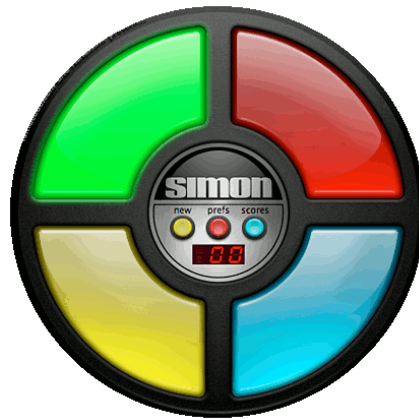


Proyecto Hardware - Proyecto Final :

Juego del Simon

Curso 2024-2025

Samuel Corpas Puerto (875301)
Daniel Salas Sayas (875308)



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza



Universidad
Zaragoza

Índice

1.	Resumen ejecutivo	2
2.	Introducción	3
3.	Objetivos	3
3.1.	Objetivos Técnicos	4
3.2.	Objetivos Funcionales	4
3.3.	Objetivos Metodológicos	4
4.	Metodología.....	4
4.1.	Tiempo	5
4.1.1.	Hal_tiempo	5
4.1.2.	Drv_timepo	6
4.2.	GPIO	6
4.3.	LEDS	7
4.4.	Consumo	7
4.5.	Monitores	7
4.6.	Cola Fifo	8
4.7.	Alarmas	8
4.8.	Gestor de eventos	9
4.9.	Interrupciones externas	9
4.10.	Botones	10
4.11.	Concurrencia	10
4.12.	WatchDog	10
4.13.	Juego final	11
4.14.	Elementos empleados	11
5.	Resultados	12
6.	Conclusión	14
7.	Referencias	16
8.	Anexos	17

1. Resumen Ejecutivo

Este proyecto documenta el desarrollo del juego Simón para los microcontroladores nRF52840 y LPC2105, como parte de la asignatura Proyecto Hardware del grado de Ingeniería Informática en la Universidad de Zaragoza. El objetivo principal fue implementar un sistema eficiente y modular que combinara LEDs, botones y temporizadores, optimizando el consumo energético y garantizando un juego fluido y cómodo para los usuarios.

El desarrollo se estructuró en tres etapas progresivas:

- 1. Etapa 1:** Configuración básica de LEDs, incluyendo encendido, apagado y parpadeo.
- 2. Etapa 2:** Implementación de temporizadores, gestión de eventos y optimización del consumo energético.
- 3. Etapa 3:** Integración de botones, gestión de interrupciones externas y desarrollo completo del juego Simón.

La metodología empleada destacó por su modularidad, separando el código en componentes reutilizables y específicos para cada microcontrolador. Entre las que encontramos en el código específico para cada microcontrolador, código relacionado con gestión de temporizadores, periféricos entrada/salida e interrupciones, así como otra serie de módulos run-time como gestor de eventos, alarmas o una cola fifo, que son necesarios para el correcto funcionamiento del juego final.

Para su desarrollo, se utilizaron herramientas avanzadas principalmente en el entorno de desarrollo de Keil uVision5, que permite la creación y depuración del código, y Power Profiler II para análisis de consumo en el caso del microcontrolador nRF52840.

Entre los resultados obtenidos, el juego Simón alcanzó una versión estable, con un consumo promedio de 733.99 μA en el nRF52840, significativamente menor a versiones previas. Además, el sistema demostró ser robusto, con reseteo automático ante fallos y optimización para el modo de bajo consumo y una experiencia de juego cómoda y agradable, pudiendo estar jugando al juego tanto tiempo como el usuario quiera sin tener errores.

Este proyecto no solo logró desarrollar el juego Simón, sino que estableció una base modular para futuros proyectos que involucren LEDs y botones, demostrando la importancia de la programación eficiente y modular.

