondientes. El código se desarrolla todo en el archivo *rt_GE.c*, pero tenemos dos ficheros de cabecera: ***rt_GE**.h* y ***svc_GE**.h* ya que según la situación, nos interesa incluir únicamente una parte de las funciones de este módulo.

Primeramente, las funciones *svc_GE_suscribir* y *svc_GE_cancelar* añaden y eliminan respectivamente el evento indicado de la lista de suscripciones.

La función *rt_GE_iniciar* se encarga de iniciar esta lista de suscripciones vacía y suscribir los eventos de tiempo, los eventos de usuario y el evento de inactividad a la función *rt_GE_tratar*[2]. *rt_GE_lanzador*[3] es la función que está ejecutándose continuamente. Si no hay ningún evento pendiente en la cola, pone el procesador a dormir reduciendo el consumo. Si se despierta por una interrupción, extrae el/los eventos pendientes en la cola y llama a sus correspondientes funciones suscritas en la lista. *rt_GE_tratar* reprograma la alarma inactividad si ha ocurrido cualquier evento de usuario, ya que eso significa que ha habido interacción por parte del usuario. Si el evento que ha llegado es *ev_INACTIVIDAD*, pone el procesador a dormir. En cualquier caso, llama a la función de callback del juego que se ha guardado al iniciar.

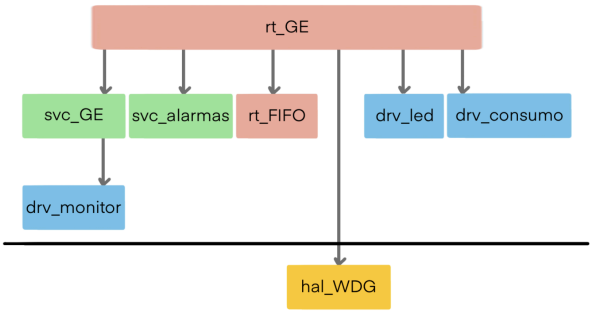

Figura 10: Diagrama de interacción entre módulos del gestor de eventos

[2]: ANEXO II: código de rt_GE_lanzador
[3]: ANEXO III: código de rt_GE_tratar

ESTADÍSTICAS

En el modo de depuración, justo cuando acabas la partida están disponibles las estadísticas del juego, las cuales se implementan en el módulo **rt_estadisticas**. Podemos observar el tiempo medio y máximo que un elemento permanece en la cola, los tiempos medio y máximo de respuesta del usuario, los tiempos medio y máximo de atender una interrupción y el tiempo que el programa permanece despierto y el que permanece dormido.

WATCHDOG

El módulo del Watchdog, **hal_WDG**, tiene como función establecer un tiempo máximo en segundos que le pasa el programador, el cual si se agota y no se le ha dado de comer en ese tiempo, puede significar que ha ocurrido algún problema en el programa ejecutado, por lo que reiniciará el sistema y volverá a empezar a ejecutar desde cero. En consecuencia, incorpora las funciones *hal_WDG_iniciar* y *hal_WDG_feed*.

4.4. TEST / VERIFICACIONES

Con el fin de probar distintos módulos que íbamos construyendo, hicimos varios test que vamos a ir explicando a continuación:

- **Test 1**[4]: Este test prueba el módulo rt_FIFO. Primeramente, inicializa el monitor que posteriormente marcará cuando detecte el overflow en la cola. Después encola *FIFO_TAM*-1 eventos periódicos en la cola para no dar overflow. Ahora revisa las estadísticas y comprueba que se hayan encolado el número correcto de eventos periódicos. Y ya finalmente, encola un último evento que provoca el overflow, marca el monitor y se queda en un bucle infinito.

  Este test ha sido desarrollado únicamente para el proyecto lpc, ya que no prueba ningún aspecto hardware, únicamente el software de la cola.

- **Test 2**[5]: Este test prueba que funcione el Watchdog, para lo que lo inicializa con un tiempo de 10 segundos, le da de comer 1 vez para comprobar que funciona esa función y ya va a la función esperar. Transcurridos 10 segundos, el Watchdog detectará que se ha agotado el tiempo y no se le ha dado de comer durante dicho tiempo, y reiniciará el programa, empezando otra vez a ejecutar el test desde el principio.

  Este test ha sido desarrollado para ambas plataformas, lpc y nrf, ya que el código en este caso es dependiente del hardware.

- **Test 3**[6]: Este último test pone el procesador a dormir y prueba que se despierte con los botones indicados específicamente. Tanto para el caso del lpc como para el del nrf, cualquiera de los botones puede despertar al procesador.

[4]: ANEXO IV: código del test 1 de rt_FIFO
[5]: ANEXO V: código del test 2 del Watchdog
[6]: ANEXO VI: código del test 3 de despertar al procesador

5. RESULTADOS

Tras la implementación de los módulos comentados en la sección anterior y las respectivas verificaciones parciales, conseguimos el funcionamiento de todos los programas, superando los objetivos impuestos para cada uno de ellos: los diferentes tipos de blinks, el bit counter strike y el juego del simón.

En lo que respecta al juego del simón, este es un programa capaz de funcionar en ambas plataformas, tanto para el LPC2105 como para el nRF52840 DK. Además, tal y como hemos diseñado el juego del simón, tenemos 3 modos de juego:

1. Modo normal: los LEDs de la secuencia aparecen de uno en uno
2. Modo rápido: la velocidad de la secuencia disminuye más rápido que en el modo normal, y los LEDs también aparecen de uno en uno
3. Modo doble: los LEDs de la secuencia aparecen de dos en dos

Nada más comenzar la partida, se mostrará la secuencia de elección de modo de juego y se encenderán las 3 primeras LEDs correspondientes a cada modo de juego. El usuario debe pulsar el botón del modo al que quiera jugar, y entonces, se mostrará la secuencia de inicio de la partida y comenzará la secuencia según el modo que haya elegido.

El simón irá mostrando la secuencia que corresponda y una vez mostrada, el usuario deberá pulsar en el mismo orden los botones indicados por la secuencia. Si se confunde de botón en algún momento, el juego mostrará la secuencia de fin y volverá a iniciar la partida. En caso contrario, el juego irá aumentando la dificultad (tanto en número de LEDs como en velocidad), hasta que el usuario falle o se rinda.

Para rendirse, el usuario deberá pulsar durante 3 segundos dos botones simultáneamente, entonces el juego mostrará la secuencia de fin y volverá a iniciar la partida desde el principio.

5.1. ESTADÍSTICAS

Seguidamente vamos a profundizar en los resultados obtenidos en una sesión de juego en el Simón. Para ello hemos jugado una partida en modo normal, durante una secuencia de 5 LEDs hasta que hemos fallado. Podemos analizar los datos específicos gracias al módulo *rt_estadistica*[7], que permite con sus operaciones mostrar el seguimiento del rendimiento del juego y la experiencia del usuario.

Name	Location/Value	Type
num_ev_VOID	209	auto - unsigned int
num_ev_T_PERIODICO	58	auto - unsigned int
num_ev_PULSAR_BOTON	17	auto - unsigned int
num_ev_INACTIVIDAD	0	auto - unsigned int
num_ev_BOTON_RETAR...	118	auto - unsigned int
num_ev_SOLTAR_BOTON	16	auto - unsigned int
num_ev_T_ESPORADICO	0	auto - unsigned int
estadistica_tiempo_me...	8.765550239234	auto - double
estadistica_tiempo_ma...	10	auto - unsigned int
estadistica_tiempo_me...	1252427.166667	auto - double
estadistica_tiempo_ma...	3814962	auto - unsigned int
estadistica_tiempo_me...	46.88235294118	auto - double
estadistica_tiempo_ma...	48	auto - unsigned int

Figura 11: Datos obtenidos en una partida en modo depuración

[7]: ANEXO VII: código para ver las estadísticas del simón

ESTADÍSTICAS DE EVENTOS Y RESPUESTAS

El número de *eventos void* muestra el total de eventos que han sido procesados durante la partida, es decir, 209. Esto sugiere que **se han realizado muchas operaciones internas que no han resultado en una acción directa visible para el jugador**, sino que principalmente han sido verificaciones de estado.
El número de *eventos periódicos* es 58, que implica que las acciones repetitivas necesarias para el juego, se manejan adecuadamente, manteniendo el juego en movimiento y reaccionando a las interacciones del usuario.
Las *pulsaciones de botón*, 17, reflejan la cantidad de veces que el jugador ha interactuado directamente con el juego. Este resultado es acorde ya que entre la selección del modo de juego y el número de veces que hay que pulsar hasta llegar al 5º botón de la secuencia número 5, suman 17.
Los *eventos de retardo de botón*, 118, muestran una **alta frecuencia de manejo de los rebotes de los botones**, lo cual es muy importante para asegurar que cada pulsación se registre correctamente una sola vez.
Los *eventos de soltar botón* son uno menos que el de pulsar botón, ya que el último botón no ha sido despulsado, por tanto los resultados cuadran correctamente.
En cuanto a los eventos que se mantienen en 0, *inactividad* y *esporádico*, estos son nulos ya que ni ha habido inactividad por parte del usuario, ni se ha intentado resetear la partida mediante el pulsado de los dos botones.

TIEMPOS DE EVENTOS SIN TRATAR

Para realizar esta medición, hemos calculado desde que se encola un evento hasta que el lanzador de *rt_GE*, lo extrae y lo trata. Observamos que el tiempo promedio que un evento pasa en la cola antes de ser tratado es de aproximadamente 8.76 microsegundos, con un máximo registrado de 10 microsegundos.
Esto sugiere que, aunque la mayoría de los eventos son procesados rápidamente, puede haber picos donde algunos eventos tarden más tiempo en ser atendidos, seguramente debido a atascos temporales. En sistemas que utilizan interrupciones, **un evento puede ser retrasado debido a la necesidad de atender otras tareas más críticas**, causando este posible atasco del que hablábamos.

TIEMPOS DE RESPUESTA DEL USUARIO

En cuanto a esta medición, se ha realizado desde que se muestra la última LED de la secuencia, hasta que el usuario reacciona y pulsa un botón (justo cuando encola el evento, ya que el tiempo desde que se maneja el pulsado en la IRQ_handler hasta que llega a *rt_FIFO* mediante la función de callback, es mínimo). El tiempo promedio de respuesta del usuario es de 1.25 segundos aproximadamente, con un tiempo máximo registrado de 3.81 segundos. La diferencia notable entre ambos tiempos, reside en la variabilidad de cómo y cuándo los usuarios responden. Esta variabilidad puede deberse a la **complejidad** de la secuencia a medida que el juego avanza y a la **carga cognitiva** que esto representa para el jugador.

TIEMPOS DE ATENDER LA INTERRUPCIÓN

Los tiempos medidos se realizan entre dos puntos: cuando se pulsa el botón (es decir, cuando se va a encolar un tipo de evento de pulsar botón) y cuando se extrae un evento de pulsar botón.
En lo que respecta a los tiempos medios y máximos, estos son 46.88 microsegundos y 48 microsegundos. Como se puede observar, hay una diferencia significativa con los tiempos de eventos sin tratar, y esto es porque los primeros analizados, se medían una vez ya se habían encolado, y los que estamos analizando actualmente, son antes de encolar. Por tanto la diferencia de tiempo entre ambos, radica principalmente en el tiempo de encolado.
Otro aspecto a destacar, es el hecho de que los **tiempos obtenidos son relativamente bajos y consistentes, que conlleva un manejo de interrupciones eficiente** (lo cual es crítico en juegos como el Simón, donde la capacidad de respuesta del sistema afecta directamente a la experiencia del usuario).

5.2. CONSUMOS

A continuación, vamos a ir analizando los consumos de cada uno de los juegos que hemos desarrollado en la asignatura.

BLINK_V2

Esta versión del blink, tal y como hemos comentado anteriormente, utiliza una espera activa para encender y apagar la LED constantemente.

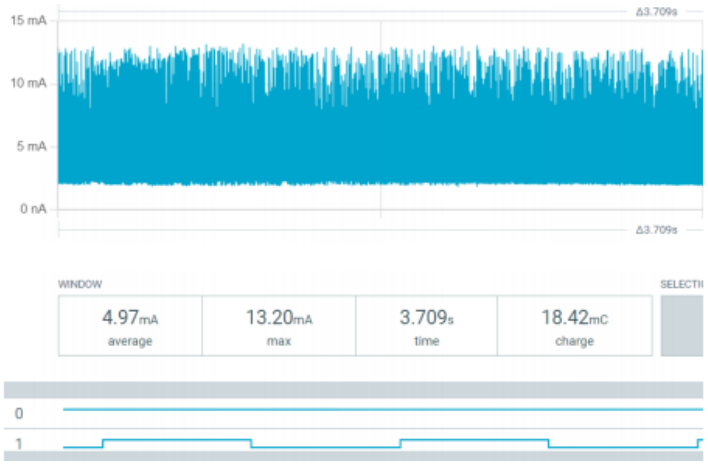

Figura 12: Imagen del consumo general del blink_v2

Debido a ello, obtenemos un **consumo medio** de **4.97 mA**, una medida muy elevada y causada por estar constantemente midiendo el tiempo actual y comprobando si hemos alcanzado el tiempo en el que queremos conmutar la LED .

De hecho, el **consumo máximo** se sitúa en **13.2 mA**, el cual está relacionado con la medición del tiempo actual.

La **duración de la medición** es de **3.709 s**, un tiempo breve pero suficiente para observar el patrón de consumo del programa. En este intervalo de tiempo, se producen unas 7 conmutaciones del LED.

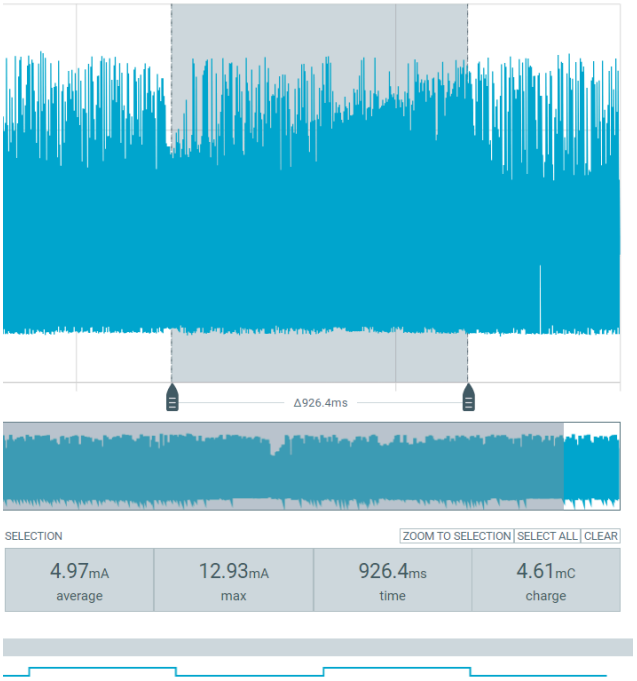

Figura 13: *Imagen del consumo de 1 parpadeo del led del blink_v2*

La **carga total** acumulada durante 1 ciclo (1 encendido del LED y 1 apagado) es de **4.61 mC**, es decir, la electricidad consumida en aproximadamente 1 segundo. Es una carga elevada ya que el procesador no entra nunca en modos de bajo consumo.

BLINK_V3

Esta segunda versión que vamos a analizar, mejoramos considerablemente con respecto a las demás versiones anteriores. En este caso, introducimos por primera vez las interrupciones periódicas, por lo que el procesador se mantiene en espera o modo de bajo consumo, hasta que le llega una interrupción, conmuta el LED y vuelve a dicho modo.

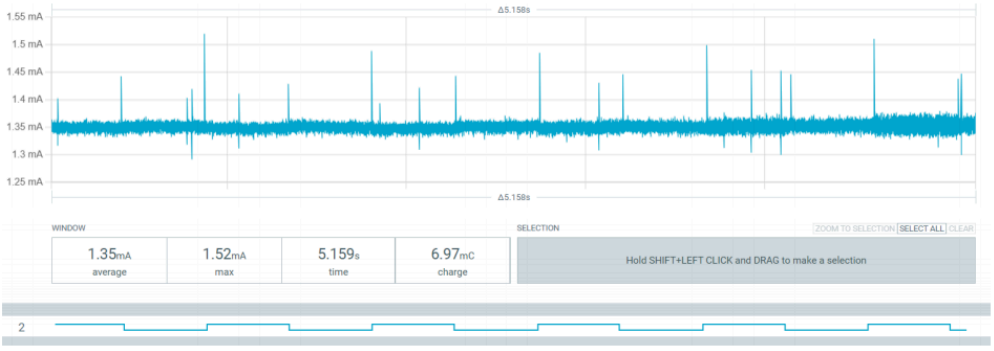

Figura 14: Imagen del consumo general del blink_v3

Como primera medida, el **consumo medio** de esta versión es de **1.35 mA**. Hemos reducido el consumo medio un 72%, lo cual conseguimos gracias a poner el procesador en modo de bajo consumo.

El **consumo máximo** está en **1.52 mA**, una medida un 88% inferior a la de la versión anterior. Ahora los picos de consumo se limitan al encendido de la LED, que es la acción que más energía consume.

La **duración de la medición** es de **5.159 segundos**, un tiempo ligeramente superior al anterior, pero el cual no nos afecta, ya que podemos observar igualmente el patrón de consumo y las distintas medidas.

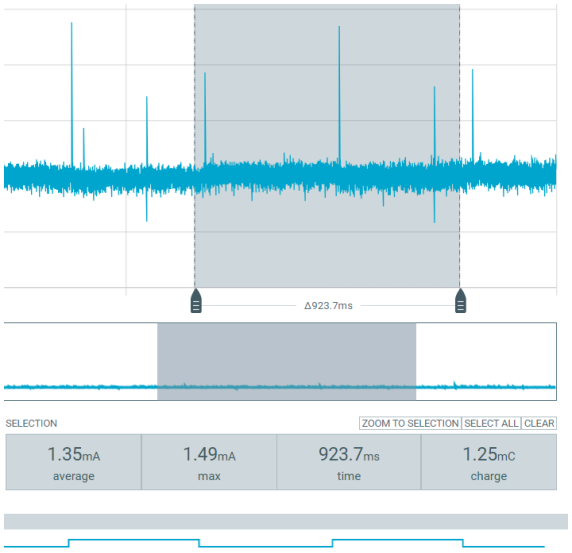

Figura 15: Imagen del consumo de 1 parpadeo del led del blink_v3

La **carga total** acumulada durante 1 ciclo (1 encendido del LED y 1 apagado) es de **1.25 mC**, un 72% menos de la carga acumulada por blink_v2. Volvemos a ratificar el gran ahorro energético que conseguimos con esta versión.

Hemos conseguido reducir el consumo más del 72% en cualquiera de las medidas, ya que estamos evitando que el procesador tenga en marcha recursos que no va a utilizar, siendo que estos consumen una gran cantidad de energía. En conclusión, la gestión de los modos de consumo del procesador es un recurso potencial para reducir la carga energética de los programas, por lo que lo aplicaremos en todas las versiones restantes.

BLINK_V3_BIS

Esta versión similar a la anterior, en lo que se diferencia es en que enciende el LED únicamente 10 veces cada 500ms, de igual modo que antes, con interrupciones periódicas y poniendo al procesador en espera. Una vez hechos los 10 parpadeos, el procesador se pone en modo power-down, el cual reduce su consumo casi a 0 y únicamente se despertará cuando el usuario pulse un botón, lo cual generará una interrupción externa y volverá a empezar de nuevo.

Figura 16: Imagen del consumo energético general del blink_v3_bis

El **consumo medio** que obtenemos es de **511.22 uA**, el cual volvemos a mejorar un 62% con respecto al anterior. Si seleccionamos sólo la parte en la que se conmutan los LEDs, la media de consumo es de **685 uA**. Sin embargo, si miramos la parte en la que el procesador está dormido, el consumo medio es de **0.46 uA**. Si no hubiésemos dormido al procesador en esa parte y lo hubiésemos mantenido activo, el 99.93% de la energía que habríamos consumido hubiera sido malgastada, ya que en realidad sólo necesitamos ese 0.07%.

El **consumo máximo** en esta versión es de **8.59 mA**. En este caso, esta medida ha aumentado, pero es debida al reseteo que realiza el sistema al volver del modo power-down, el cual necesita una gran cantidad de energía.

La **carga total** obtenida durante 1 ejecución completa (10 encendidos del LED, el procesador en modo power-down y el reseteo del inicio) es de **6.36 mC**, con una duración de aproximadamente 12 segundos. De esta carga total, el reseteo consume aproximadamente 5 uC.

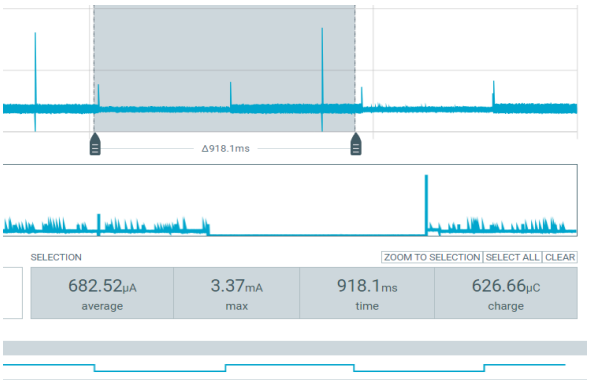

Figura 17: Imagen del consumo energético de 1 parpadeo del LED del blink_v3_bis

La **carga total** obtenida durante 1 ciclo (1 encendido del LED y 1 apagado) es de **626.66 uC**, un 49% inferior a la medida anterior.