2. Introducción

El propósito de esta memoria es documentar el desarrollo del juego Simón para los microcontroladores nRF52840 y LPC2105, con énfasis en garantizar una jugabilidad fluida y un consumo energético optimizado. Se describen los objetivos, la metodología seguida y los resultados obtenidos a lo largo de tres meses de trabajo en la asignatura Proyecto Hardware, del grado de Ingeniería Informática en la Universidad de Zaragoza.

El proyecto combina el manejo de temporizadores, LEDs y botones, adaptándose a las características específicas de cada microcontrolador. Por ello, el código incluye partes comunes y secciones específicas para cada hardware.

En el desarrollo de este proyecto se pueden diferenciar claramente 3 etapas, ordenadas de forma progresiva en cuanto al desarrollo del juego final, partiendo del parpadeo de un led, a poder desarrollar juegos con botones y leds:

- **Etapas 1:** Encendido, apagado y parpadeo de un LED.
- **Etapas 2:** Optimización del consumo energético, manejo de interrupciones temporales periódicas y gestión de una cola de eventos.
- **Etapas 3:** Integración de botones, implementación de un Watchdog y desarrollo completo del juego Simón.

3. Objetivos

Debido al periodo de tiempo que ha llevado el desarrollo de este proyecto y las distintas etapas, a lo largo de este se pueden encontrar distintos objetivos, los cuales son:

3.1. Objetivos técnicos

- Manipular y manejar periféricos de E/S y Temporizadores en ambos tipos de hardware.
- Desarrollar las Rutinas de Servicio Interrupción (ISR) en C.
- Optimizar rendimiento, tamaño y eficiencia energética.
- Diseñar un gestor de eventos para ejecutar tareas de forma orientada a eventos, incluyendo:
 - Gestión de una cola FIFO de eventos.

- Activación y manejo de alarmas asociadas a eventos.
- Reducción de tiempos de alarma mediante interrupciones temporales periódicas.
- Depurar eventos concurrentes asíncronos.
- Introducir interrupciones externas asíncronas asignadas a botones y gestionar su tratamiento.
- Identificar condiciones de carrera y su causa, así como solucionarlas.

3.2. Objetivos funcionales

- Obtener métricas y estadísticas en depuración que nos permitan adaptar algunas estructuras.
- Realizar pruebas independientes de cada uno de los módulos que componen el proyecto.
- Lograr una jugabilidad fluida y cómoda para el usuario.

3.3. Objetivos metodológicos

- Usar documentación, manuales y hojas técnicas
- Utilizar Programación modular
- Aplicar Abstracción del hardware
- Seguir una estructura fija de proyecto en lo referido a nombres y tipos de ficheros

4. Metodología

Como bien se ha nombrado en apartados anteriores, este proyecto se ha realizado de forma muy progresiva, partiendo de un simple parpadeo de led, hasta llegar al juego completo. Para facilitar su organización y comprensión, se estableció una estructura modular dividida en tres directorios:

1. **Común:** Contiene código independiente del hardware, compartido por ambos microcontroladores.
2. **LPC2105:** Incluye código específico para este microcontrolador.
3. **NRF52840:** Contiene código específico para este hardware.

Dentro de estos directorios, los módulos se clasificaron según su dependencia del hardware:

- **Independientes del hardware:** Sin un prefijo específico.
- **Abstracción de hardware:** Nombres que comienzan con *drv_*, implementan funciones reutilizables sin depender del hardware.
- **Dependientes del hardware:** Nombres que comienzan con *hal_*, implementan funciones específicas del microcontrolador.

Esta estructura se puede apreciar mejor en la **imagen 1** en la que se ven los módulos del proyecto con flechas que apuntan a los módulos utilizados por cada uno. La organización de colores simboliza las distintas capas, siendo la azul el juego, la verde los módulos run-time, la naranja los drivers y la morada los dependientes del hardware. Esta estructura modular garantiza claridad y reutilización del código durante el desarrollo, obteniendo de esta manera los siguientes módulos:

4.1. Tiempo

A la hora de realizar un proyecto de este tipo, el tiempo y su gestión a través de periféricos es un elemento principal. Para ello se utilizan los temporizadores hardware proporcionados por los microprocesadores.

4.1.1. Hal_tiempo(.c y .h)

Este módulo configura y maneja los timer 0 y 1 de los microcontroladores. El timer0 se emplea para medir el tiempo con alta precisión, contabilizando el tiempo de forma circular. En cuanto a el timer1 se emplea para generar interrupciones periódicas, ejecutando una función, que se pasará como parámetro, cada vez que llegue una interrupción. Debido a esto, su configuración varía en cuestión del microcontrolador:

- **LPC 2105:** En lo que respecta al timer0, se establece en el registro de comparación el máximo que el temporizador debe alcanzar antes de generar una interrupción y se configura para que genere una interrupción cada vez que llega a este valor, de forma que cada vez que llegue se resetea el contador y se suma un ciclo completado. Para habilitar sus interrupciones se conecta con el puerto 0 del *VICVectAddr* y con el canal 4 que es el correspondiente al timer0.

En cuanto al timer1, se establece en su registro de comparación el tiempo que se ha pasado como parámetro - 1 porque el timer empieza en 0, y de la misma manera que en el timer0 se genera una interrupción cada vez que llegue a este registro. Una vez salta la interrupción se llama a la función que se ha pasado como parámetro y se restablece el timer. Para habilitar las

interrupciones en este caso, se conecta con el puerto 1 del *VICVectAddr* y con el canal 5 que es el correspondiente al timer1.

- **nrf52840:** Ambos timers tienen una configuración inicial similar, inicializando su modo como timer, con resolución de 32 bits, con el prescaler a 0 para que la frecuencia sea de 16 MHz. En ambos se establece en su CC[0] (registro de comparación) el máximo que pueden alcanzar, siendo el máximo valor para timer0 y el valor pasado como parámetro para timer1. A su vez, ambos se configuran para que cuando alcancen dichos valores se reinicien y habiliten la interrupción mediante (*INTENSENSET*). En caso del timer0, cuando llega la interrupción, se limpian los eventos (mediante *EVENTS_COMPARE[0] = 0*) y se suma un ciclo recorrido, mientras que en el timer1, una vez se limpian los eventos pendientes, se llama a la función de callback pasada como parámetro.

4.1.2. drv_tiempo(.c y .h)

A partir del *hal_tiempo*, se pueden implementar una serie de funciones que empleen dichas funcionalidades del hardware. Estos ficheros cuentan con funciones para iniciar el temporizador, para calcular el tiempo actual en distintas unidades de medida y esperar un tiempo pasado por parámetro. Estas esperas son esperas activas que provocan un gran consumo en el procesador y que solo fueron empleadas a la hora de realizar los primeros parpadeos de led. La función mas importante de estos ficheros es *drv_tiempo_periodico_ms*, la cual recibe como parámetros, un tiempo, una función y un entero, siendo así los segundos que tardarán en llegar interrupciones periódicas y la función será la que se ejecutará cuando estas lleguen.

4.2. GPIO (hal_gpio)

Este módulo hace referencia a un conjunto de pines configurables presentes en ambos microcontroladores que permiten la comunicación con dispositivos externos y gestionar los mecanismos de entrada/salida, contando con las funciones necesarias, es decir, inicializar (todos los pines de entrada), leer, escribir y sentido (para configurar de entrada o salida).

- **LPC 2105:** En este microcontrolador, todo lo relacionado con la configuración de la dirección de un pin, entrada o salida, se realiza mediante el registro IODIR, poniendo un 0 cuando sea de entrada y un 1 de salida. De la misma forma, cuando se quiere obtener el valor de un pin, comprobamos el *IOPIN*, mientras que si queremos modificarlo emplearemos *IOSET*.
- **nrf52840:** Por parte de este microcontrolador, la dirección se configura mediante *NRF_GPIO->DIR*, aunque se añaden una serie de características a

cada pin, modificando su *PIN_CNF*, buscando desactivar la resistencia interna, conectar cada pin a lógica interna del microcontrolador y desactivando la detección de eventos. Esta configuración será modificada posteriormente al añadir botones. Para leer el valor del pin se emplea *NRF_GPIO -> IN* y para modificarlo *NRF_GPIO -> OUTSET*.