Con esta versión sacamos en conclusión que el modo power-down es muy útil cuando queremos reducir al mínimo el consumo si nuestro procesador no tiene nada que hacer ni de lo que estar pendiente internamente. Las interrupciones externas sí que despertarán al procesador, pero cuando despierte, si que observaremos un pico de consumo ya que tiene que recuperar el estado y reiniciar sus variables.

SIMON

Dado que en el simón pueden darse distintas situaciones, primero vamos a analizar una ejecución en modo normal, en la que aparecen los elementos de la secuencia de uno en uno. Vamos a acertar todos los botones durante 3 secuencias completas, y cuando llegue la cuarta secuencia, fallamos con el primer botón.

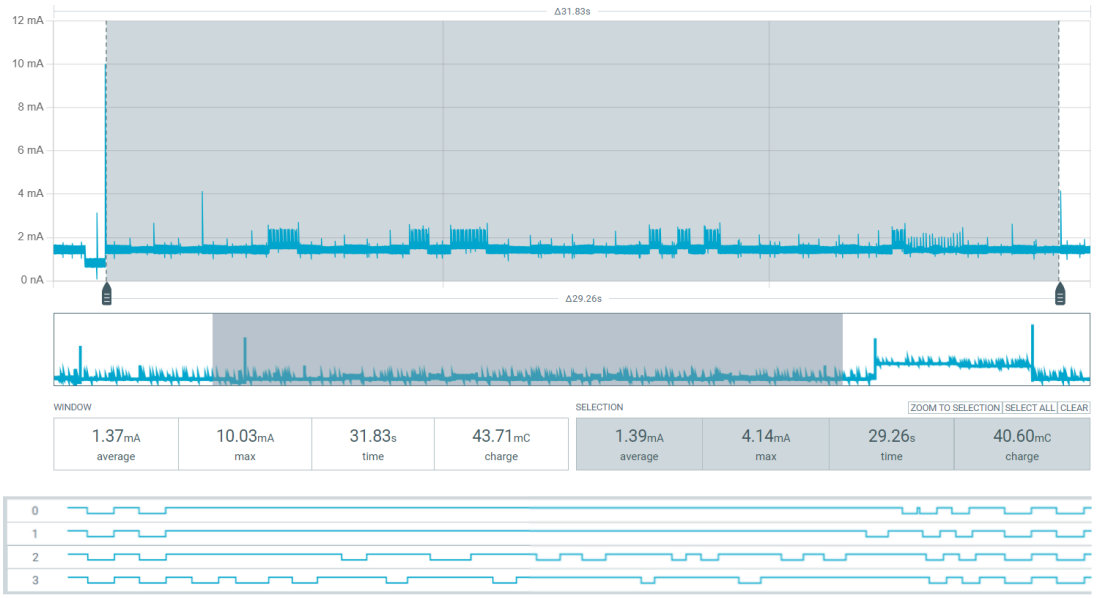

Figura 18: Imagen del consumo general del simón con 3 secuencias completas y fallo en el primer elemento de la cuarta secuencia

El **consumo medio** obtenido durante esta ejecución es de **1.39 mA**. Es un consumo bastante reducido para la cantidad de tareas que está haciendo el procesador, ya que está programando distintos eventos periódicos o no, atendiendo las interrupciones tanto temporales como de botones, entre otros. Pero conseguimos esta medida gracias a que ponemos el procesador en espera entre cada una de estas tareas, por lo que sólo tendrá un mayor consumo durante la realización de estas tareas.

El **consumo máximo** se sitúa en **4.14 mA** justo en el momento en el que finalizan las secuencias de inicio o de fin. En este momento es cuando el procesador calcula la secuencia aleatoria de encendido de los LEDs, por lo que esta es la tarea de mayor consumo del programa.

La **carga total** obtenida durante esta ejecución es de **40.60 mC** durante **29.26 segundos**. Esta es la carga energética consumida para mostrar las secuencias de inicio y fin, atender 7 interrupciones de botón con las alarmas que estos conllevan e iniciar la secuencia aleatoria. La carga de la secuencia de inicio junto con la inicialización de todas las estructuras del main es de aproximadamente 4 mc. Mientras que la carga de la secuencia de fin es de 3.3 mC.

A continuación, analizaremos el consumo durante 5 segundos en los que no hay interacción del usuario ni del sistema, por lo que transcurrido este tiempo, el Watchdog hará un reset.

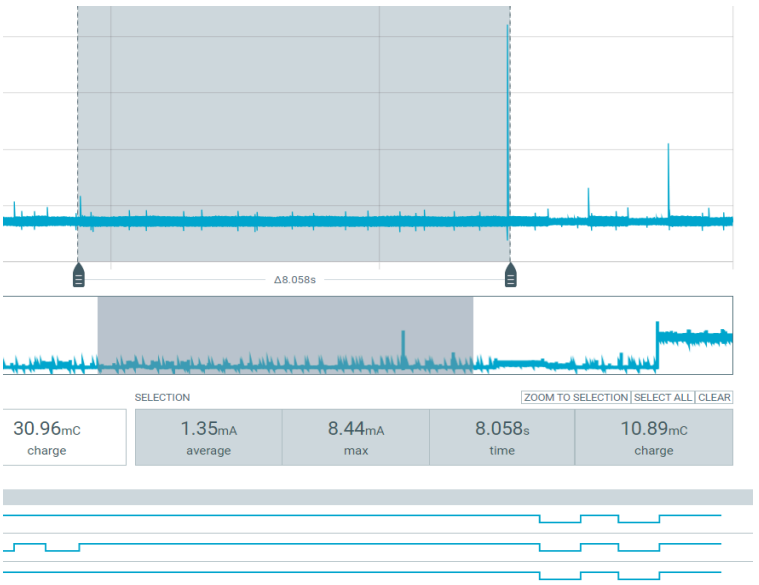

Figura 19: Imagen del consumo energético del reseteo del Watchdog en el simón

El **consumo medio** en esta etapa es de **1.35 mA**, prácticamente igual al obtenido anteriormente con una ejecución normal.

El **consumo máximo** aumenta a **8.44 mA**, debido a que el Watchdog produce un reset del microcontrolador.

La **carga total** de energía consumida durante los 5 segundos sin interacción y el reset es de **10.89 mC**, un número elevado debido principalmente al reset del Watchdog, ya que el resto del tiempo, el procesador se mantiene en espera hasta que llega la alarma del evento *ev_T_ESPORADICO*.

Por último, vamos a analizar una ejecución del simón en la que el usuario se rinde a la primera, pulsando simultáneamente durante 3 segundos los botones 3 y 4.

Figura 20: Imagen del consumo energético del simón con la pulsación simultánea de 2 botones para rendirse

El **consumo medio** se sigue manteniendo similar en **1.48 mA**.

El **consumo máximo** vuelve a una medida similar a la primera obtenida, **4.21 mA**. En este caso, como no hay reseteo, vuelve a deberse a iniciar el vector de la secuencia aleatoria de LEDs.

La **carga total** de esta ejecución es de **16.03 mC**, la cual se resume en iniciar todas las estructuras, mostrar la secuencia de inicio, iniciar la secuencia aleatoria, mostrar el primer elemento de la secuencia, atender la doble pulsación y mostrar la secuencia de fin. Es una medida razonable para las tareas que realiza, ya que tiene que atender numerosas interrupciones y gestionar múltiples alarmas.

En conclusión, hemos conseguido un juego eficiente energéticamente, lo que hemos logrado gracias a un buen diseño inicial, junto con la utilización efectiva de los recursos que nos ofrece el procesador y que hemos ido aprendiendo con cada versión del blink. Nuestro objetivo ha sido disminuir lo máximo posible el consumo energético, identificando las situaciones en las que el procesador no necesita apenas recursos y poniéndolo en espera o a dormir. Por ejemplo, la carga energética correspondiente al encendido de un LED la hemos reducido un 86%. Lo mismo ha ocurrido con otras medidas que hemos ido explicando.

6. CONCLUSIONES

Para concluir, queremos destacar que hemos cumplido los objetivos planteados inicialmente. Hemos conseguido un proyecto compatible con ambas plataformas, tanto con el *LPC2105* como con el *nRF52840 DK*, con un código sólido e independiente que gestiona todo el sistema de interrupciones y eventos, y con una pequeña parte del código que implementa funciones específicas y dependientes del hardware en concreto. Esto lo hemos conseguido gracias al diseño modular en el que tenemos drivers o gestores y una capa de abstracción del hardware, tal y como explicamos en el apartado del diseño.

Nuestro objetivo principal ha sido desarrollar un programa eficiente energéticamente, que haga una buena gestión de los recursos del procesador, intentando evitar consumos innecesarios. Para esto nos han sido muy útiles las funciones de esperar y dormir, que reducen al máximo posible el consumo energético del procesador en cada situación. A grandes rasgos, hemos conseguido reducir la carga energética que supone 1 parpadeo del LED en un 86%, así como hemos reducido el consumo máximo un 79%. Todo ello repercute directamente en una mayor duración de la batería del dispositivo en el que estemos ejecutando nuestro programa, por lo que tenemos menos dependencia energética y de carga, así como reducimos el mantenimiento y el enfriamiento de los dispositivos. Si extrapolamos nuestro pequeño programa a que esté ejecutándose simultáneamente en miles o millones de dispositivos, la demanda energética global disminuye considerablemente.

Finalmente, otro de nuestros objetivos al desarrollar el simón ha sido mejorar la experiencia del usuario, favoreciendo la jugabilidad y la atracción del usuario. Esto lo hemos logrado creando 3 modos de juego distintos, que el usuario puede elegir según sus intereses, y ajustando los tiempos de pulsado y reacción del programa para que el usuario sienta que sus acciones tienen efecto en el tiempo real.

Nosotras creemos que hemos cumplido con los objetivos de la asignatura, realizando a tiempo las entregas, corrigiendo práctica a práctica aquellos aspectos a mejorar y midiendo los consumos en las sesiones correspondientes. Creemos que hemos entendido el propósito de la asignatura, que no es solo desarrollar el juego del simón, sino conseguir un juego energéticamente eficiente, aprovechando al máximo los recursos disponibles y disminuyendo los consumos de cada versión desarrollada. En conclusión, estamos contentas con nuestro trabajo y nos pondríamos un notable.

7. RESUMEN EJECUTIVO

Este documento presenta los resultados y metodologías del proyecto que hemos llevado a cabo durante este curso, el cual se centraba en la interacción con plataformas de hardware específicas y el desarrollo de software eficiente energéticamente. Los dos objetivos principales del proyecto eran:

1) **Interacción con plataformas hardware**: se logró un entendimiento global y profundo del entorno LPC2105 y nRF52840 DK, adaptando el código para asegurar la compatibilidad entre las diferentes plataformas hardware.
2) **Desarrollo de programas eficientes en energía**: se analizó y optimizó el consumo energético identificando y mejorando las áreas del código que necesitaban una cantidad mayor de energía.

No obstante, además de estos objetivos, se abordaron otros aspectos cruciales como la optimización del tiempo de uso de la placa, el dominio de los timers en la programación, la implementación de múltiples mains en un solo programa, etc. Cada uno de estos elementos contribuyó a crear el proyecto que finalmente hemos realizado.

El entorno que utilizamos fue el entorno de desarrollo **Keil uVision5** para el diseño, compilación y depuración del código. La medición del consumo energético se realizó utilizando la herramienta **Power Profiler** en **nRF Connect**, proporcionando datos precisos para poder emplearlos continuamente para analizarlos.

Los resultados fueron altamente satisfactorios, **demostrando que los objetivos del proyecto se alcanzaron con éxito**. El proyecto concluyó con el desarrollo del juego del Simón, juego que permite **jugar** a los usuarios **con fluidez** y permite **realizar todas las acciones posibles evitando paros o problemas en la ejecución**. Se permite, desde elegir un modo de juego (tanto normal, como rápido o doble), visualizar la secuencia de leds, aumentar la velocidad de visionado de leds, pulsar botones gestionando los rebotes, reiniciar el juego gracias al pulsado de dos botones durante 3 segundos y mostrar estadísticas jugando en modo depuración. Todo esto se logró manteniendo un consumo energético adecuado y gestionando eficientemente las condiciones de carrera a través de secciones críticas y el watchdog.

El consumo energético obtenido en las distintas mediciones de cada versión del blink fue mejorando, logrando disminuir la carga total del parpadeo de un led en un 86%, o el consumo máximo en un 79%.

En resumen, este proyecto no solo ha cumplido con sus objetivos iniciales, sino que también ha superado nuestras expectativas en cuanto al funcionamiento correcto de todo el proyecto.

8. REFERENCIAS

Arduino. (n.d.). Arduino Nano 33 BLE nRF52840 Product Specification. Retrieved from *https://content.arduino.cc/assets/Nano_BLE_MCU-nRF52840_PS_v1.1.pdf*

Challenge validation. (s. f.-a). https://community.arm.com/support-forums/f/keil-forum/40982/uvision-include-file-dependencies-not-available-problem

Challenge validation. (s. f.-b). https://community.arm.com/support-forums/f/keil-forum/33484/if-x-0-conditional-compilation-does-not-work

GmbH, A. L. A. A. G. (s. f.-a). *Interrupt Simulation in the Keil uVision Debugger*. Copyright (C) 2024, ARM Ltd And ARM Germany GmbH. All Rights Reserved. https://www.keil.com/products/uvision/db_sim_interrupts.asp

GmbH, A. L. A. A. G. (s. f.-b). *Power-Saving Modes in µVision*. Copyright (C) 2024, ARM Ltd And ARM Germany GmbH. All Rights Reserved. https://www.keil.com/products/uvision/db_sim_prf_power.asp

GmbH, A. L. A. A. G. (s. f.-c). *STM32 Timer example*. Copyright (C) 2024, ARM Ltd And ARM Germany GmbH. All Rights Reserved. https://www.keil.com/download/docs/363.asp

GmbH, A. L. A. A. G. (s. f.-d). *µVision User's Guide: Simulate Interrupts and Clock Inputs*. Copyright (C) 2024, ARM Ltd And ARM Germany GmbH. All Rights Reserved. https://www.keil.com/support/man/docs/uv4cl/uv4cl_db_simulinterpts_clk_inpts.htm

Interrupt handling with 8051 C using Keil uVision. (s. f.). Stack Overflow. https://stackoverflow.com/questions/20862964/interrupt-handling-with-8051-c-using-keil-uvision

Interrupts and Exceptions. (s. f.). https://www.keil.com/pack/doc/CMSIS_Dev/Core_A/html/group__irq__ctrl__gr.html

nRF52840 Advanced Development With the nRF5 SDK - SparkFun Learn. (s. f.). https://learn.sparkfun.com/tutorials/nrf52840-advanced-development-with-the-nrf5-sdk/all

NRF52840 DK. (s. f.). nordicsemi.com. https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK

nRF Connect for Desktop. (s. f.). nordicsemi.com. https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop

NXP Semiconductors. (2016). LPC17xx User Manual. Retrieved from *https://www.nxp.com/docs/en/user-guide/UM10275.pdf*

Power Profiler Kit II. (s. f.). nordicsemi.com. https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2

9. ANEXOS

ANEXO I: svc_alarma_activar

```
void svc_alarma_activar(uint32_t retardo_ms, EVENTO_T ID_evento, uint32_t auxData) {
	// Si retardo_ms es 0, desprogramamos la alarma
	if (retardo_ms == 0) {
		num_alarmas_programadas--;

		// Reinicializamos los valores de la alarma
		alarmas_programadas[ID_evento].activada = false;
		alarmas_programadas[ID_evento].esPeriodica = false;
		alarmas_programadas[ID_evento].retardo_ms = 0;
		alarmas_programadas[ID_evento].ID_evento = ev_VOID;
		alarmas_programadas[ID_evento].auxData = 0;

		if (ID_evento == ev_INACTIVIDAD) {
			drv_tiempo_programar_alarma(0, 0, false, svc_alarma_tratar);
		}
		else {
			drv_tiempo_programar_alarma(1, 0, false, svc_alarma_tratar);
		}
	}
	else {
	// Si ya hay programadas el máximo de alarmas, activamos el monitor y entramos en un bucle infinito
		if (alarmas_programadas[ID_evento].activada == false) {
			if (num_alarmas_programadas == svc_ALARMAS_MAX) {
				drv_monitor_marcar(overflow_monitor_pin);
				while(1) {}
			}
			num_alarmas_programadas++;
		}

		uint8_t esPeriodica = (retardo_ms >> 31);
		retardo_ms &= 0x7FFFFFFF;

		alarmas_programadas[ID_evento].activada = true;
		alarmas_programadas[ID_evento].esPeriodica = esPeriodica;
		alarmas_programadas[ID_evento].retardo_ms = retardo_ms;
		alarmas_programadas[ID_evento].ID_evento = ID_evento;
		alarmas_programadas[ID_evento].auxData = auxData;

		if (ID_evento == ev_INACTIVIDAD) {
			drv_tiempo_programar_alarma(0, retardo_ms, esPeriodica, svc_alarma_tratar);
		}
		else {
			drv_tiempo_programar_alarma(1, retardo_ms, esPeriodica, svc_alarma_tratar);
		}
	}
}
```

ANEXO II: rt_GE_lanzador

```
void rt_GE_lanzador(void) {
  // Activa una alarma de inactividad que se disparará después de 20 segundos
        svc_alarma_activar(TIEMPO_INACTIVIDAD, ev_INACTIVIDAD, 0/*, rt_FIFO_encolar*/);

        EVENTO_T EV_ID_evento;
        uint32_t EV_auxData;
        Tiempo_us_t EV_TS;

        void (*callback)(EVENTO_T, uint32_t);

        // Bucle que procesa eventos en la cola
        while (1) {
                // Extraer eventos de la cola
                if (rt_FIFO_extraer(&EV_ID_evento, &EV_auxData, &EV_TS)){
                        // Mandar los eventos a sus tareas suscritas
                        hal_WDG_feed();
                        for (int i=0; i<lista_suscripciones[EV_ID_evento].num_suscritos; i++) {
                                // Eventos específicos
                                callback = lista_suscripciones[EV_ID_evento].f_callbacks[i];
                                callback(EV_ID_evento, EV_auxData);
                        }
                }
                else{
                        // Si no hay eventos, el sistema entra en modo de espera para reducir consumo
                        drv_consumo_esperar();
                }
        }
}
```

ANEXO III: rt_GE_tratar

```
void rt_GE_tratar(uint32_t ID_evento, uint32_t auxData) {
        if (ID_evento == ev_PULSAR_BOTON) {
                // Reprogramar la alarma de inactividad
                svc_alarma_activar(TIEMPO_INACTIVIDAD, ev_INACTIVIDAD, 0);
        }
        else if (ID_evento == ev_SOLTAR_BOTON) {
                // Reprogramar la alarma de inactividad
                svc_alarma_activar(TIEMPO_INACTIVIDAD, ev_INACTIVIDAD, 0);
        }
        else if (ID_evento == ev_INACTIVIDAD) {
                for(uint8_t i=1; i<=4; i++) {
                        drv_led_apagar(i);
                }
                drv_consumo_dormir();
        }

        (*juego)(ID_evento, auxData);
}
```

ANEXO IV: test 1

```
void run_tests_fifo(uint32_t pin_monitor_overflow) {
	uint32_t pin = pin_monitor_overflow;
	rt_FIFO_inicializar(pin);

	uint32_t auxData = 100; // valor auxiliar para cada evento

	// Test básico de encolar eventos hasta antes de overflow (FIFO_TAM-1 eventos)
	for (int i = 0; i < FIFO_TAM-1; i++) {
		rt_FIFO_encolar(ev_T_PERIODICO, auxData + i);
	}

	// Revisa las estadísticas
	uint32_t num_VOID = rt_FIFO_estadisticas(ev_VOID);
	uint32_t num_T_PERIODICO = rt_FIFO_estadisticas(ev_T_PERIODICO);
	uint32_t num_PULSAR_BOTON = rt_FIFO_estadisticas(ev_PULSAR_BOTON);

	// Test básico de overflow (FIFO_TAM eventos en la cola)
	// => marcará monitor 4
	rt_FIFO_encolar(ev_T_PERIODICO, auxData + FIFO_TAM-1);
}
```

ANEXO V: test 2

```
hal_WDG_iniciar(10);
hal_WDG_feed();
drv_consumo_esperar();
```

ANEXO VI: test 3

```
uint8_t num_botones = drv_botones_iniciar(rt_FIFO_encolar);
drv_consumo_dormir();
// Para despertar al procesador: cualquier botón
```

ANEXO VII: estadísticas

```
#if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
	uint32_t num_ev_VOID = rt_FIFO_estadisticas(ev_VOID);
	uint32_t num_ev_T_PERIODICO = rt_FIFO_estadisticas(ev_T_PERIODICO);
	uint32_t num_ev_PULSAR_BOTON = rt_FIFO_estadisticas(ev_PULSAR_BOTON);
	uint32_t num_ev_INACTIVIDAD = rt_FIFO_estadisticas(ev_INACTIVIDAD);
	uint32_t num_ev_BOTON_RETARDO = rt_FIFO_estadisticas(ev_BOTON_RETARDO);
	uint32_t num_ev_SOLTAR_BOTON = rt_FIFO_estadisticas(ev_SOLTAR_BOTON);
	uint32_t num_ev_T_ESPORADICO = rt_FIFO_estadisticas(ev_T_ESPORADICO);

	double estadistica_tiempo_medio_cola = get_tiempo_medio_cola();
	uint32_t estadistica_tiempo_max_cola = get_tiempo_max_cola();

	double estadistica_tiempo_medio_respuesta_usuario = get_tiempo_medio_respuesta_usuario();
	uint32_t estadistica_tiempo_max_respuesta_usuario = get_tiempo_max_respuesta_usuario();

	double estadistica_tiempo_medio_atender_irq = get_tiempo_medio_tratar_irq();
	uint32_t estadistica_tiempo_max_atender_irq = get_tiempo_max_tratar_irq();
#endif
```