4.3. LEDS (*drv_leds*)

Como bien se puede intuir viendo el título del módulo, este se va a encargar de la gestión de los leds, empleando el módulo anterior. En cuestión del microcontrolador que se esté empleando, se obtiene la lista de led (pines) de este y la cantidad que hay. Para inicializar este módulo, se definen los pines de los leds como salida empleando la función de sentido del módulo anterior. También se usa la función escribir para apagar, encender o conmutar un led.

Con estos tres módulos se completa la etapa 1 nombrada anteriormente, la cual nos permite el parpadeo de un led jugando con las funciones del módulo tiempo para realizar las esperas.

4.4. Consumo (*drv_consumo*, *hal_consumo*)

Este módulo tiene la misma estructura que el módulo tiempo, sin embargo el driver únicamente llama a las funciones del hal. Esto se debe a que es un módulo totalmente dependiente del microcontrolador. Es un módulo sencillo con 3 funciones, iniciar, esperar y dormir:

- **LPC 2105:** La función iniciar no realiza ninguna tarea. Por su parte la función esperar habilita el despertar para las interrupciones externas *EINT0*, 1 y 2 mediante *EXTWAKE*, y mediante *PCON |= 1* se pone el procesador en modo bajo consumo. En cuanto al dormir, realiza lo mismo poniendo a 2 *PCON* para poner el procesador en modo sleep.
- **nrf52840:** De la misma manera que en LPC 2105, el iniciar no realiza ninguna tarea. En cuanto al esperar y el dormir únicamente realizan la instrucción *WFI*, de forma que saldrá de modo bajo consumo y sleep cuando llegue una interrupción, aunque previamente, en el dormir, se activa el modo de consumo más bajo disponible de la siguiente manera:
NRF_POWER->SYSTEMOFF=POWER_SYSTEMOFF_SYSTEMOFF_Enter;

4.5. Monitores (*drv_monitor*)

Debido a sus características, este módulo es similar a los leds. Con él, se busca marcar un monitor cuando entremos en modo bajo consumo y, posteriormente,

cuando haya overflow en alguna estructura futura. Consta de 3 funciones, iniciar, marcar y desmarcar, actuando como iniciar, encender y apagar en los leds, es decir, empleando el GPIO. La función marcar se llama en el driver de consumo, antes de entrar en modo espera bajo consumo, y desmarcar se llama al salir de esta espera.

4.6. Cola Fifo(rt_fifo)

La cola circular FIFO gestiona eventos asíncronos, almacenando el tipo de evento, su auxiliar y el tiempo de entrada. Puede almacenar hasta 64 eventos y, en caso de desbordamiento, marca el monitor recibido y entra en un bucle infinito. También sirve para depuración, permitiendo revisar el historial de eventos.. El módulo cuenta con funciones para inicializar la cola, encolar un evento siempre que se pueda y extraer el siguiente evento.

Se han realizado pruebas en modo Debug para verificar el funcionamiento, como la comprobación de desbordamiento (*test_cola_overflow*) y la correcta ejecución de las funciones de encolar y extraer (*test_cola_desencola*). Además, se han implementado variables estadísticas que registran el número de veces que se encola un evento de cada tipo y el total de eventos encolados, accesibles en modo Stats.

Con estos módulos ya es posible concluir la definida previamente como etapa 2, pudiendo realizar un parpadeo por interrupciones, teniendo en cuenta el consumo.

4.7. Alarmas(svc_alarma)

El gestor de alarmas permite programar tareas periódicas o esporádicas. Si no hay alarmas disponibles, se marca el monitor de *overflow*. Al vencer una alarma, se encola un evento y se desactiva si no es periódica.

El módulo utiliza un vector de 4 alarmas, cada una con un evento, auxiliar, tiempo inicial, tiempo actual, y flags de actividad y periodicidad.

Este módulo consta de una función iniciar, inicializando las estructuras, una función para activar alarmas, que comprueba si esa alarma es periódica o no y programa o reprograma la alarma (en caso de que exista) y por último una función tratar, la cual se ejecutará cada vez que llega una interrupción periódica de timer1 y resta los tiempos de las alarmas activas y encolando el evento en caso de que acabe la alarma.

En Debug se comprueba el *overflow* y en Stats se accede a estadísticas como el número total de alarmas activadas y el máximo de alarmas activas simultáneamente.

4.8. Gestor de eventos(rt_GE)

El gestor de eventos es el núcleo del sistema run-time, encargado de procesar los eventos y optimizar el consumo energético. Permite a las tareas suscribirse y cancelar suscripciones a eventos. Si no hay eventos, el sistema entra en modo de bajo consumo hasta que ocurra una interrupción. Además, monitoriza la inactividad del usuario mediante una alarma de inactividad, entrando en un modo de suspensión profunda si no se detecta actividad, optimizando así el consumo energético del procesador.

El módulo utiliza una estructura `rt_GE_t`, que contiene un vector con los eventos y una matriz de tareas suscritas, cada una con su callback y auxiliar, tal y como se ve en la **imagen 2**.

La función `iniciar`, inicializa esta estructura y suscribe los eventos necesarios. Por su parte la función `tratar` actúa de una forma u otra en cuestión del evento que le llega. Por último el lanzador, un bucle infinito que ejecuta los eventos finalizados, ejecutando todas las funciones suscritas a cada evento.

En modo Debug, se comprueba el *overflow* y en modo Stats se analizan variables como el número máximo de eventos por tipo.

4.9. Interrupciones externas(hal_ext_int)

Para permitir la interacción con el sistema, se asignan interrupciones externas a los pines GPIO. Este módulo es dependiente del hardware, pero implementa funciones iguales como inicializar, habilitar y deshabilitar interrupciones, y leer su estado.

- **LPC 2105:** Utilizaremos *EINT0*, *EINT1* y *EINT2* que harán la función de botones. Para ello se configura en *PINSEL0* o *PINSEL1*, dependiendo de que pin, se limpia mediante *IOCLR* y habilitamos las interrupciones externas poniendo en *VICVectAddr = 0x20* en cada pin y un 1 en *VICIntEnable*. En caso de que llegue una interrupción, se deshabilitan las interrupciones mediante *VICIntEnClr* y se llama al callback pasado como parámetro. Cuando se vuelvan a habilitar, previamente habrá que limpiar las pendientes mediante *EXTINT*.
- **nrf52840:** En este caso, se utilizan canales del *GPIOTE* para gestionar los botones. Cada botón se asigna a un canal del *GPIOTE*, que se configura modificando el registro *NRF_GPIOTE->CONFIG[channel]* para asociarlo al pin correspondiente y generar eventos en transiciones de alto a bajo. A cada pin se le asigna un callback con el pin pulsado como dato auxiliar. Cuando llega una interrupción, se ejecuta su callback asociado. Para habilitar las

interrupciones, primero se limpian los eventos pendientes con `NRF_GPIOTE->EVENTS_IN[channel] = 0` y luego se habilitan con `NRF_GPIOTE->INTENSET`. Para deshabilitar las interrupciones, se utiliza `NRF_GPIOTE->INTENCLR`.

4.10. Botones (drv_botones)

Los botones son esenciales para el juego del Simón, y este módulo se encarga de gestionarlos, utilizando el módulo de interrupciones externas. Cada botón tiene asignado un pin y un estado. Los pines se configuran como entradas, y se habilitan las interrupciones externas correspondientes.

Cada botón sigue una máquina de estados que incluye dos eventos: el de "pulsar botón" y el de "botón retardo". La máquina se puede observar en la **imagen 3**, comienza cuando un botón se pulsa, pasando de estado "reposo" a "entrando". En este estado, se activa una alarma periódica de 100 ms para verificar si el botón sigue presionado, cambiando al estado "esperando". Al soltar el botón, se pasa al estado "soltado", se habilitan las interrupciones y la máquina vuelve a su estado inicial.