ANEXO VIII: todo el código

main.c

```
/*
 * P.H.2024: Main
 *
 * Según el modo de compilación, ejecutaremos distintos programas:
 *		- modo_simon_lpc y modo_simon_nrf -> ejecutarán el juego del simón
 *  - modo_depuracion_lpc y modo_depuracion_nrf -> ejecutarán el juego del simón y mostrarán las estadísticas al terminar
 *		- modo_test_fifo -> ejecutará una prueba de rt_FIFO_estadísticas y el overflow en la cola de eventos
 *  - modo_test_WDG_lpc y modo_test_WDG_nrf -> ejecutará una prueba del Watchdog que se reiniciará transcurridos 10 segundos
 *  - modo_test_inactividad_lpc y modo_test_inactividad_nrf -> llevará el procesador a dormir y sólo lo podremos despertar con los botones indicados posteriormente
 */

#if defined(modo_simon_lpc) || defined(modo_simon_nrf)
	#include "simon.h"
	#include "board.h"
#elif defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
	#include "simon.h"
	#include "board.h"
	#include "rt_estadisticas.h"
#elif defined(modo_test_fifo)
	#include "test_rt_fifo.h"
	#include "board.h"
#elif defined(modo_test_WDG_lpc) || defined(modo_test_WDG_nrf)
	#include "hal_WDG.h"
	#include "drv_consumo.h"
#elif defined(modo_test_inactividad_lpc) || defined(modo_test_inactividad_nrf)
	#include "drv_consumo.h"
	#include "drv_botones.h"
	#include "rt_fifo.h"
#endif


int main(void) {
	#if defined(modo_simon_lpc) || defined(modo_simon_nrf) || defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
		hal_gpio_iniciar();
		uint32_t num_Leds = drv_leds_iniciar();
		if (num_Leds > 0){
			drv_tiempo_iniciar();
			hal_WDG_iniciar(6);
```

```
                        rt_FIFO_inicializar(MONITOR4);       // monitor 4 para overflow en la
cola
                        rt_GE_iniciar(MONITOR1, maq_estados_simon);    // monitor 1 para
overflow en el gestor de eventos
                        svc_alarma_iniciar(MONITOR2, rt_FIFO_encolar, ev_T_PERIODICO,
svc_alarma_codificar(1,VELOCIDAD_SECUENCIA_INICIO)); // monitor 2 para overflow en
svc_alarmas

                        #if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
                                actualizar_estadisticas_empezando_a_ejecutar();
                        #endif

                        iniciar_simon();
                }


        #elif defined(modo_test_fifo)
                // PRUEBA DE OVERFLOW AL ENCOLAR EVENTOS:
                run_tests_fifo(MONITOR4);

        #elif defined(modo_test_WDG_lpc) || defined(modo_test_WDG_nrf)
                hal_WDG_iniciar(10);
                hal_WDG_feed();
                drv_consumo_esperar();

        #elif defined(modo_test_inactividad_lpc) || defined(modo_test_inactividad_nrf)
                uint8_t num_botones = drv_botones_iniciar(rt_FIFO_encolar);
                drv_consumo_dormir();
                // Para despertar al procesador: cualquier botón

        #endif
}
```

board.h

```
/* ************************************************************************
 * P.H.2024: define la placa de desarrollo con la que estamos trabajando (pines, numero de
leds, etc
 * forma parte del HAL
 */

#ifndef BOARD
#define BOARD

#if defined(modo_simon_lpc) || defined(modo_depuracion_lpc)
        #include "board_lpc.h"
#elif defined(modo_test_fifo)
        #include "board_lpc.h"
```

```
#elif defined(modo_test_WDG_lpc)
        #include "board_lpc.h"
#elif defined(modo_test_inactividad_lpc)
        #include "board_lpc.h"
#elif defined(modo_simon_nrf) || defined(modo_depuracion_nrf)
        #include "board_nrf52840dk.h"
  #include "board_nrf52840_dongle.h"
#elif defined(modo_test_WDG_nrf)
        #include "board_nrf52840dk.h"
  #include "board_nrf52840_dongle.h"
#elif defined(modo_test_inactividad_nrf)
        #include "board_nrf52840dk.h"
  #include "board_nrf52840_dongle.h"
#elif defined(BOARD_PCA10056)
        #include "board_nrf52840dk.h"
#elif defined(BOARD_PCA10059)
  #include "board_nrf52840_dongle.h"
#else
        #error "Board is not defined"
#endif

#endif
```

drv_tiempo.h

```
/* ****************************************************************************
 * P.H.2024: Driver/Manejador de los temporizadores
 * suministra los servicios independientemente del hardware
 */

#ifndef DRV_TIEMPO
#define DRV_TIEMPO

#include <stdint.h>
#include <stdbool.h>

/*
 * define los tipos de datos asociados a tiempo de uso en la app
 */
typedef uint64_t Tiempo_us_t;
typedef uint32_t Tiempo_ms_t;


void drv_tiempo_iniciar(void);

Tiempo_us_t drv_tiempo_actual_us(void);
```

```
Tiempo_ms_t drv_tiempo_actual_ms(void);

void drv_tiempo_esperar_ms(Tiempo_ms_t ms);

Tiempo_ms_t drv_tiempo_esperar_hasta_ms(Tiempo_ms_t ms);

void drv_tiempo_periodico_ms(Tiempo_ms_t ms, void(*funcion_callback)(), uint32_t id);

void drv_tiempo_programar_alarma(uint8_t timer, uint32_t retardo_ms, uint8_t esPeriodica,
void(*funcion_callback)());

#endif
```

drv_tiempo.c

```
/* ********************************************************************************
 * P.H.2024: Driver/Manejador de los temporizadores
 * suministra los servicios independientemente del hardware
 *
 * usa los servicos de hal_tiempo.h:
 */

#include "drv_tiempo.h"
#include "hal_tiempo.h"
#define          US2MS                                                    1000
                                    //milisegundos por microsegundos

static volatile uint32_t HAL_TICKS2US = 0;

/**
 * inicializa el reloj y empieza a contar
 */
void drv_tiempo_iniciar(void){
        HAL_TICKS2US = hal_tiempo_iniciar_tick();
}

/*
 * tiempo desde que se inicio el temporizador en microsegundos
 */
Tiempo_us_t drv_tiempo_actual_us(void){
        uint32_t ticks_actual = hal_tiempo_actual_tick();
        return ticks_actual / HAL_TICKS2US;
}

/*
 * tiempo desde que se inicio el temporizador en milisegundos
 */
```

```
Tiempo_ms_t drv_tiempo_actual_ms(void){
	uint32_t ticks_us = drv_tiempo_actual_us();
	return ticks_us / US2MS;
}

/*
 * esperar hasta un determinado tiempo (en ms)
 */
void drv_tiempo_esperar_ms(Tiempo_ms_t ms){
	uint32_t us_inicio = drv_tiempo_actual_us();
	uint32_t us_a_esperar = ms * US2MS;

	while ((drv_tiempo_actual_us() - us_inicio) < us_a_esperar);

}

/*
 * esperar hasta un determinado tiempo (en ms), devuelve el tiempo actual
 */
Tiempo_ms_t drv_tiempo_esperar_hasta_ms(Tiempo_ms_t ms){
  uint32_t us_objetivo = ms * US2MS;

  while (drv_tiempo_actual_us() < us_objetivo);

  return drv_tiempo_actual_ms();
}


/*
 * ptr guarda la función de callback, y callback_id guarda un parámetro de esa función
 */
void (*ptr)();
static uint32_t callback_id;

/*
 * llama a la función de callback
 */
void funcion_que_llama() {
	(*ptr)(callback_id, drv_tiempo_actual_ms(), 0);
}

/*
 * llama a programar una interrupción pasándole la función de callback que debe ejecutar
 * cuando la interrupción salte
 */
void drv_tiempo_periodico_ms(Tiempo_ms_t ms,  void(*funcion_callback)(), uint32_t id) {
	ptr = funcion_callback;
	callback_id = id;
```

```
	hal_tiempo_reloj_periodico_tick( ms*HAL_TICKS2US*US2MS, funcion_que_llama,
id);
}


void drv_tiempo_programar_alarma(uint8_t timer, uint32_t retardo_ms, uint8_t esPeriodica,
void(*funcion_callback)()) {
	hal_tiempo_reloj_tick(timer, retardo_ms*HAL_TICKS2US*US2MS, esPeriodica,
funcion_callback);
}
```

drv_leds.h

```
/* ************************************************************************
 * P.H.2024: Driver/Manejador de los Leds
 * suministra los servicios de iniciar, encender, apagar, conmutar... independientemente del
hardware
 */

#ifndef DRV_LEDS
#define DRV_LEDS

#include <stdint.h>

uint32_t drv_leds_iniciar(void);

void drv_led_encender(uint32_t id);

void drv_led_apagar(uint32_t id);

void drv_led_conmutar(uint32_t id);

#endif
```

drv_leds.c

```
/* ************************************************************************
 * P.H.2024: Driver/Manejador de los Leds
 * suministra los servicios de iniciar, encender, apagar, conmutar... independientemente del
hardware
 *
 * recoge de la definicion de la placa:
 * LEDS_NUMBER - numero de leds existentes
 * LEDS_LIST - array de leds que existen y sus gpios
 * el gpio de cada LED_x
 * LEDS_ACTIVE_STATE - si los leds son activos con el gpio a nivel alto o bajo
```

```
 *
 * Usa los servicios y tipos de datos del hal_gpio.h
 */

#include "hal_gpio.h"
#include "drv_leds.h"
#include "board.h"


#if LEDS_NUMBER > 0
	static const uint8_t led_list[LEDS_NUMBER] = LEDS_LIST;
#endif


/**
 * inicializa los led, los deja apagados y devuelve el numero de leds disponibles en la
plataforma
 */
uint32_t drv_leds_iniciar(){
	#if LEDS_NUMBER > 0
		for (uint32_t i = 0; i < LEDS_NUMBER; ++i)			{
			hal_gpio_sentido(led_list[i], HAL_GPIO_PIN_DIR_OUTPUT);
			drv_led_apagar(i+1);
		}
  #endif //LEDS_NUMBER > 0

	return LEDS_NUMBER;
}

/**
 * funcion para encender led con identificador id
 */
void drv_led_encender(uint32_t id){
	#if LEDS_NUMBER > 0
		if ((id <= LEDS_NUMBER) && (id >0)) hal_gpio_escribir(led_list[id-1],
LEDS_ACTIVE_STATE);
  #endif //LEDS_NUMBER > 0
}


/*
 * funcion para apagar led con identificador id
 */
void drv_led_apagar(uint32_t id){
	#if LEDS_NUMBER > 0
		if ((id <= LEDS_NUMBER) && (id >0)) hal_gpio_escribir(led_list[id-1],
~LEDS_ACTIVE_STATE);
  #endif //LEDS_NUMBER > 0
```

```
}


/*
 * conmuta el led de on a off y viceversa
 * primero consulta el estado y lo invierte
 */
void drv_led_conmutar(uint32_t id){
	#if LEDS_NUMBER > 0
		if ((id <= LEDS_NUMBER) && (id >0)){
			hal_gpio_escribir(led_list[id-1], ~hal_gpio_leer(led_list[id-1]));
		}
  #endif //LEDS_NUMBER > 0
}
```

drv_botones.h

```
/* ********************************************************************************
 * P.H.2024: Driver/Manejador de los Leds
 * suministra los servicios de iniciar, encender, apagar, conmutar... independientemente del
hardware
 */

#ifndef DRV_BOTONES
#define DRV_BOTONES

#include <stdint.h>

uint32_t drv_botones_iniciar(void(*funcion_callback)());

void drv_botones_tratar(uint32_t ID_evento, uint32_t auxData);


#endif
```

drv_botones.c

```
/* ********************************************************************************
 * P.H.2024: Driver/Manejador de los Botones
 * suministra los servicios independientemente del hardware
 *
 * proporciona funciones para iniciar los botones
 *		y tratar las pulsaciones
 */

#include "drv_botones.h"
```

```
#include "hal_gpio.h"
#include "hal_ext_int.h"

#include "svc_GE.h"
#include "svc_alarmas.h"

#include "board.h"

#if defined(modo_simon_lpc) || defined(modo_depuracion_lpc) || defined(modo_test_fifo) ||
defined(modo_test_WDG_lpc) || defined(modo_test_inactividad_lpc)
	#define	TRP		100
	#define		TEP		400
	#define		TRD		100
#endif
#if defined(modo_simon_nrf) || defined(modo_depuracion_nrf) ||
defined(modo_test_WDG_nrf) || defined(modo_test_inactividad_nrf)
	#define	TRP		10
	#define		TEP		40
	#define		TRD		10
#endif


// Lista con todos los botones disponibles
//#if BUTTONS_NUMBER > 0
//	static const uint8_t buttons_list[BUTTONS_NUMBER] = BUTTONS_LIST;
//#endif

// Definición de los distintos posibles estados de los botones
typedef enum {
	e_reposo,
	e_entrando,
	e_esperando,
	e_soltado
} ESTADO_ACTUAL;

// Vector para guardar el estado de cada botón
static ESTADO_ACTUAL estado_boton[BUTTONS_NUMBER];

void (*ptr_botones)();


void drv_botones_tratar_irq(uint32_t boton) {
	(*ptr_botones)(ev_PULSAR_BOTON, boton);
}

/*
 * inicializa todos los botones como pines de entrada
 * y devuelve el número de botones disponibles
```

```
 */
uint32_t drv_botones_iniciar(void(*funcion_callback)()) {
	ptr_botones = funcion_callback;

	#if BUTTONS_NUMBER > 0
		hal_gpio_botones_iniciar();
		hal_ext_int_botones_iniciar(drv_botones_tratar_irq);
  #endif //BUTTONS_NUMBER > 0

	// Inicializa el estado de cada botón a estado e_reposo
	for (uint32_t i=0 ; i< BUTTONS_NUMBER ; i++){
		estado_boton[i] = e_reposo;
	}

	// suscribe los eventos, y así, si se detecta que un botón ha sido
	// pulsado, que ha habido un retardo porque el botón sigue presionado
	// tras un tiempo (para evitar rebotes) o si se ha soltado el botón,
	// se manejarán con la función drv_botones_tratar
	svc_GE_suscribir(ev_PULSAR_BOTON, drv_botones_tratar);
	svc_GE_suscribir(ev_BOTON_RETARDO, drv_botones_tratar);
	//svc_GE_suscribir(ev_SOLTAR_BOTON, drv_botones_tratar);

	return BUTTONS_NUMBER;
}

/*
 * gestiona los eventos relacionados con los botones. Se llama a esta función
 * cada vez que se detecta un evento para los botones.
 */
void drv_botones_tratar(uint32_t ID_evento, uint32_t auxData) {

	if (estado_boton[auxData-1] == e_reposo) {
		// Cambia el estado a entrando
		estado_boton[auxData-1] = e_entrando;

		// Activa una alarma que revisará el estado del botón después de un breve
retardo
		svc_alarma_activar(TRP, ev_BOTON_RETARDO, auxData);	// rebote
de presión
	}
	else if (estado_boton[auxData-1] == e_entrando) {
		// Comprueba si el botón sigue siendo presionado
		if (comprobar_boton_pulsado(auxData)) {
			// Si el botón sigue presionado, activa una alarma para volver a
comprobar en 1 segundo
			estado_boton[auxData-1] = e_esperando;
			svc_alarma_activar(TEP, ev_BOTON_RETARDO, auxData);
		}
```

```
                else{
                        // Si el botón ya no está presionado, cambia el estado a soltado
//                      estado_boton[auxData-1] = e_soltado;
//                      svc_alarma_activar(TRD, ev_BOTON_RETARDO, auxData);        //
rebote de depresión
                        // Vuelve al estado de reposo y habilita las interrupciones de botones
                        estado_boton[auxData-1] = e_reposo;
                        // Encola el evento de 'soltar botón'
                        (*ptr_botones)(ev_SOLTAR_BOTON, auxData);
                        // Habilitamos las interrupciones del botón
                        hal_ext_int_habilitar_boton_irqs(auxData);
                }
        }
        else if (estado_boton[auxData-1] == e_esperando) {
                // Si el botón ya no está presionado
                if (!comprobar_boton_pulsado(auxData)) {
//                      estado_boton[auxData-1] = e_soltado;
//                      svc_alarma_activar(TRD, ev_BOTON_RETARDO, auxData);        //
rebote de depresión
                        // Vuelve al estado de reposo y habilita las interrupciones de botones
                        estado_boton[auxData-1] = e_reposo;
                        // Encola el evento de 'soltar botón'
                        (*ptr_botones)(ev_SOLTAR_BOTON, auxData);
                        // Habilitamos las interrupciones del botón
                        hal_ext_int_habilitar_boton_irqs(auxData);
                }
                else {
                        // Si sigue presionado, programa otra comprobación en 40 ms
                        svc_alarma_activar(TEP, ev_BOTON_RETARDO, auxData);        //
encuesta periódica
                }
        }
        else if (estado_boton[auxData-1] == e_soltado) {
                // Vuelve al estado de reposo y habilita las interrupciones de botones
                estado_boton[auxData-1] = e_reposo;

                // Encola el evento de 'soltar botón'
                (*ptr_botones)(ev_SOLTAR_BOTON, auxData);

                // Habilitamos las interrupciones del botón
                hal_ext_int_habilitar_boton_irqs(auxData);
        }
        else {
                // Si el estado es indefinido, entra en un bucle infinito (indica un error)
                while (1) {}
        }
}
```

drv_monitor.h

```
#ifndef DRV_MONITOR
#define DRV_MONITOR

#include <stdint.h>


uint32_t drv_monitor_iniciar(void);

uint32_t drv_monitor_iniciar_unico(uint32_t id);

void drv_monitor_marcar(uint32_t id);

void drv_monitor_desmarcar(uint32_t id);

#endif
```

drv_monitor.c

```
/* ********************************************************************************
 * P.H.2024: Driver/Manejador de los monitores
 * suministra los servicios independientemente del hardware
 *
 * proporciona las funciones para iniciar,
 *                  marcar y desmarcar los monitores
 */


#include "drv_monitor.h"
#include "board.h"
#include "hal_gpio.h"


/**
 * inicializa los monitores
 */
#if MONITOR_NUMBER > 0
        static const uint8_t monitor_list[MONITOR_NUMBER] = MONITOR_LIST;
#endif

/*
 * inicializa todos los monitores
 */
uint32_t drv_monitor_iniciar(){
        #if MONITOR_NUMBER > 0
                        hal_gpio_iniciar_monitor();
  #endif
```

```
	return MONITOR_NUMBER;
}

/*
 * inicializa un monitor en concreto
 */
uint32_t drv_monitor_iniciar_unico(uint32_t id){
	#if MONITOR_NUMBER > 0
				if ((id <= MONITOR_NUMBER) && (id >0))
hal_gpio_iniciar_monitor_unico(monitor_list[id-1]);
  #endif

	return MONITOR_NUMBER;
}

/*
 * marcará el monitor indicado por el id, poniendo en estado alto el pin correspondiente.
 */
void drv_monitor_marcar(uint32_t id){
	hal_gpio_monitor_marcar(id);
}

/*
 * desmarcará el monitor indicado por el id, poniendo en estado alto el pin correspondiente.
 */
void drv_monitor_desmarcar(uint32_t id){
	hal_gpio_monitor_desmarcar(id);
}
```

drv_consumo.h

```
/* *
* P.H.2024:
*/

#ifndef DRV_CONSUMO
#define DRV_CONSUMO

#include <stdint.h>

void drv_consumo_iniciar(uint32_t id_monitor);

void drv_consumo_esperar(void);

void drv_consumo_dormir(void);

#endif
```

drv_consumo.c

```
/* ********************************************************************************
 * P.H.2024: Driver/Manejador del consumo
 * suministra los servicios independientemente del hardware
 *
 * proporciona las funciones para dormir
 *                  y poner en espera al procesador
 */

#include "drv_consumo.h"

#include "drv_monitor.h"
#include "hal_consumo.h"
#include "board.h"      // para la lista de monitores

#if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
        #include "rt_estadisticas.h"
#endif

#if MONITOR_NUMBER > 0
        static const uint8_t monitor_list[MONITOR_NUMBER] = MONITOR_LIST;
#endif

/*
 * no hace nada
 */
void drv_consumo_iniciar(uint32_t id_monitor){}

/*
 * pone al programa en espera
 */
void drv_consumo_esperar(void){
        drv_monitor_marcar(monitor_list[3]); // para medir consumo

        #if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
                actualizar_estadisticas_entrando_a_esperar();
        #endif

        hal_consumo_esperar();

        #if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
                actualizar_estadisticas_saliendo_de_esperar();
        #endif

        drv_monitor_desmarcar(monitor_list[3]);       // para medir consumo
}
```

```
/*
 * pone al programa a dormir, en modo power_down
 */
void drv_consumo_dormir(void){
	drv_monitor_marcar(monitor_list[3]); // para medir consumo

	hal_consumo_dormir();

	drv_monitor_desmarcar(monitor_list[3]);	// para medir consumo
}
```