El evento "pulsar botón" se genera cuando se detecta una pulsación, mientras que el evento "botón retardo" gestiona el flujo de la máquina de estados.

4.11. Concurrencia(hal_concurrencia)

Para evitar condiciones de carrera al acceder a recursos compartidos como la cola de eventos, se implementó un módulo que asegura la correcta gestión de secciones críticas. Este módulo utiliza funciones implementadas previamente en el startup para deshabilitar interrupciones antes de encolar, extraer y alimentar el WDT, minimizando las ventanas de error.

Esto se aplica debido a que sin su implementación cabe la posibilidad de que llegue una interrupción cuando se esté realizando una operación de la cola que modifica sus valores. Por eso es necesario desactivar las interrupciones cuando se encolan eventos desde el programa principal, ya que en modo interrupción estas ya están desactivadas. En el caso del LPC, no fue posible implementar esta solución porque en modo usuario no se pueden modificar los flags del CPSR.

4.12. WatchDog (hal_WDT)

El WDT es un elemento que se emplea para reiniciar el procesador si no es alimentado dentro de un tiempo específico, útil para evitar bloqueos o bucles infinitos. En este proyecto, se alimenta desde la función lanzador del gestor de

eventos. Aunque es dependiente del microcontrolador, ambos procesadores comparten funciones para inicializarlo y alimentarlo:

- **LPC 2105:** Para inicializarlo se introduce en tiempo en *WDTC*. Este tiempo se obtiene de los segundos recibidos como parámetro pasandolos a ciclos de reloj dependiendo de la frecuencia del procesador (15000000) y el pre-escalado del WDT (4). Una vez establecido el tiempo se habilita con *WDMOD*. Para alimentarlo, se emplea *WDFEED* y según especifican los manuales de LPC 2105, se ha de alimentar de forma atómica.
- **nrf52840:** Por parte de este procesador, se inicializa el tiempo en *NRF_WDT->CRV*, obteniendo este de la multiplicación de los segundos por 32768, siendo esta la frecuencia en hercios del reloj. Para poder alimentarlo se habilita el canal *WDT_REEN_RR0*. Para realizar dicha alimentación se iguala el registro a 0x6E524635, constante que se emplea en NRF52840.

4.13. Juego final (Simon)

Llegados a este punto, se tienen implementados todos los módulos necesarios para realizar juegos que únicamente necesiten botones y LEDs, el Simon en este caso. Para ello se ha realizado un módulo que contiene únicamente su inicialización y su máquina de estados. En su inicialización se genera una secuencia random de números en cuestión de los botones que hay, y se suscriben las funciones necesarias para cada evento. En este juego se cuenta con 9 tipos de eventos según se puede ver en la **imagen 2** junto a sus funciones.

La máquina de estados, compuesta por 5 estados (**ver imagen 4**), comienza encendiendo y apagando todos los LEDs como señal de inicio, seguido se muestra la primera secuencia y se activa una alarma de 5 segundos:

- **Si se pulsa un botón correcto pero no es el último de la secuencia:** Se reprograma la alarma y se espera otra pulsación.
- **Si se completa correctamente la secuencia:** Los LEDs parpadean indicando acierto, se añade un paso a la secuencia, y se reduce el tiempo de parpadeo.
- **Si hay un fallo o llega antes la alarma de 5 segundos:** Los LEDs parpadean en cascada, y el juego se reinicia desde el inicio de la secuencia.

La dificultad aumenta con cada ronda, reduciendo el tiempo de parpadeo, inicialmente de 800 ms, en 200 ms por ronda, hasta un mínimo de 100 ms, que hace casi imposible recordar secuencias largas.