hal_gpio.h

```
/* ************************************************************************
 * P.H.2024: hal_gpio, interface que nos independiza del hardware concreto
 */

#ifndef HAL_GPIO
#define HAL_GPIO

#include <stdint.h>
#include <stdbool.h>

/*
 * Dirección de los registros de GPIO (E o S)
 */
enum {
	HAL_GPIO_PIN_DIR_INPUT = 0,
	HAL_GPIO_PIN_DIR_OUTPUT = 1,
} typedef hal_gpio_pin_dir_t;

/**
 * Tipo de datos para los pines
 */
typedef uint32_t HAL_GPIO_PIN_T;


void hal_gpio_iniciar(void);

void hal_gpio_sentido(HAL_GPIO_PIN_T gpio, hal_gpio_pin_dir_t direccion);
uint32_t hal_gpio_leer(HAL_GPIO_PIN_T gpio);
void hal_gpio_escribir(HAL_GPIO_PIN_T gpio, uint32_t valor);

void hal_gpio_sentido_n(HAL_GPIO_PIN_T gpio_inicial, uint8_t num_bits,
hal_gpio_pin_dir_t direccion);
uint32_t hal_gpio_leer_n(HAL_GPIO_PIN_T gpio_inicial, uint8_t num_bits);
```

```
void hal_gpio_escribir_n(HAL_GPIO_PIN_T bit_inicial, uint8_t num_bits, uint32_t valor);


void hal_gpio_iniciar_monitor(void);
void hal_gpio_iniciar_monitor_unico(uint32_t id_monitor);
void hal_gpio_monitor_marcar(uint32_t id);
void hal_gpio_monitor_desmarcar(uint32_t id);


void hal_gpio_botones_iniciar(void);

#endif
```

hal_tiempo.h

```
/* ************************************************************************
 * P.H.2024: hal_tiempos, interface que nos independiza del hardware
 */

#ifndef HAL_TIEMPO
#define HAL_TIEMPO

#include <stdint.h>
#include <stdbool.h>

uint32_t hal_tiempo_iniciar_tick(void);

uint64_t hal_tiempo_actual_tick(void);

void hal_tiempo_reloj_periodico_tick(uint32_t periodo_en_tick, void(*funcion_callback)(),
uint32_t id);

void hal_tiempo_reloj_tick(uint8_t timer, uint32_t periodo_en_tick, bool esPeriodica,
void(*funcion_callback)());

#endif
```

hal_ext_int.h

```
/* ************************************************************************
 * P.H.2024: Driver/Manejador de las interrupciones
 * suministra los servicios independientemente del hardware
 *
 * proporciona funciones para habilitar y deshabilitar las interrupciones
 *                          iniciar los botones y comprobar si hay alguno pulsado
 */
```

```
#ifndef HAL_EXT_INT
#define HAL_EXT_INT

#include <stdint.h>

void hal_ext_int_deshabilitar_botones_irqs(void);
void hal_ext_int_habilitar_botones_irqs(void);

void hal_ext_int_deshabilitar_boton_irqs(uint8_t boton);
void hal_ext_int_habilitar_boton_irqs(uint8_t boton);


void hal_ext_int_botones_iniciar(void(*funcion_callback)());

uint32_t comprobar_boton_pulsado(uint8_t boton);

uint8_t num_botones_pulsados_realmente(void);


#endif
```

hal_consumo.h

```
/* *
* P.H.2024: hal_tiempos, interface que nos independiza del hardware
*/

#ifndef HAL_CONSUMO
#define HAL_CONSUMO

#include <stdint.h>


void hal_consumo_iniciar(void);

void hal_consumo_esperar(void);

void hal_consumo_dormir(void);

#endif
```

hal_WDG.h

```
/* ******************************************************************************
 * P.H.2024: Driver/Manejador del Watchdog
 */
```

```
#ifndef HAL_WDG
#define HAL_WDG

#include <stdint.h>

void hal_WDG_iniciar(uint32_t sec);

void hal_WDG_feed(void);

#endif
```

rt_evento_t.h

```
/* ****************************************************************************
 * P.H.2024: Definición de los eventos del procesador
 */

#ifndef RT_EVENTO_T
#define RT_EVENTO_T


typedef enum {
	ev_VOID = 0,
	ev_T_PERIODICO = 1,
	ev_PULSAR_BOTON = 2,
	ev_INACTIVIDAD = 3,
	ev_BOTON_RETARDO = 4,
	ev_SOLTAR_BOTON = 5,
	ev_T_ESPORADICO = 6,
} EVENTO_T;

#define EVENT_TYPES 7

#define ev_NUM_EV_USUARIO 2
#define ev_USUARIO {ev_PULSAR_BOTON, ev_SOLTAR_BOTON}

#endif
```

rt_sc.h

```
/* ****************************************************************************
 * P.H.2024: Driver/Manejador de la sección crítica
 *
 * proporciona funciones para entrar y salir de la sección crítica
 */

#ifndef RT_SC
#define RT_SC
```

```
void entrar_SC(void);

void salir_SC(void);

#endif
```

rt_sc.c

```
/*
 * P.H.2024: Módulo para gestionar la sección crítica
 */

#include "rt_sc.h"
#include <stdint.h>


void entrar_SC(void) {
  // Deshabilita las interrupciones globalmente (modo sección crítica)
        __disable_irq();
}

void salir_SC(void) {
        // Habilita las interrupciones globales (modo normal)
        __enable_irq();
}
```

rt_fifo.h

```
#ifndef RT_FIFO
#define RT_FIFO

#include <stdint.h>
#include "rt_evento_t.h"
#include "drv_tiempo.h"

#define FIFO_TAM 64

void rt_FIFO_inicializar(uint32_t pin_monitor_overflow);

void rt_FIFO_encolar(uint32_t ID_evento, uint32_t auxData);

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

uint32_t rt_FIFO_estadisticas(EVENTO_T ID_evento);

#endif
```

rt_fifo.c

```
/* ******************************************************************************
 * P.H.2024: Módulo para manejar una cola FIFO de eventos
 *
 * proporciona las funciones para inicializar,
 *                  encolar y extraer elementos de la cola,
 *                  y para imprimir las estadísticas
 */

#include "rt_fifo.h"
#include "rt_sc.h"
#include "drv_monitor.h"

#if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
        #include "rt_estadisticas.h"
#endif

typedef struct {
        EVENTO_T ID_EVENTO;
        uint32_t auxData;
        Tiempo_us_t TS;
} EVENTO;

typedef uint8_t indice_cola_t;

// Cola de eventos de tamaño FIFO_TAM
static EVENTO fifo[FIFO_TAM];

// variables para gestionar la cola
static indice_cola_t ultimo_tratado = 0;                                        // Índice
del último evento tratado
static indice_cola_t siguiente_a_tratar = 0;                                    // Índice del
siguiente evento a tratar

// Pin del monitor para detectar overflow
static uint32_t overflow_monitor_pin;

// contador para llevar registro del número de eventos de cada tipo de evento
static uint32_t contador_eventos[EVENT_TYPES] = {0};

/*
 * inicializa la cola FIFO y el monitor de overflow
 */
void rt_FIFO_inicializar(uint32_t pin_monitor_overflow){
        // resetea los índices de la cola
        ultimo_tratado = 0;
```

```
  siguiente_a_tratar = 0;

        // asigna el pin para el monitor de overflow
  overflow_monitor_pin = pin_monitor_overflow;

        // Inicializa el contador de eventos a 0
        for (int i=0; i<EVENT_TYPES; i++) {
                contador_eventos[i] = 0;
        }
}

/*
 * encola un nuevo evento en la cola, y se le pasa el identificador del evento, un
 * auxData donde suele pasarse muchas veces el tiempo, y un boton que es para guardar
 * a veces el botón (si no se desea porque es irrelevante, se le pasa un 0).
 */
void rt_FIFO_encolar(uint32_t ID_evento, uint32_t auxData){
        #if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
                if (ID_evento == ev_PULSAR_BOTON) {
                        actualizar_estadisticas_tras_respuesta_usuario();
                        actualizar_estadisticas_antes_de_atender_irq();
                }
        #endif

        entrar_SC();

        // Guardamos el número de eventos encolados totales
        contador_eventos[ev_VOID] = contador_eventos[ev_VOID] + 1;
        indice_cola_t siguiente_pos = (siguiente_a_tratar + 1);

        // si hemos llegado al final de la cola, reiniciamos a la posición 0
        if (siguiente_pos == FIFO_TAM){
                siguiente_pos=0;
        }

  // Verifica si la cola está llena (overflow)
  if (siguiente_pos == ultimo_tratado) {
    // si está llena, activa el monitor del overflow y entra en bucle infinito
                drv_monitor_marcar(overflow_monitor_pin);
    while (1) {}
        }

  // Encola el evento
  fifo[siguiente_a_tratar].ID_EVENTO = (EVENTO_T)ID_evento;
  fifo[siguiente_a_tratar].auxData = auxData;
  fifo[siguiente_a_tratar].TS = drv_tiempo_actual_us();

        // Incrementa el contador de eventos para el tipo de evento encolado
```

```
        if (ID_evento<EVENT_TYPES){
                contador_eventos[ID_evento]++;
        }

  // Actualiza el índice siguiente_a_tratar para el proximo evento
  siguiente_a_tratar = siguiente_pos;
        salir_SC();
}

/*
 * extrae un evento de la cola FIFO, y se podrá acceder a los valores extraídos gracias
 * a los punteros.
 * Devuelve 1 si extrajo un evento, y 0 si la cola está vacía.
 */
uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t* auxData, Tiempo_us_t *TS){
        entrar_SC();
  // La cola está vacía, no hay eventos para extraer
        if (ultimo_tratado == siguiente_a_tratar) {
                salir_SC();
    return 0;
  }

        // La cola no está vacía, y extrae el evento del índice último_tratado
        *ID_evento = fifo[ultimo_tratado].ID_EVENTO;
  *auxData = fifo[ultimo_tratado].auxData;
  *TS = fifo[ultimo_tratado].TS;

  // Actualiza ultimo_tratado para que avance al siguiente evento en la cola
        ultimo_tratado++;
        if (ultimo_tratado == FIFO_TAM){
                ultimo_tratado = 0;
        }
        salir_SC();

        #if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
                actualizar_estadisticas_cola(*TS);
                if (*ID_evento == ev_PULSAR_BOTON) {
                        actualizar_estadisticas_tras_atender_irq();
                }
        #endif

        return 1;
}

/*
 * devuelve el número de eventos encolados de un evento en concreto.
 */
uint32_t rt_FIFO_estadisticas(EVENTO_T ID_evento){
```

```
        if (ID_evento < EVENT_TYPES) {
                return contador_eventos[ID_evento];
  }
  return 0;
}
```

rt_estadisticas.h

```
#ifndef RT_ESTADISTICAS
#define RT_ESTADISTICAS

#include <stdint.h>

void actualizar_estadisticas_cola(uint32_t tiempo_anterior);
double get_tiempo_medio_cola(void);
uint32_t get_tiempo_max_cola(void);

// ---------------------------------------------------------------

void actualizar_estadisticas_antes_de_respuesta_usuario(void);
void actualizar_estadisticas_tras_respuesta_usuario(void);
double get_tiempo_medio_respuesta_usuario(void);
uint32_t get_tiempo_max_respuesta_usuario(void);

// ---------------------------------------------------------------

void actualizar_estadisticas_antes_de_atender_irq(void);
void actualizar_estadisticas_tras_atender_irq(void);
double get_tiempo_medio_tratar_irq(void);
uint32_t get_tiempo_max_tratar_irq(void);

// ---------------------------------------------------------------

void actualizar_estadisticas_empezando_a_ejecutar(void);
void actualizar_estadisticas_entrando_a_esperar(void);
void actualizar_estadisticas_saliendo_de_esperar(void);
uint64_t get_tiempo_despierto(void);
uint64_t get_tiempo_esperando(void);

#endif
```

rt_estadisticas.c

```
/*
 * P.H.2024: Módulo para gestionar las estadísticas
 *
 * proporciona funciones para obtener los tiempos medios y máximos de:
 *		- la cola de eventos
 *  - el tiempo de respuesta del usuario
 *  - el tiempo de tratar la interrupción de pulsar el botón
 */

#include "rt_estadisticas.h"
#include "drv_tiempo.h"
#include "rt_sc.h"

static double tiempo_medio_cola = 0.0;
static uint32_t tiempo_max_cola = 0;
static uint32_t num_encolados = 0;

void actualizar_estadisticas_cola(uint32_t tiempo_anterior) {
	uint32_t tiempo_actual_cola = drv_tiempo_actual_us();
	uint32_t tiempo_tratar_cola;
	if (tiempo_actual_cola < tiempo_anterior) {
		tiempo_anterior = 0xFFFFFFFF - tiempo_anterior;
		tiempo_tratar_cola = tiempo_anterior + tiempo_actual_cola;
	}
	else {
		tiempo_tratar_cola = tiempo_actual_cola - tiempo_anterior;
	}
	num_encolados++;

	double tiempo_aux = (tiempo_medio_cola * (num_encolados - 1) +
tiempo_tratar_cola) / num_encolados;
	tiempo_medio_cola = tiempo_aux;

	if (tiempo_tratar_cola > tiempo_max_cola) {
		tiempo_max_cola = tiempo_tratar_cola;
	}
}

/* Devuelve el tiempo medio que el programa tarda en tratar cualquier evento encolado en
us */
double get_tiempo_medio_cola(void) {
	return tiempo_medio_cola;
}

/* Devuelve el tiempo máximo que el programa tarda en tratar cualquier evento encolado en
us */
```

```
uint32_t get_tiempo_max_cola(void) {
	return tiempo_max_cola;

}

// ------------------------------------------------------------------------

static double tiempo_medio_respuesta_usuario;
static uint32_t tiempo_max_respuesta_usuario;
static uint8_t ya_ha_respondido = 0;
static uint32_t tiempo_antes_de_respuesta;
static uint32_t num_respuestas_usuario;

void actualizar_estadisticas_antes_de_respuesta_usuario(void) {
	tiempo_antes_de_respuesta = drv_tiempo_actual_us();
	ya_ha_respondido = 0;
}

void actualizar_estadisticas_tras_respuesta_usuario(void) {
	if (ya_ha_respondido == 0) {
		ya_ha_respondido = 1;
		num_respuestas_usuario++;

		uint32_t tiempo_tras_respuesta = drv_tiempo_actual_us();
		uint32_t tiempo_respuesta_usuario;
		if (tiempo_tras_respuesta < tiempo_antes_de_respuesta) {
			tiempo_antes_de_respuesta = 0xFFFFFFFF -
tiempo_antes_de_respuesta;
			tiempo_respuesta_usuario = tiempo_antes_de_respuesta +
tiempo_tras_respuesta;
		}
		else {
			tiempo_respuesta_usuario = tiempo_tras_respuesta -
tiempo_antes_de_respuesta;
		}

		tiempo_medio_respuesta_usuario = (tiempo_medio_respuesta_usuario *
(num_respuestas_usuario - 1) + tiempo_respuesta_usuario) / num_respuestas_usuario;

		if (tiempo_respuesta_usuario > tiempo_max_respuesta_usuario) {
			tiempo_max_respuesta_usuario = tiempo_respuesta_usuario;
		}
	}
}

/* Devuelve el tiempo medio que un usuario tarda en pulsar un botón en us */
double get_tiempo_medio_respuesta_usuario(void) {
	return tiempo_medio_respuesta_usuario;
```

```
}

/* Devuelve el tiempo máximo que un usuario tarda en pulsar un botón en us */
uint32_t get_tiempo_max_respuesta_usuario(void) {
	return tiempo_max_respuesta_usuario;
}


// -------------------------------------------------------------------------------

static double tiempo_medio_irq;
static uint32_t tiempo_max_irq;
static uint8_t ya_se_ha_tratado_irq = 0;
static uint32_t tiempo_antes_de_tratar_irq;
static uint32_t num_irqs_tratadas;

void actualizar_estadisticas_antes_de_atender_irq(void) {
	tiempo_antes_de_tratar_irq = drv_tiempo_actual_us();
	ya_se_ha_tratado_irq = 0;
}

void actualizar_estadisticas_tras_atender_irq(void) {
	if (ya_se_ha_tratado_irq == 0) {
		ya_se_ha_tratado_irq = 1;
		num_irqs_tratadas++;

		uint32_t tiempo_tras_atender_irq = drv_tiempo_actual_us();
		uint32_t tiempo_atender_irq;
		if (tiempo_tras_atender_irq < tiempo_antes_de_tratar_irq) {
			tiempo_antes_de_tratar_irq = 0xFFFFFFFF -
tiempo_antes_de_tratar_irq;
			tiempo_atender_irq = tiempo_antes_de_tratar_irq +
tiempo_tras_atender_irq;
		}
		else {
			tiempo_atender_irq = tiempo_tras_atender_irq -
tiempo_antes_de_tratar_irq;
		}

		tiempo_medio_irq = (tiempo_medio_irq * (num_irqs_tratadas - 1) +
tiempo_atender_irq) / num_irqs_tratadas;

		if (tiempo_atender_irq > tiempo_max_irq) {
			tiempo_max_irq = tiempo_atender_irq;
		}
	}
}
```

```
/* Devuelve el tiempo medio que el programa tarda en tratar el pulsado de un botón en us */
double get_tiempo_medio_tratar_irq(void) {
	return tiempo_medio_irq;
}

/* Devuelve el tiempo máximo que el programa tarda en tratar el pulsado de un botón en us
*/
uint32_t get_tiempo_max_tratar_irq(void) {
	return tiempo_max_irq;
}


// -------------------------------------------------------------------------------

uint64_t tiempo_despierto = 0;
uint64_t tiempo_esperando = 0;
uint64_t tiempo_inicial_despierto;
uint64_t tiempo_inicial_esperando;

void actualizar_estadisticas_empezando_a_ejecutar(void) {
	tiempo_inicial_despierto = drv_tiempo_actual_us();
}

void actualizar_estadisticas_entrando_a_esperar(void) {
	uint64_t tiempo_actual = drv_tiempo_actual_us();

	uint64_t calc_tiempo_despierto;
	if (tiempo_actual < tiempo_inicial_despierto) {
		calc_tiempo_despierto = 0xFFFFFFFF - tiempo_inicial_despierto +
tiempo_actual;
	}
	else {
		calc_tiempo_despierto = tiempo_actual-tiempo_inicial_despierto;
	}

	tiempo_despierto += calc_tiempo_despierto;
	tiempo_inicial_esperando = tiempo_actual;
}

void actualizar_estadisticas_saliendo_de_esperar(void) {
	uint64_t tiempo_actual = drv_tiempo_actual_us();

	uint64_t calc_tiempo_esperando;
	if (tiempo_actual < tiempo_inicial_esperando) {
		calc_tiempo_esperando = 0xFFFFFFFF - tiempo_inicial_esperando +
tiempo_actual;
	}
	else {
```

```
                calc_tiempo_esperando = tiempo_actual-tiempo_inicial_esperando;
        }

        tiempo_esperando += calc_tiempo_esperando;
        tiempo_inicial_despierto = tiempo_actual;
}

uint64_t get_tiempo_despierto(void) {
        return tiempo_despierto;
}

uint64_t get_tiempo_esperando(void) {
        return tiempo_esperando;
}
```

rt_GE.h

```
/* ************************************************************************
 * P.H.2024: Módulo del gestor de eventos
 */

#ifndef RT_GE
#define RT_GE

#include <stdint.h>

void rt_GE_iniciar(uint32_t pin_monitor_overflow, void(*juego_callback)());

void rt_GE_lanzador(void);

void rt_GE_tratar(uint32_t ID_evento, uint32_t auxData);

#endif
```

rt_GE.c

```
/* ************************************************************************
 * P.H.2024: Módulo del gestor de eventos
 *
 * proporciona las funciones para iniciar,
 *                 lanzar el gestor y tratar los eventos
 * también inicializa una secuencia aleatoria de 64 números
 *                 entre 1 y num_botones
 */
#include <stdint.h>
#include <stddef.h>  // Para definir NULL
```

```
#include "rt_GE.h"
#include "svc_GE.h"

#include "rt_fifo.h"
#include "rt_evento_t.h"
#include "svc_alarmas.h"

#include "drv_monitor.h"
#include "drv_consumo.h"
#include "drv_leds.h"

#include "hal_WDG.h"

#define  rt_GE_MAX_SUSCRITOS            4
#define          TIEMPO_INACTIVIDAD                      20000

typedef struct {
        uint32_t ID_evento;
        uint8_t num_suscritos;
        void (*f_callbacks[rt_GE_MAX_SUSCRITOS])(EVENTO_T, uint32_t);
} SUSCRIPCION;


const EVENTO_T eventos_usuario[ev_NUM_EV_USUARIO] = ev_USUARIO; // Eventos
definidos por el usuario
static SUSCRIPCION lista_suscripciones[EVENT_TYPES];          // Lista de suscripciones

void (*juego)();
static uint32_t overflow_monitor_pin;


/*
 * Función para inicializar el gestor de eventos y las suscripciones
 */
void rt_GE_iniciar(uint32_t pin_monitor_overflow,  void(*juego_callback)()) {
        juego = juego_callback;
        overflow_monitor_pin = pin_monitor_overflow;

        // Inicializa la lista de suscripciones vacía
        for (int i=0; i<EVENT_TYPES; i++) {
                lista_suscripciones[i].ID_evento = ev_VOID;
                lista_suscripciones[i].num_suscritos = 0;               // No hay suscriptores al
inicio
                for (int j=0; j<rt_GE_MAX_SUSCRITOS; j++) {
                        lista_suscripciones[i].f_callbacks[j] = NULL;
                }
        }
```

```
  // Suscribe los eventos que no son del usuario a rt_GE_tratar
        svc_GE_suscribir(ev_INACTIVIDAD, rt_GE_tratar);
        svc_GE_suscribir(ev_T_PERIODICO, rt_GE_tratar);
        svc_GE_suscribir(ev_T_ESPORADICO, rt_GE_tratar);

        // Suscribe los eventos del usuario a rt_GE_tratar
        for (int i=0; i<ev_NUM_EV_USUARIO; i++) {
                svc_GE_suscribir(eventos_usuario[i], rt_GE_tratar);
        }
}

/*
 * Función que se ejecuta en un bucle infinito, gestionando eventos del sistema
 */
void rt_GE_lanzador(void) {
  // Activa una alarma de inactividad que se disparará después de 20 segundos
        svc_alarma_activar(TIEMPO_INACTIVIDAD, ev_INACTIVIDAD, 0/*,
rt_FIFO_encolar*/);

        EVENTO_T EV_ID_evento;
        uint32_t EV_auxData;
        Tiempo_us_t EV_TS;

        void (*callback)(EVENTO_T, uint32_t);

        // Bucle que procesa eventos en la cola
        while (1) {
                // Extraer eventos de la cola
                if (rt_FIFO_extraer(&EV_ID_evento, &EV_auxData, &EV_TS)){
                        // Mandar los eventos a sus tareas suscritas
                        hal_WDG_feed();
                        for (int i=0; i<lista_suscripciones[EV_ID_evento].num_suscritos; i++) {
                                // Eventos específicos
                                callback = lista_suscripciones[EV_ID_evento].f_callbacks[i];
                                callback(EV_ID_evento, EV_auxData);
                        }
                }
                else{
                        // Si no hay eventos, el sistema entra en modo de espera para reducir
consumo
                        drv_consumo_esperar();
                }
        }
}

/*
 * Función para tratar diferentes tipos de eventos
 */
```

```
void rt_GE_tratar(uint32_t ID_evento, uint32_t auxData) {
	if (ID_evento == ev_PULSAR_BOTON) {
		// Reprogramar la alarma de inactividad
		svc_alarma_activar(TIEMPO_INACTIVIDAD, ev_INACTIVIDAD, 0);
	}
	else if (ID_evento == ev_SOLTAR_BOTON) {
		// Reprogramar la alarma de inactividad
		svc_alarma_activar(TIEMPO_INACTIVIDAD, ev_INACTIVIDAD, 0);
	}
	else if (ID_evento == ev_INACTIVIDAD) {
		for(uint8_t i=1; i<=4; i++) {
			drv_led_apagar(i);
		}
		drv_consumo_dormir();
	}

	(*juego)(ID_evento, auxData);
}




/*
 * Función para suscribir una función de callback a un evento específico.
 * Si el número de suscripciones excede el límite, se produce un overflow.
 */
void svc_GE_suscribir(uint32_t ID_evento, void(*f_callback)()/*, uint32_t auxData*/) {
  // Verifica si el número actual de suscripciones ha alcanzado el máximo permitido
	if (lista_suscripciones[ID_evento].num_suscritos >= rt_GE_MAX_SUSCRITOS) {
		drv_monitor_marcar(overflow_monitor_pin);
		while(1) {}
	}

	if (ID_evento == ev_PULSAR_BOTON &&
lista_suscripciones[ID_evento].num_suscritos == 1) {
		// Por nuestro diseño, nos interesa que el evento ev_PULSAR_BOTON
		// vaya primero a drv_botones_tratar y luego a rt_GE_tratar
		lista_suscripciones[ID_evento].num_suscritos++;
		lista_suscripciones[ID_evento].f_callbacks[1] =
lista_suscripciones[ID_evento].f_callbacks[0];
		lista_suscripciones[ID_evento].f_callbacks[0] = f_callback;
	}
	else {
		lista_suscripciones[ID_evento].ID_evento = ID_evento;
		lista_suscripciones[ID_evento].num_suscritos++;
		uint8_t id_f_callback = lista_suscripciones[ID_evento].num_suscritos - 1;
		lista_suscripciones[ID_evento].f_callbacks[id_f_callback] = f_callback;
```

```
        }
}

/*
 * Función para cancelar la suscripción de una función de callback a un evento en concreto
 * Si la función está suscrita, se elimina de la lista.
 */
void svc_GE_cancelar(uint32_t ID_evento, void(*f_callback)()) {
        for (uint32_t i = 0; i < lista_suscripciones[ID_evento].num_suscritos; i++) {
                if (lista_suscripciones[ID_evento].f_callbacks[i] == f_callback) {
                        // Si encuentra la función suscrita, la elimina desplazando todos los
siguientes una posición hacia atrás
    for (uint32_t j = i; j < lista_suscripciones[ID_evento].num_suscritos - 1; j++) {
                                lista_suscripciones[ID_evento].f_callbacks[j] =
lista_suscripciones[ID_evento].f_callbacks[j + 1];
    }
    // Establece el último elemento del array de callbacks a NULL después de compactarlo
    lista_suscripciones[ID_evento].f_callbacks[lista_suscripciones[ID_evento].num_suscritos
- 1] = NULL;
    // Decrementa el contador de suscritos
    lista_suscripciones[ID_evento].num_suscritos--;
                        break;
   }
  }
}
```

svc_GE.h

```
/* ********************************************************************************
 * P.H.2024: Módulo del gestor de eventos que gestiona las suscripciones
 */

#ifndef SVC_GE
#define SVC_GE

#include <stdint.h>

void svc_GE_suscribir(uint32_t ID_evento, void(*f_callback)());

void svc_GE_cancelar(uint32_t ID_evento, void(*f_callback)());

#endif
```

svc_alarmas.h

```
/* ********************************************************************************
 * P.H.2024: Driver/Manejador de las alarmas
```

```
 */

#ifndef SVC_ALARMAS
#define SVC_ALARMAS

#include <stdint.h>
#include "rt_evento_t.h"


void svc_alarma_iniciar(uint32_t monitor_overflow, void(*f_callback)(), EVENTO_T
ID_evento, uint32_t ms);

uint32_t svc_alarma_codificar(uint8_t periodico, uint32_t retardo);

void svc_alarma_activar(uint32_t retardo_ms, EVENTO_T ID_evento, uint32_t auxData);

void svc_alarma_tratar(EVENTO_T ID_evento, uint32_t auxData);


#endif
```

svc_alarmas.c

```
/* ************************************************************************
 * P.H.2024: Manejador de las alarmas
 *
 * proporciona funciones para iniciar, activar y tratar
 */

#include <stddef.h>  // Para definir NULL
#include <stdbool.h>

#include "svc_alarmas.h"
#include "drv_tiempo.h"
#include "drv_monitor.h"

// Definimos el número máximo de alarmas que se pueden programar
#define svc_ALARMAS_MAX 4


// Estructura que representa una alarma
typedef struct {
        bool activada;
        bool esPeriodica;
        uint32_t retardo_ms;
        EVENTO_T ID_evento;
        uint32_t auxData;
```

```
} ALARMA_T;

// Variables globales para la gestión de alarmas
static volatile uint8_t num_alarmas_programadas = 0;        // Número de alarmas
programadas actualmente
static ALARMA_T alarmas_programadas[EVENT_TYPES];                    // Vector con las
alarmas programadas
static uint32_t overflow_monitor_pin;                                // Pin del monitor
para overflow

static void(*ptr_svc_alarmas)();  // Puntero a la función de callback para la interrupción

/*
 * función para inicializar una alarma periódica o no
 */
void svc_alarma_iniciar(uint32_t monitor_overflow, void(*f_callback)(), EVENTO_T
ID_evento, uint32_t ms) {
        ptr_svc_alarmas = f_callback;

        // pin con el que se detectará overflow, se inicializa en estado bajo
        overflow_monitor_pin = monitor_overflow;

  // Activar la alarma
        svc_alarma_activar(ms, ID_evento, 0);
}

/*
 * Codifica el tipo de alarma (periódica o esporádica) junto con el retardo
 */
uint32_t svc_alarma_codificar(uint8_t periodico, uint32_t retardo) {
        periodico = periodico & 0x01;
        uint32_t resultado = (periodico << 31) | retardo;
        return resultado;
}

/*
 * Activa o desactiva una alarma según el retardo especificado
 */
void svc_alarma_activar(uint32_t retardo_ms, EVENTO_T ID_evento, uint32_t auxData) {
        // Si retardo_ms es 0, desprogramamos la alarma
        if (retardo_ms == 0) {
                num_alarmas_programadas--;

                // Reinicializamos los valores de la alarma
                alarmas_programadas[ID_evento].activada = false;
                alarmas_programadas[ID_evento].esPeriodica = false;
                alarmas_programadas[ID_evento].retardo_ms = 0;
                alarmas_programadas[ID_evento].ID_evento = ev_VOID;
```

```
                alarmas_programadas[ID_evento].auxData = 0;

                // Aquí es donde realmente se desactiva la alarma
                if (ID_evento == ev_INACTIVIDAD) {
                        drv_tiempo_programar_alarma(0, 0, false, svc_alarma_tratar);
                }
                else {
                        drv_tiempo_programar_alarma(1, 0, false, svc_alarma_tratar);
                }
        }
        // Si no
        else {
   // Si ya hay programadas el máximo de alarmas, activamos el monitor y entramos en un
bucle infinito
                if (alarmas_programadas[ID_evento].activada == false) {
                        if (num_alarmas_programadas == svc_ALARMAS_MAX) {
                                drv_monitor_marcar(overflow_monitor_pin);
                                while(1) {}
                        }

                        num_alarmas_programadas++;
                }

                uint8_t esPeriodica = (retardo_ms >> 31);
                retardo_ms &= 0x7FFFFFFF;

                // Actualizar la estructura con los datos de la nueva alarma
                alarmas_programadas[ID_evento].activada = true;
                alarmas_programadas[ID_evento].esPeriodica = esPeriodica;
                alarmas_programadas[ID_evento].retardo_ms = retardo_ms;
                alarmas_programadas[ID_evento].ID_evento = ID_evento;

                alarmas_programadas[ID_evento].auxData = auxData;

                // Aquí activamos la alarma realmente
                if (ID_evento == ev_INACTIVIDAD) {
                        drv_tiempo_programar_alarma(0, retardo_ms, esPeriodica,
svc_alarma_tratar);
                }
                else {
                        drv_tiempo_programar_alarma(1, retardo_ms, esPeriodica,
svc_alarma_tratar);
                }
        }
}
```

```
/*
 * Maneja las alarmas cuando se disparan los eventos
 */
void svc_alarma_tratar(EVENTO_T timer, uint32_t auxData) {
	// ha llegado la alarma de inactividad
	if (timer == 0) {
		alarmas_programadas[ev_INACTIVIDAD].activada = false;
		num_alarmas_programadas--;

		(*ptr_svc_alarmas)(ev_INACTIVIDAD,
alarmas_programadas[ev_INACTIVIDAD].auxData); // rt_FIFO_encolar
	}
	// ha llegado una alarma que no es inactividad
	else if (timer == 1) {
		uint8_t encontrado = 0;
		uint8_t ind = 0;
		while (!encontrado && ind < EVENT_TYPES) {
			if (alarmas_programadas[ind].activada) {
				encontrado = true;
			}
			else {
				ind++;
				if (ind == ev_INACTIVIDAD) {
					ind++;
				}
			}
		}

		if (alarmas_programadas[ind].esPeriodica == false) {
			alarmas_programadas[ind].activada = false;
			num_alarmas_programadas--;
		}

		(*ptr_svc_alarmas)(alarmas_programadas[ind].ID_evento,
alarmas_programadas[ind].auxData);
	}
}


/*
 * Marca el monitor en caso de overflow (se alcanza número máximo de alarmas permitidas)
 */
void svc_alarma_marcar_overflow(void) {
	drv_monitor_marcar(overflow_monitor_pin);
}
```

simon.h

```
#ifndef SIMON_H
#define SIMON_H

#include <stdint.h>

#include "drv_leds.h"
#include "drv_botones.h"
#include "drv_tiempo.h"

#include "rt_evento_t.h"
#include "rt_GE.h"
#include "rt_fifo.h"
#include "svc_GE.h"
#include "svc_alarmas.h"

#include "hal_gpio.h"
#include "hal_WDG.h"

// Constantes del juego del simón
#define VELOCIDAD_SECUENCIA_INICIO      700

// Declaración de funciones
void iniciar_simon(void);
void maq_estados_simon(uint32_t ID_evento, uint32_t auxData);

#endif
```

simon.c

```
/*
 * P.H.2024: Juego del simón
 *
 * El juego mostrará una secuencia de luces (irá incrementando su número
 * y disminuyendo la velocidad de aparición de cada luz de la secuencia)
 *
 * Al inicio de la partida, se dan a elegir 3 modos de juego:
 *          - Modo normal (se incrementan las luces de la secuencia de una en una y la
velocidad se decrementa más lentamente)
 *  - Modo rápido (se incrementan las luces de la secuencia de una en una y la velocidad se
decrementa más rápidamente)
 *  - Modo doble (se incrementan las luces de la secuencia de dos en dos y la velocidad se
decrementa más lentamente)
 */

#include "simon.h"
#include "board.h"
#include "hal_WDG.h"
```

```
#include "hal_ext_int.h"

#include <stdlib.h> // Para rand() y srand()

#if defined(modo_depuracion_lpc) || defined(modo_depuracion_nrf)
	#include "rt_estadisticas.h"
#endif

#define VELOCIDAD_NORMAL		800
#define DECREMENTO_VELOCIDAD_NORMAL	50
#define VELOCIDAD_RAPIDO		600
#define DECREMENTO_VELOCIDAD_RAPIDO	100
#define MIN_VELOCIDAD			200
#define VELOCIDAD_SECUENCIA_TERMINAR	150
#define TIEMPO_DOBLE_PULSACION		3000

typedef enum {
   normal,
   rapido,
   doble,
} MODOS_JUEGO;

typedef enum {
   s_elegir_modo_juego,
   s_secuencia_inicio,
   s_secuencia,
   s_esperar_pulsar,
   s_secuencia_terminar,
   s_esperar_soltar_para_reseteo,
} ESTADO_SIMON;


// Variables utilizadas en el juego del simón
static uint8_t secuencia_leds[64];			// Vector con secuencia aleatoria de los
leds
static uint32_t ind_secuencia = 0;
static uint32_t posicion_secuencia; // Posición actual del vector secuencia
static uint8_t incremento_secuencia;

static uint32_t cont_secuencia_inicio_fin = 0;

static uint32_t frecuencia_encendido;		// Frecuencia con la que los leds están
encendidos en la secuencia
static uint32_t decremento_velocidad;
static uint32_t estado_led;
			// 0: apagada, 1: encendida

static uint8_t num_botones;
```

```
static uint8_t num_botones_pulsados = 0;

static uint32_t contador_semilla = 4;
static ESTADO_SIMON estado_juego = s_secuencia_inicio;
static MODOS_JUEGO modo_juego;


// Implementación de las funciones del simón
void iniciar_simon(void) {
	num_botones = drv_botones_iniciar(rt_FIFO_encolar);
	hal_ext_int_deshabilitar_botones_irqs();

	estado_juego = s_elegir_modo_juego;
	drv_led_encender(1);
	drv_led_encender(3);

	rt_GE_lanzador();
}


void empezar_partida(void){
	for (uint8_t j=1; j<=4; j++){
		drv_led_conmutar(j);
	}
}

void terminar_partida(void){
	if (cont_secuencia_inicio_fin%5 != 0) {
		drv_led_conmutar(cont_secuencia_inicio_fin%5);
	}
}

void secuencia_modo_juego(void) {
	for (uint8_t i=1; i<=4; i++) {
		drv_led_conmutar(i);
	}
}


/*
 * Genera una secuencia aleatoria de LEDS según el número de botones disponibles
 */
void iniciar_secuencia_aleatoria(void) {
	contador_semilla++;
	srand(contador_semilla);
	// Generar 64 valores aleatorios en el rango [1, num_botones]
	for (uint8_t i = 0; i < 64; i++) {
		secuencia_leds[i] = (rand() % num_botones) + 1;
```

```
	}
}


void maq_estados_simon(uint32_t ID_evento, uint32_t auxData) {
	// 1. Estado s_elegir_modo_juego: muestra la secuencia de leds de elegir el modo de juego,
	// enciende las 3 luces de cada modo de juego y espera a que el jugador seleccione una de ellas
	if (estado_juego == s_elegir_modo_juego) {
		if (ID_evento == ev_T_PERIODICO) {
			if (cont_secuencia_inicio_fin == 0 || cont_secuencia_inicio_fin == 1 || cont_secuencia_inicio_fin == 2) {
				secuencia_modo_juego();
			}
			else if (cont_secuencia_inicio_fin == 3) {
				for (uint8_t i=1; i<=4; i++) {
					drv_led_apagar(i);
				}
			}
			else if (cont_secuencia_inicio_fin == 4) {
				for (uint8_t i=1; i<=3; i++) {
					drv_led_encender(i);
				}
				svc_alarma_activar(0, ev_T_PERIODICO, 0);
				hal_ext_int_habilitar_botones_irqs();
			}
			cont_secuencia_inicio_fin++;
		}

		else if (ID_evento == ev_PULSAR_BOTON) {
			if (comprobar_boton_pulsado(1)) {
				drv_led_apagar(1);
				modo_juego = normal;
				decremento_velocidad = DECREMENTO_VELOCIDAD_NORMAL;
				frecuencia_encendido = VELOCIDAD_NORMAL;
				incremento_secuencia = 1;
				posicion_secuencia = 0;
			}
			else if (comprobar_boton_pulsado(2)) {
				drv_led_apagar(2);
				modo_juego = rapido;
				decremento_velocidad = DECREMENTO_VELOCIDAD_RAPIDO;
				frecuencia_encendido = VELOCIDAD_RAPIDO;
				incremento_secuencia = 1;
				posicion_secuencia = 0;
```

```
                }
                else if (comprobar_boton_pulsado(3)) {
                        drv_led_apagar(3);
                        modo_juego = doble;
                        decremento_velocidad =
DECREMENTO_VELOCIDAD_NORMAL;
                        frecuencia_encendido = VELOCIDAD_NORMAL;
                        incremento_secuencia = 2;
                        posicion_secuencia = 1;
                }
        }

        else if (ID_evento == ev_SOLTAR_BOTON) {
                for (uint8_t i=1; i<=3; i++) {
                        drv_led_apagar(i);
                }

                hal_ext_int_deshabilitar_botones_irqs();

                svc_alarma_activar(svc_alarma_codificar(1,
VELOCIDAD_SECUENCIA_INICIO), ev_T_PERIODICO, 0);
                cont_secuencia_inicio_fin = 0;
                estado_juego = s_secuencia_inicio;
                return;
        }
}



// 2. Estado s_secuencia_inicio: muestra la secuencia de leds de inicio de la partida
if (estado_juego == s_secuencia_inicio) {
        if (ID_evento == ev_T_PERIODICO) {
                empezar_partida();
                if (cont_secuencia_inicio_fin < 3) {
                        cont_secuencia_inicio_fin++;
                }
                else {
                        cont_secuencia_inicio_fin=1;
                        ind_secuencia = 0;
                        if (modo_juego == normal || modo_juego == rapido) {
                                posicion_secuencia = 0;
                        }
                        else if (modo_juego == doble) {
                                posicion_secuencia = 1;
                        }
                        estado_led = 0;
                        iniciar_secuencia_aleatoria();
```

```
                                svc_alarma_activar(svc_alarma_codificar(1,
frecuencia_encendido), ev_T_PERIODICO, 0);
                                estado_juego = s_secuencia;
                                return;
                        }
                }
        }

        // 3. Estado s_secuencia: muestra la secuencia de leds aleatoria a reproducir por el
usuario
        else if (estado_juego == s_secuencia) {
                if (ID_evento == ev_T_PERIODICO) {
                        if (estado_led == 0) {
                                drv_led_encender(secuencia_leds[ind_secuencia]);
                                estado_led = 1;
                        }
                        else {
                                drv_led_apagar(secuencia_leds[ind_secuencia]);
                                estado_led = 0;

                                // Si ya hemos mostrado el último elemento de la secuencia
                                if (ind_secuencia == posicion_secuencia) {
                                        ind_secuencia = 0;
                                        svc_alarma_activar(0, ev_T_PERIODICO, 0);

                                        hal_ext_int_habilitar_botones_irqs();
                                        estado_juego = s_esperar_pulsar;

                                        #if defined(modo_depuracion_lpc) ||
defined(modo_depuracion_nrf)

actualizar_estadisticas_antes_de_respuesta_usuario();
                                        #endif
                                        return;
                                }
                                // Si aún nos quedan elementos de la secuencia por mostrar
                                else {
                                        ind_secuencia++;
                                }
                        }
                }
        }

        // 4. Estado s_esperar_pulsar: trata las pulsaciones del usuario
        else if (estado_juego == s_esperar_pulsar) {
                if (ID_evento == ev_PULSAR_BOTON) {
                        num_botones_pulsados++;
                        drv_led_encender(auxData);
```

```
                        if (num_botones_pulsados == 2) {
                                uint8_t num_botones_pulsados_real =
num_botones_pulsados_realmente();
                                if (num_botones_pulsados_real == 2) {
                                        svc_alarma_activar(0, ev_BOTON_RETARDO, 0);
                                        svc_alarma_activar(TIEMPO_DOBLE_PULSACION,
ev_T_ESPORADICO, 0);
                                }
                                else {
                                        num_botones_pulsados =
num_botones_pulsados_real;
                                }
                        }
                }

                else if (ID_evento == ev_SOLTAR_BOTON) {
                        num_botones_pulsados--;

                        if (auxData == secuencia_leds[ind_secuencia]) {
                                drv_led_apagar(secuencia_leds[ind_secuencia]);

                                // Si aún no hemos llegado a la última pulsación de la
secuencia
                                if (ind_secuencia != posicion_secuencia) {
                                        ind_secuencia++;
                                }
                                // Si ya hemos llegado a la última pulsación de la secuencia
                                else {
                                        ind_secuencia = 0;
                                        posicion_secuencia += incremento_secuencia;
                                        if (frecuencia_encendido > MIN_VELOCIDAD){

frecuencia_encendido=frecuencia_encendido-decremento_velocidad;
                                        }

svc_alarma_activar(svc_alarma_codificar(1,frecuencia_encendido), ev_T_PERIODICO, 0);

                                        hal_ext_int_deshabilitar_botones_irqs();
                                        estado_juego = s_secuencia;
                                        return;
                                }
                        }
                        else {
                                drv_led_apagar(auxData);

                                hal_ext_int_deshabilitar_botones_irqs();
```

```
                        if (modo_juego == normal) {
                                frecuencia_encendido = VELOCIDAD_NORMAL;
                        } else if (modo_juego == doble) {
                                frecuencia_encendido = VELOCIDAD_NORMAL;
                        } else if (modo_juego == rapido) {
                                frecuencia_encendido = VELOCIDAD_RAPIDO;
                        }


svc_alarma_activar(svc_alarma_codificar(1,VELOCIDAD_SECUENCIA_TERMINAR),
ev_T_PERIODICO, 0);

                        estado_juego = s_secuencia_terminar;
                        return;
                }
        }

        else if (ID_evento == ev_T_ESPORADICO) {
                uint8_t num_botones_pulsados_real =
num_botones_pulsados_realmente();
                if (num_botones_pulsados_real == 2) {
                        for (uint8_t i=1; i<=num_botones; i++) {
                                drv_led_apagar(i);
                        }


svc_alarma_activar(svc_alarma_codificar(1,VELOCIDAD_SECUENCIA_TERMINAR),
ev_T_PERIODICO, 0);

                        hal_ext_int_deshabilitar_botones_irqs();

                        if (modo_juego == normal) {
                                frecuencia_encendido = VELOCIDAD_NORMAL;
                        }
                        else if (modo_juego == doble) {
                                frecuencia_encendido = VELOCIDAD_NORMAL;
                        }
                        else if (modo_juego == rapido) {
                                frecuencia_encendido = VELOCIDAD_RAPIDO;
                        }
                        estado_led = 0;

                        estado_juego = s_secuencia_terminar;
                        return;
                }
                else {
                        // han soltado los botones antes de tiempo
```

```
                                num_botones_pulsados = 0;
                                for (uint8_t i=1; i<=num_botones; i++) {
                                        drv_led_apagar(i);
                                }

                                // REINICIAR CALLBACKS
                                rt_GE_iniciar(MONITOR1, maq_estados_simon);    // monitor 1
para overflow en el gestor de eventos
                                svc_alarma_activar(svc_alarma_codificar(1,
VELOCIDAD_SECUENCIA_INICIO), ev_T_PERIODICO, 0);
                                num_botones = drv_botones_iniciar(rt_FIFO_encolar);
                                hal_ext_int_deshabilitar_botones_irqs();
                                //

                                ind_secuencia = 0;
                                posicion_secuencia += incremento_secuencia;
                                estado_led = 0;

                                estado_juego = s_secuencia;
                                return;
                        }
                }
        }

        // 5. Estado s_secuencia_terminar: muestra la secuencia de leds de fin de la partida
        else if (estado_juego == s_secuencia_terminar) {
                if (ID_evento == ev_T_PERIODICO) {
                        terminar_partida();

                        if (cont_secuencia_inicio_fin == 19) {
                                num_botones_pulsados = 0;
                                cont_secuencia_inicio_fin = 0;
                                if (modo_juego == normal) {
                                        posicion_secuencia = 0;
                                }
                                else if (modo_juego == rapido) {
                                        posicion_secuencia = 0;
                                }
                                else if (modo_juego == doble) {
                                        posicion_secuencia = 1;
                                }

                                // REINICIAR CALLBACKS
                                rt_GE_iniciar(MONITOR1, maq_estados_simon);    // monitor 1
para overflow en el gestor de eventos
                                svc_alarma_activar(svc_alarma_codificar(1,
VELOCIDAD_SECUENCIA_INICIO), ev_T_PERIODICO, 0);
                                num_botones = drv_botones_iniciar(rt_FIFO_encolar);
```

```
                                hal_ext_int_deshabilitar_botones_irqs();
                                //

                                #if defined(modo_depuracion_lpc) ||
defined(modo_depuracion_nrf)
                                        uint32_t num_ev_VOID =
rt_FIFO_estadisticas(ev_VOID);
                                        uint32_t num_ev_T_PERIODICO =
rt_FIFO_estadisticas(ev_T_PERIODICO);
                                        uint32_t num_ev_PULSAR_BOTON =
rt_FIFO_estadisticas(ev_PULSAR_BOTON);
                                        uint32_t num_ev_INACTIVIDAD =
rt_FIFO_estadisticas(ev_INACTIVIDAD);
                                        uint32_t num_ev_BOTON_RETARDO =
rt_FIFO_estadisticas(ev_BOTON_RETARDO);
                                        uint32_t num_ev_SOLTAR_BOTON =
rt_FIFO_estadisticas(ev_SOLTAR_BOTON);
                                        uint32_t num_ev_T_ESPORADICO =
rt_FIFO_estadisticas(ev_T_ESPORADICO);

                                        double estadistica_tiempo_medio_cola =
get_tiempo_medio_cola();
                                        uint32_t estadistica_tiempo_max_cola =
get_tiempo_max_cola();

                                        double estadistica_tiempo_medio_respuesta_usuario =
get_tiempo_medio_respuesta_usuario();
                                        uint32_t estadistica_tiempo_max_respuesta_usuario =
get_tiempo_max_respuesta_usuario();

                                        double estadistica_tiempo_medio_atender_irq =
get_tiempo_medio_tratar_irq();
                                        uint32_t estadistica_tiempo_max_atender_irq =
get_tiempo_max_tratar_irq();
                                #endif

                                        estado_juego = s_secuencia_inicio;
                                        return;
                                }
                                else {
                                        cont_secuencia_inicio_fin++;
                                }
                        }
                }
        }
}
```