4.14. Elementos empleados

El desarrollo de los módulos se realizó en Keil uVision5, que facilitó la creación, compilación y depuración del código gracias a sus herramientas avanzadas para monitorear registros y periféricos en tiempo real.

Las mediciones de consumo se llevaron a cabo exclusivamente para el nRF52840, utilizando la placa Power Profiler II y la aplicación nRF Connect Desktop, que permitió analizar consumos a través de gráficas y métricas detalladas.

5. Resultados

En esta sección se presentan los resultados obtenidos tras la implementación y pruebas de los módulos desarrollados en el juego Simon. Como principales resultados se definen los aspectos relacionados con la funcionalidad del juego, la estabilidad del sistema y el consumo energético, buscando de esta forma cumplir todos o la mayoría de objetivos propuestos.

El juego Simon ha alcanzado una versión estable y funcional, operando de manera fluida en ambos microcontroladores sin errores. El juego sigue la guía establecida, sin esperas activas ni bloqueos, con reseteo automático en caso de fallo. Además, tras 20 segundos sin interacción, el procesador entra en modo de bajo consumo y puede ser despertado por los botones.

Otro de los puntos principales del proyecto era el consumo, buscando optimizar lo máximo posible el rendimiento y la eficacia energética. Para ello se han empleado una serie de medidas explicadas en apartados anteriores con las que hemos logrado el consumo que se muestra en la siguiente imagen:

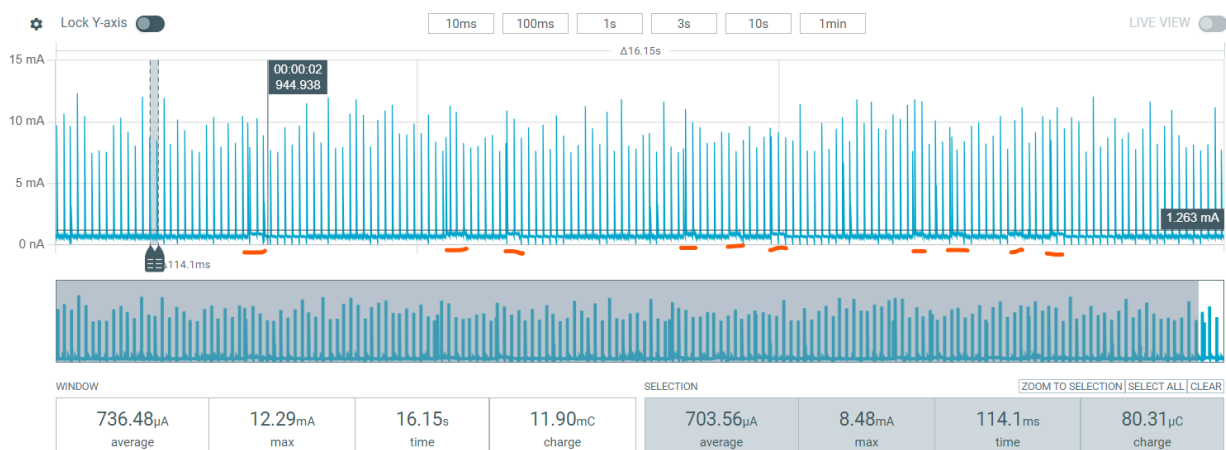


Imagen 5. Medidas de consumo proyecto final

La imagen que se presenta a continuación corresponde a un gráfico obtenido tras medir el programa final utilizando la aplicación nRF Connect Desktop. En ella se pueden observar diversas métricas, entre las cuales destaca la media de consumo, que es de 733.99 μ A, lo cual indica un consumo eficiente, especialmente si se compara con versiones anteriores del sistema. En esas versiones previas, la