test_rt_fifo.h

```
#ifndef TEST_RT_FIFO
#define TEST_RT_FIFO

#include <stdint.h>

void run_tests_fifo(uint32_t pin_monitor_overflow);

#endif
```

test_rt_fifo.c

```
#include "test_rt_fifo.h"
#include "rt_fifo.h"

void run_tests_fifo(uint32_t pin_monitor_overflow) {
	uint32_t pin = pin_monitor_overflow;
	rt_FIFO_inicializar(pin);

	uint32_t auxData = 100; // valor auxiliar para cada evento

	// Test básico de encolar eventos hasta antes de overflow (FIFO_TAM-1 eventos)
	for (int i = 0; i < FIFO_TAM-1; i++) {
		rt_FIFO_encolar(ev_T_PERIODICO, auxData + i);
	}

	// Revisa las estadísticas
	uint32_t num_VOID = rt_FIFO_estadisticas(ev_VOID);
	uint32_t num_T_PERIODICO = rt_FIFO_estadisticas(ev_T_PERIODICO);
	uint32_t num_PULSAR_BOTON = rt_FIFO_estadisticas(ev_PULSAR_BOTON);

	// Test básico de overflow (FIFO_TAM eventos en la cola)
	// => marcará monitor 4
	rt_FIFO_encolar(ev_T_PERIODICO, auxData + FIFO_TAM-1);
}
```

board_lpc.h

```
/* ************************************************************************
 * P.H.2024: definicion de la placa simulada del LPC2105 en Keil
 */

#ifndef BOARD_LPC
#define BOARD_LPC

#include <LPC210x.H>                    /* LPC210x definitions */
```

```
#include "reserva_gpio_lpc2105.h"

// LEDs definitions for LPC2105 simulado

#define LEDS_NUMBER    4

#define LED_1          (LED1_GPIO)
#define LED_2          (LED2_GPIO)
#define LED_3          (LED3_GPIO)
#define LED_4          (LED4_GPIO)

#define LEDS_ACTIVE_STATE 1

#define LEDS_LIST { LED_1, LED_2, LED_3, LED_4 }

//BOTONES
#define BUTTONS_NUMBER 3

#define BUTTON_1       (INT_EXT1)
#define BUTTON_2       (INT_EXT2)
#define BUTTON_3       (INT_EXT3)

#define BUTTONS_LIST { BUTTON_1, BUTTON_2, BUTTON_3 }

#define BUTTONS_ACTIVE_STATE 0


//MONITOR
#define MONITOR_NUMBER 4

#define MONITOR_LIST { MONITOR1,MONITOR2,MONITOR3,MONITOR4 }

#define MONITOR1       (MONITOR1_GPIO)
#define MONITOR2       (MONITOR2_GPIO)
#define MONITOR3       (MONITOR3_GPIO)
#define MONITOR4       (MONITOR4_GPIO)

#endif
```

hal_gpio_lpc.c

```
/* ************************************************************************
 * P.H.2024: GPIOs en LPC2105
 * implementacion para cumplir el hal_gpio.h
 * interrupciones externas para los botones lo dejamos para otro modulo aparte
 */

#include "hal_gpio.h"
```

```
#include <LPC210x.H>                    /* LPC210x definitions */

#define IODIR0                          (*((volatile unsigned char *) 0xE0028008))

/**
 * Permite emplear el GPIO y debe ser invocada antes
 * de poder llamar al resto de funciones de la biblioteca.
 * re-configura todos los pines como de entrada (para evitar cortocircuitos)
 */
void hal_gpio_iniciar(void){
  // Reiniciamos los pines todos como salida (igual al reset):
  IODIR = 0x0; // GPIO Port Direction control register.
                                        // Controla la dirección de cada puerto pin
}



/**
 * El gpio se configuran como entrada o salida según la dirección.
 */
void hal_gpio_sentido(HAL_GPIO_PIN_T gpio, hal_gpio_pin_dir_t direccion){
        uint32_t masc = (1UL << gpio);
        if (direccion == HAL_GPIO_PIN_DIR_INPUT){
                        IODIR = IODIR & ~masc;
        }
        else if (direccion == HAL_GPIO_PIN_DIR_OUTPUT){
                        IODIR = IODIR | masc;
        }
}

/**
 * La función devuelve del led indicado un entero de si está encendido (1) o apagado (0)
 */
uint32_t hal_gpio_leer(HAL_GPIO_PIN_T gpio){
        uint32_t masc = (1UL << gpio);        // máscara de la posición 1 del gpio
        return ((IOPIN & masc)!=0);                     // aplicamos la máscara para ver si está
encendido o apagado
}


/**
 * Escribe en el gpio el valor
 */
void hal_gpio_escribir(HAL_GPIO_PIN_T gpio, uint32_t valor){
        uint32_t masc = (1UL << gpio);                                  // máscara de la
posición 1 del gpio
```

```
	if ((valor & 0x01) == 0) IOCLR = masc;		// verifica el valor a escribir en el pin, si el valor es 0, escribirá un 0
	else IOSET = masc;
		// sino escribirá un 1
}


/*
 * Los bits indicados se configuran como
 * entrada o salida según la dirección.
 */
void hal_gpio_sentido_n(HAL_GPIO_PIN_T gpio_inicial, uint8_t num_bits, hal_gpio_pin_dir_t direccion){
	// máscara para seleccionar los pines deseados (desde el bit gpio_inicial hasta num_bits posiciones)
	uint32_t masc = ((1 << num_bits) - 1) << gpio_inicial;

	// Si es de entrada
	if (direccion == HAL_GPIO_PIN_DIR_INPUT){
			IODIR = IODIR & ~masc;	// Pone los bits seleccionados de modo entrada
	}
	// Si es de salida
	else if (direccion == HAL_GPIO_PIN_DIR_OUTPUT){
			IODIR = IODIR | masc;			// Pone los bits seleccionados de modo salida
	}
}

/**
 * La función devuelve un entero con el valor de los bits indicados.
 * Ejemplo:
 *		- valor de los pines: 0x0F0FAFF0
 *		- bit_inicial: 12 num_bits: 4
 *		- valor que retorna la función: 10 (lee los 4 bits 12-15)
 */
uint32_t hal_gpio_leer_n(HAL_GPIO_PIN_T gpio_inicial, uint8_t num_bits){
	// máscara para seleccionar los pines deseados (desde el bit gpio_inicial hasta num_bits posiciones)
	uint32_t masc = ((1 << num_bits) - 1) << gpio_inicial;

	return (IOPIN & masc) >> gpio_inicial;
	// IOPIN : GPIO Port Pin value register. Contiene el estado de los
	  // puertos pines configurados independientemente de la direccion.
}

/**
 * Escribe en los bits indicados el valor
```

```
 * (si valor no puede representarse en los bits indicados,
 *  se escribirá los num_bits menos significativos a partir del inicial).
 */
void hal_gpio_escribir_n(HAL_GPIO_PIN_T bit_inicial, uint8_t num_bits, uint32_t valor){
	// obtiene los pins necesarios del valor y los desplaza a la posición inicial
	uint32_t masc_value = (valor & ((1 << num_bits) - 1)) << bit_inicial;

	// máscara para seleccionar los pines deseados (desde el bit gpio_inicial hasta
num_bits posiciones)
	uint32_t masc = ((1 << num_bits) - 1) << bit_inicial;

	// Limpia los bits correspondientes en el registro "IOPIN" antes de escribir
	uint32_t temp = IOPIN & ~masc;

	// Actualiza "IOPIN" escribiendo los nuevos valores en los bits seleccionados
	IOPIN = temp | masc_value;
	// limpia la mascara en el iopin y cambia sus bits de golpe
}


/*
 * No hace nada
 */
void hal_gpio_iniciar_monitor(void){
}

/*
 * Inicializa a estado bajo el monitor con identificador id
 */
void hal_gpio_iniciar_monitor_unico(uint32_t id){
  IOCLR = (1 << id);  // Asegura que el pin está en bajo inicialmente
}

/*
 * Pone a estado bajo el monitor con identificador id
 */
void hal_gpio_monitor_marcar(uint32_t id){
	IOSET = (1UL << id); // Activa el monitor (marca el pin en alto)
}

/*
 * Pone a estado alto el monitor con identificador id
 */
void hal_gpio_monitor_desmarcar(uint32_t id){
	IOCLR = (1UL << id); // Desactiva el monitor (marca el pin en bajo)
}

/*
```

```
 * Inicializa los botones del lpc (son 3) como entradas
 */
void hal_gpio_botones_iniciar(void) {
	// ACTIVAR PIN P0.16 (BUTTON_3)
	PINSEL1 |= 0x00000001; // P0.16 (sets bits 1:0)

	// ACTIVAR PIN P0.15 (BUTTON_2)
	PINSEL0 |= 0x80000000; // P0.15 (sets bits 31:30)

	// ACTIVAR PIN P0.14 (BUTTON_1)
	PINSEL0 |= 0x20000000; // P0.14 (sets bits 29:28)
}
```

hal_tiempo_lpc.c

```
/* ************************************************************************
 * P.H.2024: Temporizadores en LPC2105, Timer 0 y Timer 1
 * implementacion para cumplir el hal_tiempo.h
 * Timer0 cumple maxima frecuencia, minimas interrupciones contando en ticks
 * Timer1 avisa cada periodo de activacion en ticks
 */

#include "hal_tiempo.h"

#include <LPC210x.H> /* LPC210x definitions */

#include <stdint.h>
#include <stdbool.h>

#define MAX_COUNTER_VALUE 0xFFFFFFFE                    // Maximo valor del
contador de 32 bits
#define HAL_TICKS2US                    15
	// funcionamos PCLK a 15 MHz de un total de 60 MHz CPU Clock
#define US2MS                                                   1000
                        //milisegundos por microsogundos



static volatile uint32_t timer0_int_count = 0;  // contador de 32 bits de veces que ha saltado
la RSI Timer0

/*
 * Timer 0 Interrupt Service Routine
 * Incrementa el contador de interrupciones y limpia las interrupciones.
 */
void timer0_ISR (void) __irq {
   timer0_int_count++;
```

```
    T0IR = 1;              // Limpia la bandera de interrupción
    VICVectAddr = 0;       // indica que la interrupción actual ya ha sido atendida
}

/*
 * Programa un contador de tick sobre Timer0, con maxima precisión y minimas
interrupciones
 */
uint32_t hal_tiempo_iniciar_tick() {
        timer0_int_count = 0;                                   // Resetea el contador de
interrupciones a 0.
        T0MR0 = MAX_COUNTER_VALUE;          // Interrupción cuando TC alcanza el
valor máximo
  T0MCR = 3;                                // Configuración binaria: (0x11) (interrupción y
reinicio).

        // Configura la dirección del vector de interrupción para el Timer 0.
        VICVectAddr0 = (unsigned long)timer0_ISR;    // interrupción tratada por timer0_ISR

        // 0x20 --> bit 5 : habilita interrupcciones vectorizadas
        // 4 es el número de la interrupción que corresponde al timer0
        VICVectCntl0 = 0x20 | 4;
  VICIntEnable = VICIntEnable | 0x00000010;   // Habilita interrupción 4 (operaciones de
arriba) del timer0

        T0TCR = 2; // Reincia los contadores
  T0TCR = 1;  // Empieza la cuenta del timer0

        return HAL_TICKS2US;                //devuelve el factor conversion de ticks a
microsegundos de este hardware
}


/**
 * Lee el tiempo que lleva contando el contador y lo devuelve en ticks.
 */
uint64_t hal_tiempo_actual_tick() {
        uint64_t time;
  time = ((MAX_COUNTER_VALUE+1)*timer0_int_count) + (uint64_t)T0TC;
        return time;
}


/* ****************************************************************************
 */

static void(*f_callbacks[2])();  // Vector de funciones de callback
```

```
void timer0__ISR (void) __irq {
	if (T0IR & 0x01) {
   T0IR = 0x01;  // Limpia la bandera de interrupción
	}
	f_callbacks[0]();
	T0IR = 1;                                                    // Limpia la bandera de
interrupción
	VICVectAddr = 0;                // indica que la interrupción actual ya ha sido atendida
}

void timer1__ISR (void) __irq {
	if (T1IR & 0x01) {
   T1IR = 0x01;  // Limpia la bandera de interrupción
	}
	f_callbacks[1]();
	T1IR = 1;                                                    // Limpia la bandera de
interrupción
	VICVectAddr = 0;                // indica que la interrupción actual ya ha sido atendida
}


/*
 * Programa el reloj para que llame a la función de callback cada periodo.
 * El periodo se indica en tick. Si el periodo es cero se para el temporizador.
 */
void hal_tiempo_reloj_periodico_tick(uint32_t periodo_en_tick, void(*funcion_callback)(),
uint32_t id){

	f_callbacks[1] = funcion_callback;

	if (periodo_en_tick != 0) { //Si el periodo es cero solo se para el temporizador.
		T1MR0 = periodo_en_tick  - 1;                    // 15 Ticks (ciclos) por
microsegundo.

                                                        // (periodo_en_ms * HAL_TICKS2US *
US2MS) - 1;

                                                        // resto uno por como incrementa y
compara
		// ALARMA PERIÓDICA
		T1MCR = 3;

		// Habilita la interrupción del Timer1
		VICVectAddr1 = (unsigned long)timer1__ISR;
		// 0x20 --> bit 5 : habilita interrupcciones vectorizadas
		// 5 es el número de la interrupción que corresponde al timer1
		VICVectCntl1 = 0x20 | 5;
```

```
                VICIntEnable = VICIntEnable | 0x00000020;

                T1TCR = 3;  // Reincia los contadores
                T1TCR = 1;  // Empieza la cuenta
        } else {
                // Detiene el temporizador
                T1TCR = 0;
   VICIntEnClr = 0x20;   // Deshabilita la interrupción del Timer 1
        }
}




// ************************************************************************
// ********************** NUEVO PARA EL BIS ***************************
// ************************************************************************

static void(*f_callbacks_lpc_nuevas[2])();

void timer0___ISR (void) __irq {
        if (T0IR & 0x01) {
   T0IR = 0x01;  // Limpia la bandera de interrupción
        }
        f_callbacks_lpc_nuevas[0](0,0);
        T0IR = 1;                                       // Limpia la bandera de
interrupción
        VICVectAddr = 0;                // indica que la interrupción actual ya ha sido atendida
}

void timer1___ISR (void) __irq {
        if (T1IR & 0x01) {
   T1IR = 0x01;  // Limpia la bandera de interrupción
        }
        f_callbacks_lpc_nuevas[1](1,0);
        T1IR = 1;                                       // Limpia la bandera de
interrupción
        VICVectAddr = 0;                // indica que la interrupción actual ya ha sido atendida
}


/*
 * Programa el temporizador indicado en 'timer'
 */
void hal_tiempo_reloj_tick(uint8_t timer, uint32_t periodo_en_tick, bool esPeriodica,
void(*funcion_callback)()) {
```

```
	// Activamos la alarma
	if (periodo_en_tick != 0) {
		if (timer == 0) {
			f_callbacks_lpc_nuevas[0] = funcion_callback;		// guardamos la
función de callback a la que debe llamar

													// cuando llegue la
interrupción
			// Limpiar interrupciones del temporizador (si están activas)
			T0IR = 0xFF;  // Limpiar todas las interrupciones pendientes del Timer
0
			T0MR0 = periodo_en_tick  - 1;
			// Configura el modo de interrupción:
    // - Si es periódica: genera interrupción y reinicia al alcanzar MR0.
    // - Si no es periódica: solo genera una interrupción sin reiniciar.
			if (esPeriodica) {
				T0MCR = 3;
			} else {
				T0MCR = 1;
			}
			VICVectAddr0 = (unsigned long)timer0___ISR;
			VICVectCntl0 = 0x20 | 4;
			VICIntEnable |= 0x00000010;
			T0TCR = 3;
			T0TCR = 1;
		}
		else if (timer == 1) {
			f_callbacks_lpc_nuevas[1] = funcion_callback;		// guardamos la
función de callback a la que debe llamar

													// cuando llegue la
interrupción
			// Limpiar interrupciones del temporizador (si están activas)
			//T1TCR = 0;  // Detener el temporizador
			T1IR = 0xFF;  // Limpiar todas las interrupciones pendientes
			//T1MCR = 0;  // Limpiar cualquier configuración previa del registro
MCR
			T1MR0 = periodo_en_tick  - 1;

			// Configura el modo de interrupción:
    // - Si es periódica: genera interrupción y reinicia al alcanzar MR1.
    // - Si no es periódica: solo genera una interrupción sin reiniciar.
			if (esPeriodica) {
				T1MCR = 3;
			} else {
				T1MCR = 1;
			}
			VICVectAddr1 = (unsigned long)timer1___ISR;
```

```
                        VICVectCntl1 = 0x20 | 5;
                        VICIntEnable |= 0x00000020;
                        T1TCR = 3;
                        T1TCR = 1;
                }
                // si quieren programar un timer fuera del rango establecido
                else {
                        while(1) {
                                // error: timer fuera del rango establecido
                        }
                }
        }

        // Desactivamos la alarma
        else {
                if (timer == 0) {
                        T0TCR = 0;     // Detiene el Timer0
                        VICIntEnClr = 0x10;                     // Deshabilita la
interrupción del Timer0
                }
                else if (timer == 1) {
                        T1TCR = 0;     // Detiene el Timer1
                        VICIntEnClr = 0x20;                     // Deshabilita la
interrupción del Timer1
                }
                // si quieren programar un timer fuera del rango establecido
                else {
                        while(1) {
                                // error: timer fuera del rango establecido
                        }
                }
        }
}
```

reserva_gpio_lpc2105.h

```
/* ************************************************************************
 * P.H.2024: Configuracion y reserva de pines y gpios en el simulador del LPC2105 en Keil
 */

#ifndef RESERVA_GPIO
#define RESERVA_GPIO

//gpios - pines

//LEDS
enum {          LED1_GPIO = 0,
```

```
                                LED2_GPIO = 1,
                                LED3_GPIO = 2,
                                LED4_GPIO = 3,
};

//BOTONES
enum {  INT_EXT1 = 14, //eINT1
                                INT_EXT2 = 15, //eINT2
                                INT_EXT3 = 16, //eINT0
};
// MONITORES
enum {          MONITOR1_GPIO = 28,
                                MONITOR2_GPIO = 29,
                                MONITOR3_GPIO = 30,
                                MONITOR4_GPIO = 31,
};
#endif
```

hal_consumo_lpc.c

```
/*
 * P.H.2024: Módulo para el consumo en el LPC2105
 */

#include "hal_consumo.h"

#include <LPC210x.H>

//definida en Startup.s
extern void switch_to_PLL(void);
extern void Reset_Handler(void);

/*
 * No hace nada
 */
void hal_consumo_iniciar(void) {
}

/*
 * Pone al procesador en estado de espera para reducir su consumo
 */
void hal_consumo_esperar(void)  {
  EXTWAKE = 7;                          // (0x111) --> configura los pines EXTINT0, EXTINT1 y
EXTINT2
                                                                        // para despertar
al procesador gracias a su interrupción
```

```
	PCON |= 0x01;			// Establece el bit 0 (0x1) del registro `PCON` para poner el
								// procesador en modo "espera" (Idle). Saldrá de espera con interrupción
}

/*
 * Duerme al procesador para minimizar su consumo
 */
void hal_consumo_dormir(void)  {
	EXTWAKE = 7;				// (0x7) --> los botones conectados a EXTINT0, EXTINT1 y EXTINT2
								// podrán despertar al procesador (BUTTON_1, BUTTON_2 y BUTTON_3)

	PCON |= 0x02;			// Establece el bit 1 (0x10) del registro `PCON` para poner el
								// procesador en modo "dormir" (PowerDown).

	switch_to_PLL(); //PLL aranca a 12Mhz cuando volvemos de powewr down
}
```

hal_ext_int_lpc.c

```
/*
 * P.H.2024: Módulo para gestionar las interrupciones en el LPC2105
 */

#include "hal_ext_int.h"

#include <LPC210x.H>

#define EXTMODE		(*((volatile unsigned char *) 0xE01FC148))
#define EXTPOLAR	(*((volatile unsigned char *) 0xE01FC14C))

void (*ptr_hal_ext_int_lpc)();

/*
 * Deshabilita las interrupciones de los botones
 */
void hal_ext_int_deshabilitar_botones_irqs(void) {
	VICIntEnClr = 0x0001C000;    // deshabilita EINT0, EINT1 y EINT2
}

/*
 * Habilita las interrupciones de los botones
```

```
 */
void hal_ext_int_habilitar_botones_irqs(void) {
	// Limpia los flags pendientes de las interrupciones externas
  EXTINT |= 0x00000007;  // Borra los flags de EINT0 (bit 0), EINT1 (bit 1), EINT2 (bit 2)


  VICIntEnable |= 0x00004000;		// habilita EINT0
	VICIntEnable |= 0x00008000;// habilita EINT1
  VICIntEnable |= 0x00010000;		// habilita EINT2
}

/*
 * Deshabilita las interrupciones del botón que le pase como argumento
 */
void hal_ext_int_deshabilitar_boton_irqs(uint8_t boton) {
	if (boton == 3) {
		VICIntEnClr = 0x00004000;  // deshabilita EINT0
	}
	else if (boton == 1) {
		VICIntEnClr = 0x00008000;  // deshabilita EINT1
	}
	else if (boton == 2) {
		VICIntEnClr = 0x00010000;  // deshabilita EINT2
	}
}

/*
 * Habilita las interrupciones del botón que le pase como argumento
 */
void hal_ext_int_habilitar_boton_irqs(uint8_t boton) {
	if (boton == 3) {
		VICIntEnable |= 0x00004000;// Habilita la interrupción de EXTINT0
	}
	else if (boton == 1) {
		VICIntEnable |= 0x00008000;// Habilita la interrupción de EXTINT1
	}
	else if (boton == 2) {
		VICIntEnable |= 0x00010000;// Habilita la interrupción de EXTINT2
	}
}


void eint0_ISR (void) __irq {
	hal_ext_int_deshabilitar_boton_irqs(3);

	EXTINT |= 1;		// borra el flag de interrupcion de EXTINT0
	VICVectAddr = 0;	// indica que la interrupción actual ya ha sido atendida
```

```
  (*ptr_hal_ext_int_lpc)(3);
}

void eint1_ISR (void) __irq {
	hal_ext_int_deshabilitar_boton_irqs(1);

	EXTINT |= 2;		// borra el flag de interrupcion de EXTINT1
	VICVectAddr = 0;	// indica que la interrupción actual ya ha sido atendida

  (*ptr_hal_ext_int_lpc)(1);
}

void eint2_ISR (void) __irq {
	hal_ext_int_deshabilitar_boton_irqs(2);

  EXTINT |= 4;		// borra el flag de interrupcion de EXTINT2
  VICVectAddr = 0;	// indica que la interrupción actual ya ha sido atendida

	(*ptr_hal_ext_int_lpc)(2);
}

/*
 * Configura las interrupciones externas (EINT0, EINT1 y EINT2) para los botones
 * y establece la función de callback que ejecutará cuando se detecten estas interrupciones
 */
void hal_ext_int_botones_iniciar(void(*funcion_callback)()) {
	ptr_hal_ext_int_lpc = funcion_callback;

	// Configura el modo de las interrupciones externas (EXTINT):
	EXTMODE	=	1;			//Configura el modo en el que se
detectan las interrupciones externas
                                                                        // 1->
sensible a bordes, así se activará cuando ocurra un cambio
                                                                        //     de
estado del pin (LoToHi o HiToLo)

	EXTPOLAR	=	0;			// Configura la polaridad del borde
                                                                        // 0-> la
interrupción se dispara de HiToLo
                                                                        // 1-> la
interrupción se dispara de LoToHi

	EXTINT |= 1;			// Limpia cualquier interrupción externa pendiente

	// ACTIVAR PIN P0.16 (BUTTON_3)
	VICVectAddr4 = (unsigned long)eint0_ISR;
	VICVectCntl4 = 0x20 | 14;	// 14 es el número de interrupción asignado (se
corresponde con el EINT0)
```

```
	// ACTIVAR PIN P0.15 (BUTTON_2)
  VICVectAddr6 = (unsigned long)eint2_ISR;
	VICVectCntl6 = 0x20 | 16;    // 16 es el número de interrupción asignado (se
corresponde con el EINT2)

	// ACTIVAR PIN P0.14 (BUTTON_1)
	VICVectAddr5 = (unsigned long)eint1_ISR;
	VICVectCntl5 = 0x20 | 15;    // 15 es el número de interrupción asignado (se
corresponde con el EINT1)

	hal_ext_int_habilitar_botones_irqs();
}


uint8_t esta_pulsado_boton_1(void) {
	EXTINT |= 2;
  return ((EXTINT & 2) != 0);
}

uint8_t esta_pulsado_boton_2(void) {
	EXTINT |= 4;
  return ((EXTINT & 4) != 0);
}

uint8_t esta_pulsado_boton_3(void) {
	EXTINT |= 1;
  return ((EXTINT & 1) != 0);
}

/*
 * devuelve el número del botón que está pulsado, si no hay ninguno, devuelve 0.
 */
uint32_t comprobar_boton_pulsado(uint8_t boton) {
	if (boton == 1) {
		return esta_pulsado_boton_1();
	}
	if (boton == 2) {
		return esta_pulsado_boton_2();
	}
	if (boton == 3) {
		return esta_pulsado_boton_3();
	}
	return 0;
}


uint8_t num_botones_pulsados_realmente(void) {
```

```
	uint8_t num_pulsados = 0;
	for (uint8_t i=1; i<=3; i++) {
		if (comprobar_boton_pulsado(i)) {
			num_pulsados++;
		}
	}
	return num_pulsados;
}
```

hal_WDG_lpc.c

```
/* ****************************************************************************
 * P.H.2024: Watchdog para el LPC2105
 */

#include "hal_WDG.h"
#include "rt_sc.h"
#include <LPC210x.H>                    /* LPC210x definitions */

/*
 * Esta función configura el Watchdog Timer para que reinicie el sistema si no
 * se "alimentado" antes de que se agote el tiempo especificado
 */
void hal_WDG_iniciar(uint32_t sec) {
	WDTC = sec * (15000000 / 4);		// Tiempo hasta que expire el temporizador del
watchdog
	WDMOD = 0x03;				// se configura que cuando expire el tiempo se reinicie
el sistema
                                                                        // -> bit 0 : habilita
watchdog timer
                                                                        // -> bit 1 : configura
reinicio al expirar
	hal_WDG_feed();		// alimenta para que empiece a contar ya
}

/*
 * Función que se llama periodicamente para "alimentar" y se verifique
 * que el sistema sigue activo
 */
void hal_WDG_feed(void) {
	entrar_SC();
	WDFEED = 0xAA; // Primer valor para alimentar el Watchdog.
	WDFEED = 0x55; // Segundo valor para alimentar el Watchdog.
	salir_SC();
}
```

hal_gpio_nrf.c

```
/* ****************************************************************************
 * P.H.2024: GPIOs en NRF52840
 * implementacion para cumplir el hal_gpio.h
 */

#include <nrf.h>
#include <board_nrf52840dk.h>

#include <stdbool.h>


/*
 * Inicializa los pines asociados a los leds para ser usados como salidas.
 */
void hal_gpio_iniciar(){
	// Configura los pin del GPIO para los LEDS como salida

	// Sigue este esquema:
	//		modo de dirección: pin como salida
	//		modo de conducción: estándar (tanto en alto como en bajo)
	//		permite leer su estado (conectado internamente para leer su estado)
	//		sin resistencia pull-up o pull-down
	//		desactiva la detección de eventos
  NRF_GPIO->PIN_CNF[LED_1] = (GPIO_PIN_CNF_DIR_Output <<
GPIO_PIN_CNF_DIR_Pos) |
                           (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                           (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                           (GPIO_PIN_CNF_PULL_Disabled << GPIO_PIN_CNF_PULL_Pos) |
                           (GPIO_PIN_CNF_SENSE_Disabled <<
GPIO_PIN_CNF_SENSE_Pos);

  NRF_GPIO->PIN_CNF[LED_2] = (GPIO_PIN_CNF_DIR_Output <<
GPIO_PIN_CNF_DIR_Pos) |
                           (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                           (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                           (GPIO_PIN_CNF_PULL_Disabled << GPIO_PIN_CNF_PULL_Pos) |
                           (GPIO_PIN_CNF_SENSE_Disabled <<
GPIO_PIN_CNF_SENSE_Pos);

  NRF_GPIO->PIN_CNF[LED_3] = (GPIO_PIN_CNF_DIR_Output <<
GPIO_PIN_CNF_DIR_Pos) |
                           (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                           (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                           (GPIO_PIN_CNF_PULL_Disabled << GPIO_PIN_CNF_PULL_Pos) |
```

```
                        (GPIO_PIN_CNF_SENSE_Disabled <<
GPIO_PIN_CNF_SENSE_Pos);


  NRF_GPIO->PIN_CNF[LED_4] = (GPIO_PIN_CNF_DIR_Output <<
GPIO_PIN_CNF_DIR_Pos) |
                        (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                        (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                        (GPIO_PIN_CNF_PULL_Disabled << GPIO_PIN_CNF_PULL_Pos) |
                        (GPIO_PIN_CNF_SENSE_Disabled << GPIO_PIN_CNF_SENSE_Pos);
}


/**
 * Los bits indicados se configuran como
 * entrada o salida según la dirección.
 */
void hal_gpio_sentido(uint32_t num_bits, uint32_t direccion){
        // Si dirección de entrada
        if (direccion == GPIO_PIN_CNF_DIR_Input) {
                // Configura el bit en cuestión como entrada
                NRF_GPIO->PIN_CNF[num_bits] = (GPIO_PIN_CNF_DIR_Input <<
GPIO_PIN_CNF_DIR_Pos);
        }
        // Si dirección de salida
        else if (direccion == GPIO_PIN_CNF_DIR_Output) {
                // Configura el bit en cuestión como salida
                NRF_GPIO->PIN_CNF[num_bits] = (GPIO_PIN_CNF_DIR_Output <<
GPIO_PIN_CNF_DIR_Pos);
        }
}


/**
 * Escribe en el gpio el valor
 */
void hal_gpio_escribir(uint32_t gpio, uint32_t valor){
        uint32_t masc = (1UL << gpio);          // máscara de la posición 1 del gpio

        if ((valor & 0x01) == 0) {                      // verifica el valor a escribir en el pin, si
el valor es 0
                NRF_GPIO->OUTCLR = masc;                        // escribirá un 0
        }
        else {
                NRF_GPIO->OUTSET = masc;                        // sino escribirá un 1
        }
}
```

```
/**
 * La función devuelve un entero (bool) con el valor de los bits indicados.
 */
uint32_t hal_gpio_leer(uint32_t gpio){
	uint32_t masc = (1UL << gpio);				// máscara de la posición 1 del gpio
	return ((NRF_GPIO->IN & masc)!=0);		// aplicamos la máscara para ver si está encendido o apagado
}


/* ****************************************************************************
 * ********************** MONITORESSS ***************************************
 * ****************************************************************************
 */

/*
 * No hace nada
 */
void hal_gpio_iniciar_monitor(){
}

/*
 * Activa el monitor que se le pasa
 */
void hal_gpio_iniciar_monitor_unico(uint32_t id){
	NRF_GPIO->OUTSET = (1UL << id);  // Activa el monitor (procesador despierto)
}

/*
 * Pone en estado alto el monitor que se le pasa
 */
void hal_gpio_monitor_marcar(uint32_t id){
	NRF_GPIO->OUTSET = (1UL << id);  // Activa el monitor (procesador despierto)
}

/*
 * Pone en estado bajo el monitor que se le pasa
 */
void hal_gpio_monitor_desmarcar(uint32_t id){
	NRF_GPIO->OUTCLR = (1UL << id);  // Desactiva el monitor (procesador wfi o idle)
}

/*
 * Configurar el botón para generar una interrupción usando GPIOTE
 */
void hal_gpio_botones_iniciar(void) {
	// Configura los pin del GPIO para los BUTTONS como salida
```

```
        // Sigue este esquema:
        //              modo de dirección: pin como entrada
        //              modo de conducción: estándar (tanto en alto como en bajo)
        //              permite leer su estado (conectado internamente para leer su estado)
        //              resistencia en pull-up para que el pin esté en estado alto cuando el
botón no está presionado
        //              Habilita la detección de un evento cuando el pin cambia a un nivel
bajo

        NRF_GPIO->PIN_CNF[BUTTON_1] = (GPIO_PIN_CNF_DIR_Input <<
GPIO_PIN_CNF_DIR_Pos) |
                        (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                        (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                        (GPIO_PIN_CNF_PULL_Pullup << GPIO_PIN_CNF_PULL_Pos) |
                        (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos);

  NRF_GPIO->PIN_CNF[BUTTON_2] = (GPIO_PIN_CNF_DIR_Input <<
GPIO_PIN_CNF_DIR_Pos) |
                        (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                        (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                        (GPIO_PIN_CNF_PULL_Pullup << GPIO_PIN_CNF_PULL_Pos) |
                        (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos);

  NRF_GPIO->PIN_CNF[BUTTON_3] = (GPIO_PIN_CNF_DIR_Input <<
GPIO_PIN_CNF_DIR_Pos) |
                        (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                        (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                        (GPIO_PIN_CNF_PULL_Pullup << GPIO_PIN_CNF_PULL_Pos) |
                        (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos);


  NRF_GPIO->PIN_CNF[BUTTON_4] = (GPIO_PIN_CNF_DIR_Input <<
GPIO_PIN_CNF_DIR_Pos) |
                        (GPIO_PIN_CNF_DRIVE_S0S1 << GPIO_PIN_CNF_DRIVE_Pos) |
                        (GPIO_PIN_CNF_INPUT_Connect << GPIO_PIN_CNF_INPUT_Pos) |
                        (GPIO_PIN_CNF_PULL_Pullup << GPIO_PIN_CNF_PULL_Pos) |
                        (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos);

}
```

hal_gpio_nrf.h

```
/* ****************************************************************************
 * P.H.2024: TODO
 */
```

```
#ifndef HAL_GPIO
#define HAL_GPIO

#include <stdint.h>
#include <stdbool.h>

void hal_gpio_iniciar();

void hal_gpio_sentido(uint32_t num_bits, uint32_t direccion);
uint32_t hal_gpio_leer(uint32_t gpio);
void hal_gpio_escribir(uint32_t gpio, uint32_t valor);

void hal_gpio_iniciar_monitor();
void hal_gpio_iniciar_monitor_unico(uint32_t id);
void hal_gpio_monitor_marcar(uint32_t id);
void hal_gpio_monitor_desmarcar(uint32_t id);

void hal_gpio_botones_iniciar(void);


#endif
```

hal_tiempo_nrf.c

```
/* ****************************************************************************
 * P.H.2024: Tiempo en NRF52840
 * implementacion para cumplir el hal_tiempo.h
 */

#include <nrf.h>
#include <stdbool.h>

#define MAX_COUNTER_VALUE 0xFFFFFFFE      // Maximo valor del contador de 32 bits
#define HAL_TICKS2US                15                                        //
funcionamos PCLK a 15 MHz de un total de 60 MHz CPU Clock


static volatile uint32_t timer0_int_count = 0;// contador de 32 bits de veces que ha saltado la
RSI Timer0

/*
 * configura e inicializa la cuenta de tiempo en ticks del hardware y
 * devuelve la constante hal_ticks2us,
 * hal_ticks2us permite pasar de los ticks del hardware a microsegundos
 * (tip, el driver lo necesitara para trabajar en us y ms de la app y hacer la conversion a ticks
del hardware)
```

```
 */
uint32_t hal_tiempo_iniciar_tick(void) {
	timer0_int_count = 0;

	NRF_TIMER0->TASKS_STOP = 1;		// Detiene el temporizador
	NRF_TIMER0->TASKS_CLEAR = 1;		// Limpia el contador
	NRF_TIMER0->CC[0] = MAX_COUNTER_VALUE;		// Establece el valor
máximo de comparación, y así cuando

								// el temporizador
alcanza ese valor, se genera un evento

								// de comparación
	NRF_TIMER0->PRESCALER = 0;		// Configura el prescaler a 1 MHz (el
temporizador cuenta

								// con 1 tick por
microsegundo).


	// Permite configurar el temporizador en modo de 32 bits (permitiendo así un rango
de conteo grande)
  NRF_TIMER0->BITMODE = TIMER_BITMODE_BITMODE_32Bit <<
TIMER_BITMODE_BITMODE_Pos;
	// Pone el timer0 en modo temporizador
	NRF_TIMER0->MODE = TIMER_MODE_MODE_Timer;
  // Atajo para limpiar automáticamente
	NRF_TIMER0->SHORTS = TIMER_SHORTS_COMPARE0_CLEAR_Enabled <<
TIMER_SHORTS_COMPARE0_CLEAR_Pos; // Limpia automáticamente
	// Habilita interrupción para el evento de comparación COMPARE0
	NRF_TIMER0->INTENSET = TIMER_INTENSET_COMPARE0_Enabled <<
TIMER_INTENSET_COMPARE0_Pos;
	// Se trata con Timer0_IRQn
	NVIC_EnableIRQ(TIMER0_IRQn);
	// Inicia el contador
	NRF_TIMER0->TASKS_START = 1;

	return HAL_TICKS2US;
}

/*
 * nos devuelve el numero total de ticks desde que se inicio la cuenta
 */
uint64_t hal_tiempo_actual_tick(void) {
	// Guarda el valor actual del contador en el registro de comparación CC[0]
	NRF_TIMER0->TASKS_CAPTURE[0] = 1;

	// Almacena en ticks_capturados el valor que está almacenado en CC[0]
```

```
        uint64_t ticks_capturados = NRF_TIMER0->CC[0];

        // Devuelve el número total de ticks
        return ((MAX_COUNTER_VALUE+1)*timer0_int_count + ticks_capturados);
}



static void(*f_callback_nrf)();

//void TIMER1_IRQHandler(void) {
//    if (NRF_TIMER1->EVENTS_COMPARE[1]) {     // Ha ocurrido evento de comparación en
CC[1], y EVENTS_COMPARE[1]
//
                                                                    // es un flag que indica si
el temporizado ha alcanzado el valor de CC[1]
//        NRF_TIMER1->EVENTS_COMPARE[1] = 0; // resetea flag
//        f_callback_nrf();
//    }
//}




/*
 * configura el Timer1 para generar una alarma periódica basada en el periodo_en_tick
especificado.
 * Cuando el temporizador alcanza el valor de comparación, dispara una interrupción que
ejecutará
 * la función de callback.
 * Si periodo_en_tick es igual a 0, detendrá el temporizador y deshabilitará la interrupción.
 */
void hal_tiempo_reloj_periodico_tick(uint32_t periodo_en_tick, void(*funcion_callback)()) {
        f_callback_nrf = funcion_callback;

    if (periodo_en_tick != 0) { // Si el periodo no es cero, se configura el temporizador
      //NRF_TIMER1->TASKS_STOP = 1;                // Detiene el temporizador
      NRF_TIMER1->TASKS_CLEAR = 1;                 // Limpia el contador
                    NRF_TIMER1->CC[0] = periodo_en_tick - 1;       // Configura el valor de
comparación en el canal 0
                    NRF_TIMER1->PRESCALER = 0;                  // Prescaler para 1 MHz (1
us por tick)

                            // Permite configurar el temporizador en modo de 32 bits (permitiendo
así un rango de conteo grande)
```

```
    NRF_TIMER1->BITMODE = TIMER_BITMODE_BITMODE_32Bit <<
TIMER_BITMODE_BITMODE_Pos;
                        // Pone el timer1 en modo temporizador
                        NRF_TIMER1->MODE = TIMER_MODE_MODE_Timer;
                        // Atajo para limpiar automáticamente el temporizador después de
alcanzar el valor de comparación
    NRF_TIMER1->SHORTS = TIMER_SHORTS_COMPARE0_CLEAR_Enabled <<
TIMER_SHORTS_COMPARE0_CLEAR_Pos;
    // Habilita la interrupción para el evento de comparación del canal 0
    NRF_TIMER1->INTENSET = TIMER_INTENSET_COMPARE0_Enabled <<
TIMER_INTENSET_COMPARE0_Pos;
                        // Se trata con Timer1_IRQn
                        NVIC_EnableIRQ(TIMER1_IRQn);
                        // Inicia el contador
    NRF_TIMER1->TASKS_START = 1;
  } else {
    // Detener el temporizador y deshabilitar interrupciones si el período es cero
    NRF_TIMER1->TASKS_STOP = 1;
                        // Deshabilita la interrupción
                        NVIC_DisableIRQ(TIMER1_IRQn);
  }
}




static void(*f_callbacks_nrf_nuevas[2])();

void TIMER0_IRQHandler(void) {
 if (NRF_TIMER0->EVENTS_COMPARE[0]) {        // Ha ocurrido evento de comparación en
CC[0], y EVENTS_COMPARE[0]

                                                        // es un flag que indica si
el temporizado ha alcanzado el valor de CC[0]
                NRF_TIMER0->EVENTS_COMPARE[0] = 0;          // resetea flag
  f_callbacks_nrf_nuevas[0](0,0);
 }
}

void TIMER1_IRQHandler(void) {
 if (NRF_TIMER1->EVENTS_COMPARE[1]) {          // Ha ocurrido evento de comparación
en CC[1], y EVENTS_COMPARE[1]

                                                        // es un flag que indica si
el temporizado ha alcanzado el valor de CC[1]
  NRF_TIMER1->EVENTS_COMPARE[1] = 0;        // resetea flag
  f_callbacks_nrf_nuevas[1](1,0);
 }
```

```
	else if (NRF_TIMER1->EVENTS_COMPARE[0]) { // Ha ocurrido evento de
comparación en CC[0], y EVENTS_COMPARE[0]

                                                                    // es un flag que
indica si el temporizado ha alcanzado el valor de CC[0]
		NRF_TIMER1->EVENTS_COMPARE[0] = 0;                          // resetea
flag
		f_callback_nrf();
	}
}

/*
 * Programa el temporizador indicado en 'timer'
 */
void hal_tiempo_reloj_tick(uint8_t timer, uint32_t periodo_en_tick, bool esPeriodica,
void(*funcion_callback)()) {

	// Activamos la alarma
	if (periodo_en_tick != 0) {
		if (timer == 0) {
			f_callbacks_nrf_nuevas[0] = funcion_callback;
			NRF_TIMER0->TASKS_CLEAR = 1;               // Limpia el contador
			NRF_TIMER0->CC[0] = periodo_en_tick - 1;   // Configura el valor
de comparación
			NRF_TIMER0->PRESCALER = 0;                 // Prescaler para 1
MHz (1 us por tick)
    NRF_TIMER0->BITMODE = TIMER_BITMODE_BITMODE_32Bit <<
TIMER_BITMODE_BITMODE_Pos;
			NRF_TIMER0->MODE = TIMER_MODE_MODE_Timer;              //
Modo de temporizador
    if (esPeriodica) {
				// Así al limpiarse solo, puede volver a contar desde 0 sin
intervenir
				NRF_TIMER0->SHORTS =
TIMER_SHORTS_COMPARE0_CLEAR_Enabled <<
TIMER_SHORTS_COMPARE0_CLEAR_Pos; // Limpia automáticamente
    } else {
				NRF_TIMER0->SHORTS = 0;
			}
			NRF_TIMER0->INTENSET =
TIMER_INTENSET_COMPARE0_Enabled << TIMER_INTENSET_COMPARE0_Pos;
    NVIC_EnableIRQ(TIMER0_IRQn);               // Habilita la interrupción en el NVIC
    NRF_TIMER0->TASKS_START = 1;               // Inicia el temporizador
		}
		else if (timer == 1) {
			f_callbacks_nrf_nuevas[1] = funcion_callback;
    NRF_TIMER1->TASKS_CLEAR = 1;               // Limpia el contador
```

```
                        NRF_TIMER1->CC[1] = periodo_en_tick - 1;       // Configura el valor
de comparación
                        NRF_TIMER1->PRESCALER = 0;                    // Prescaler para 1
MHz (1 us por tick)
    NRF_TIMER1->BITMODE = TIMER_BITMODE_BITMODE_32Bit <<
TIMER_BITMODE_BITMODE_Pos;
                        NRF_TIMER1->MODE = TIMER_MODE_MODE_Timer;
                // Modo de temporizador
                        if (esPeriodica) {
                                // Así al limpiarse solo, puede volver a contar desde 0 sin
intervenir
                                NRF_TIMER1->SHORTS =
TIMER_SHORTS_COMPARE1_CLEAR_Enabled <<
TIMER_SHORTS_COMPARE1_CLEAR_Pos; // Limpia automáticamente
    } else {
                                NRF_TIMER1->SHORTS = 0;
                        }
                        NRF_TIMER1->INTENSET =
TIMER_INTENSET_COMPARE1_Enabled << TIMER_INTENSET_COMPARE1_Pos;
    NVIC_EnableIRQ(TIMER1_IRQn);                // Habilita la interrupción en el NVIC
    NRF_TIMER1->TASKS_START = 1;                  // Inicia el temporizador
                }
                // si quieren programar un timer fuera del rango establecido
                else {
                        while(1) {
                                // error: timer fuera del rango establecido
                        }
                }
        }

        // Desactivamos la alarma
        else {
                if (timer == 0) {
                        NRF_TIMER0->TASKS_STOP = 1;
    NVIC_DisableIRQ(TIMER0_IRQn);
                }
                else if (timer == 1) {
                        NRF_TIMER1->TASKS_STOP = 1;
    NVIC_DisableIRQ(TIMER1_IRQn);
                }
                // si quieren programar un timer fuera del rango establecido
                else {
                        while(1) {
                                // error: timer fuera del rango establecido
                        }
                }
        }
}
```

board_nrf52840dk.h

```
#ifndef BOARD_NRF
#define BOARD_NRF

// BOARD_PCA10056

/** @brief Macro for mapping port and pin numbers to values understandable for nrf_gpio functions. */
#define NRF_GPIO_PIN_MAP(port, pin) (((port) << 5) | ((pin) & 0x1F))

// LEDs definitiones para nRF52480 DK, PCA10056 en nRF SDK
#define LEDS_NUMBER    4

#define LED_1          NRF_GPIO_PIN_MAP(0,13)
#define LED_2          NRF_GPIO_PIN_MAP(0,14)
#define LED_3          NRF_GPIO_PIN_MAP(0,15)
#define LED_4          NRF_GPIO_PIN_MAP(0,16)

#define LEDS_ACTIVE_STATE 0

#define LEDS_LIST { LED_1, LED_2, LED_3, LED_4 }

// Botones
#define BUTTONS_NUMBER 4

#define BUTTON_1       NRF_GPIO_PIN_MAP(0,11)
#define BUTTON_2       NRF_GPIO_PIN_MAP(0,12)
#define BUTTON_3       NRF_GPIO_PIN_MAP(0,24)
#define BUTTON_4       NRF_GPIO_PIN_MAP(0,25)

#define BUTTON_PULL    1 //poner pullup interno a los botones

#define BUTTONS_ACTIVE_STATE 0

#define BUTTONS_LIST { BUTTON_1, BUTTON_2, BUTTON_3, BUTTON_4 }

// MONITORES
#define MONITOR_NUMBER 4

#define MONITOR_LIST { MONITOR1,MONITOR2,MONITOR3,MONITOR4 }

#define MONITOR1       NRF_GPIO_PIN_MAP(0,28)
#define MONITOR2       NRF_GPIO_PIN_MAP(0,29)
#define MONITOR3       NRF_GPIO_PIN_MAP(0,30)
```

```
#define MONITOR4       NRF_GPIO_PIN_MAP(0,31)

#endif
```

board_nrf52840_dongle.h

```
#ifndef BOARD_NRF
#define BOARD_NRF

// BOARD_PCA10059

/** @brief Macro for mapping port and pin numbers to values understandable for nrf_gpio functions. */
#define NRF_GPIO_PIN_MAP(port, pin) (((port) << 5) | ((pin) & 0x1F))

// LED definitions for PCA10059
// Each LED color is considered a separate LED
#define LEDS_NUMBER    4

#define LED1_G         NRF_GPIO_PIN_MAP(0,6)
#define LED2_R         NRF_GPIO_PIN_MAP(0,8)
#define LED2_G         NRF_GPIO_PIN_MAP(1,9)
#define LED2_B         NRF_GPIO_PIN_MAP(0,12)

#define LED_1          LED1_G
#define LED_2          LED2_R
#define LED_3          LED2_G
#define LED_4          LED2_B

#define LEDS_ACTIVE_STATE 0

#define LEDS_LIST { LED_1, LED_2, LED_3, LED_4 }

#if 0 // Botones

// There is only one button for the application
// as the second button is used for a RESET.
#define BUTTONS_NUMBER 1

#define BUTTON_1       NRF_GPIO_PIN_MAP(1,6)
#define BUTTON_PULL    1 //poner pullup interno a los botones

#define BUTTONS_ACTIVE_STATE 0

#define BUTTONS_LIST { BUTTON_1 }
#endif //botonos
```

```
#endif
```

hal_consumo_nrf.c

```
/* ****************************************************************************
 * P.H.2024: Consumo en NRF52840
 * implementacion para cumplir el hal_consumo.h
 */

#include <nrf.h>
#include <board_nrf52840dk.h>

#include <stdbool.h>

#define MAX_COUNTER_VALUE 0xFFFFFFFE                    // Maximo valor del
contador de 32 bits
#define HAL_TICKS2US                    15
        // funcionamos PCLK a 15 MHz de un total de 60 MHz CPU Clock

static volatile uint32_t timer0_int_count = 0;  // contador de 32 bits de veces que ha saltado
la RSI Timer0

/*
 * no hacemos nada
 */
void hal_consumo_iniciar(void){}

/*
 * Pone al procesador en estado de espera para reducir su consumo
 */
void hal_consumo_esperar(void){
        __WFI();                // Espera a que ocurra una interrupción
}


/*
 * Duerme al procesador para minimizar su consumo
 */
void hal_consumo_dormir(void) {

   NRF_POWER->SYSTEMOFF = POWER_SYSTEMOFF_SYSTEMOFF_Enter;

   // Solo saldrá por reset o SENSE GPIO si está programado
   while (1) {             // Solo saldrá por reset o SENSE GPIO si está programado
      __WFE();  // Espera por eventos (necesario para debug y entrar al estado de
SystemOff)
                }
```

```
}
```

hal_ext_int_nrf.c

```
/* ****************************************************************************
 * P.H.2024: Módulo de interrupciones en NRF52840
 * implementacion para cumplir el hal_ext_int.h
 */

#include <nrf.h>
#include <board_nrf52840dk.h>

#include <stdbool.h>

void (*ptr_hal_ext_int)();

/*
 * deshabilita las interrupciones
 */
void hal_ext_int_deshabilitar_botones_irqs(void) {
        NRF_GPIOTE->INTENCLR = (GPIOTE_INTENCLR_IN0_Msk << 0) |
                (GPIOTE_INTENCLR_IN0_Msk << 1) |
                (GPIOTE_INTENCLR_IN0_Msk << 2) |
                (GPIOTE_INTENCLR_IN0_Msk << 3);

  NVIC_DisableIRQ(GPIOTE_IRQn);
}

/*
 * habilita las interrupciones
 */
void hal_ext_int_habilitar_botones_irqs(void) {

        NRF_GPIOTE->EVENTS_IN[0] = 0;
  NRF_GPIOTE->EVENTS_IN[1] = 0;
  NRF_GPIOTE->EVENTS_IN[2] = 0;
  NRF_GPIOTE->EVENTS_IN[3] = 0;

        NRF_GPIOTE->INTENSET = (GPIOTE_INTENSET_IN0_Msk << 0) |
                (GPIOTE_INTENSET_IN0_Msk << 1) |
                (GPIOTE_INTENSET_IN0_Msk << 2) |
                (GPIOTE_INTENSET_IN0_Msk << 3);

        NVIC_EnableIRQ(GPIOTE_IRQn);
}
```

```
void hal_ext_int_deshabilitar_boton_irqs(uint8_t boton){
	NRF_GPIOTE->INTENSET = (GPIOTE_INTENCLR_IN0_Msk << (boton-1));
}

void hal_ext_int_habilitar_boton_irqs(uint8_t boton){
	NRF_GPIOTE->EVENTS_IN[boton-1] = 0;
	NRF_GPIOTE->INTENSET = (GPIOTE_INTENSET_IN0_Msk << (boton-1));
}



/*
 * maneja la interrupción por botón
 */
void GPIOTE_IRQHandler(void) {
	// Si han pulsado el BUTTON_1
	if (NRF_GPIOTE->EVENTS_IN[0]) {
   NRF_GPIOTE->EVENTS_IN[0] = 0;
   hal_ext_int_deshabilitar_boton_irqs(1);
		(*ptr_hal_ext_int)(1);
 }
	// Si han pulsado el BUTTON_2
	else if (NRF_GPIOTE->EVENTS_IN[1]) {
   NRF_GPIOTE->EVENTS_IN[1] = 0;
   hal_ext_int_deshabilitar_boton_irqs(2);
		(*ptr_hal_ext_int)(2);
 }
	// Si han pulsado el BUTTON_3
	else if (NRF_GPIOTE->EVENTS_IN[2]) {
   NRF_GPIOTE->EVENTS_IN[2] = 0;
   hal_ext_int_deshabilitar_boton_irqs(3);
		(*ptr_hal_ext_int)(3);
 }
	// Si han pulsado el BUTTON_4
	else if (NRF_GPIOTE->EVENTS_IN[3]) {
   NRF_GPIOTE->EVENTS_IN[3] = 0;
   hal_ext_int_deshabilitar_boton_irqs(4);
		(*ptr_hal_ext_int)(4);
 }
}


/*
 * configura cuatro canales GPIOTE para detectar eventos en los pines de botón.
 * Se configuran con una polaridad para que detecten de alto a bajo (HiToLo),
 * osea, cuando se presione el botón.
 */
void hal_ext_int_botones_iniciar(void(*funcion_callback)()){
```

```
	ptr_hal_ext_int = funcion_callback;

	// Configurar canales GPIOTE para cada botón
	NRF_GPIOTE->CONFIG[0] = (GPIOTE_CONFIG_MODE_Event <<
GPIOTE_CONFIG_MODE_Pos) |

	(BUTTON_1 << GPIOTE_CONFIG_PSEL_Pos) |

	(GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos);

	NRF_GPIOTE->CONFIG[1] = (GPIOTE_CONFIG_MODE_Event <<
GPIOTE_CONFIG_MODE_Pos) |

	(BUTTON_2 << GPIOTE_CONFIG_PSEL_Pos) |

	(GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos);

	NRF_GPIOTE->CONFIG[2] = (GPIOTE_CONFIG_MODE_Event <<
GPIOTE_CONFIG_MODE_Pos) |

	(BUTTON_3 << GPIOTE_CONFIG_PSEL_Pos) |

	(GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos);

	NRF_GPIOTE->CONFIG[3] = (GPIOTE_CONFIG_MODE_Event <<
GPIOTE_CONFIG_MODE_Pos) |

	(BUTTON_4 << GPIOTE_CONFIG_PSEL_Pos) |

	(GPIOTE_CONFIG_POLARITY_HiToLo << GPIOTE_CONFIG_POLARITY_Pos);

	// Habilitar interrupción en el módulo GPIOTE
	NRF_GPIOTE->INTENSET = (GPIOTE_INTENSET_IN0_Set <<
GPIOTE_INTENSET_IN0_Pos) |

(GPIOTE_INTENSET_IN1_Set << GPIOTE_INTENSET_IN1_Pos) |
		(GPIOTE_INTENSET_IN2_Set << GPIOTE_INTENSET_IN2_Pos) |
		(GPIOTE_INTENSET_IN3_Set << GPIOTE_INTENSET_IN3_Pos);

	hal_ext_int_habilitar_botones_irqs();
}


/*
 * devuelve el número del botón que está pulsado, si no hay ninguno, devuelve 0.
 */
uint32_t comprobar_boton_pulsado(uint8_t boton){
	if (boton == 1) {
```

```
                return !(NRF_GPIO->IN & (1 << BUTTON_1));
        }
        if (boton == 2) {
                return !(NRF_GPIO->IN & (1 << BUTTON_2));
        }
        if (boton == 3) {
                return !(NRF_GPIO->IN & (1 << BUTTON_3));
        }
        if (boton == 4) {
                return !(NRF_GPIO->IN & (1 << BUTTON_4));
        }
        return 0;
}

bool hay_dos_botones_pulsados(void) {
        uint8_t num_encendidos = 0;
        if (!(NRF_GPIO->IN & (1 << BUTTON_1))) {
                num_encendidos++;
        }
        if (!(NRF_GPIO->IN & (1 << BUTTON_2))) {
                num_encendidos++;
        }
        if (!(NRF_GPIO->IN & (1 << BUTTON_3))) {
                num_encendidos++;
        }
        if (!(NRF_GPIO->IN & (1 << BUTTON_4))) {
                num_encendidos++;
        }
        return (num_encendidos == 2);
}

uint8_t num_botones_pulsados_realmente(void) {
        uint8_t num_pulsados = 0;
        for (uint8_t i=1; i<=4; i++) {
                if (comprobar_boton_pulsado(i)) {
                        num_pulsados++;
                }
        }
        return num_pulsados;
}
```

hal_WDG_nrf.c

```
/* *
P.H.2024: Watchdog en NRF52840
implementacion para cumplir el hal_WDG.h
*/
```

```
#include <nrf.h>
#include <board_nrf52840dk.h>

#define WDT_FEED_VALUE 0x6E524635


/*
 * Esta función configura el Watchdog Timer para que reinicie el sistema si no
 * se "alimentado" antes de que se agote el tiempo especificado
 */
void hal_WDG_iniciar(uint32_t sec) {
  // Activo para que funcione hasta incluso en modo de bajo consumo
  NRF_WDT->CONFIG = (WDT_CONFIG_SLEEP_Run << WDT_CONFIG_SLEEP_Pos); //
Activo en modo de bajo consumo
  NRF_WDT->CRV = (sec * 1000 * 32768) / 1000; // Configura el tiempo de timeout
  NRF_WDT->RREN |= WDT_RREN_RR0_Msk; // Habilitar canal 0 para alimentar el WDT
  NRF_WDT->TASKS_START = 1; // Iniciar el Watchdog
}

/*
 * Función que se llama periodicamente para "alimentar" y se verifique
 * que el sistema sigue activo
 */
void hal_WDG_feed(void) {
    // Alimentar el Watchdog para evitar el reinicio por el canal 0
  NRF_WDT->RR[0] = WDT_FEED_VALUE;
}
```