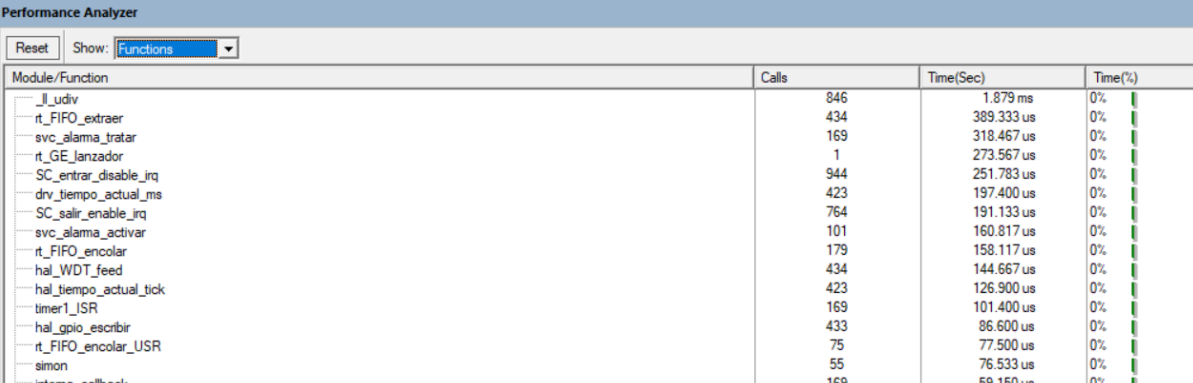
máquina de estados del juego Simon utilizaba las funciones de espera del módulo de tiempo, las cuales, en realidad, implementaban esperas activas, en lugar de utilizar alarmas. En dicha versión anterior, el consumo alcanzaba los 1.85 mA, como se puede observar en [la imagen 6](#).

En esta gráfica final se puede observar perfectamente el funcionamiento del proyecto, viendo como los picos de consumo llegan cada 100 ms, es decir, cada vez que llega una interrupción periódica del timer. Cabe señalar que algunos picos son más altos que otros, ya que ciertas interrupciones del temporizador también indican el final de una alarma, lo que provoca la ejecución de su correspondiente función de callback. Durante los períodos en los que no se produce ninguna interrupción, se observa cómo el procesador entra en modo de espera de bajo consumo, sin generar gasto alguno, tal y como se muestra en la [imagen 7](#) a mayor escala.

Asimismo, en el gráfico se pueden identificar las pulsaciones de los botones, que están marcadas con una línea naranja en la parte inferior, lo que permite seguir su progresión conforme avanza la secuencia del juego.

Como complemento también se pueden observar las distintas mediciones que se han hecho a lo largo del proyecto en la [imagen 8](#).

Por parte del LPC 2105 donde no se han medido consumos, si se pueden obtener medidas temporales referentes a funciones las cuales se pueden ver en la siguiente imagen:



Module/Function	Calls	Time(Sec)	Time(%)
_ll_udiv	846	1.879 ms	0%
rt_FIFO_extraer	434	389.333 us	0%
svc_alarma_tratar	169	318.467 us	0%
rt_GE_lanzador	1	273.567 us	0%
SC_entrar_disable_irq	944	251.783 us	0%
drv_tiempo_actual_ms	423	197.400 us	0%
SC_salir_enable_irq	764	191.133 us	0%
svc_alarma_activar	101	160.817 us	0%
rt_FIFO_encolar	179	158.117 us	0%
hal_WDT_feed	434	144.667 us	0%
hal_tiempo_actual_tick	423	126.900 us	0%
timer1_ISR	169	101.400 us	0%
hal_gpio_escribir	433	86.600 us	0%
rt_FIFO_encolar_USR	75	77.500 us	0%
simon	55	76.533 us	0%
interna_callback	169	59.150 us	0%

Imagen 9. Medida de tiempos por funciones para LPC 2105

En la imagen se puede observar que la función más costosa en términos de tiempo de ejecución es la función interna del procesador `_ll_udiv`, la cual se ejecuta debido a la división de un entero de 64 bits por uno de 32 bits. Omitiendo esta función, se puede ver que la operación que más tiempo consume es la extracción de elementos de la cola, lo cual se debe al tiempo de espera en caso de que la cola esté vacía. A continuación, se encuentra la función "alarma_tratar", que se encarga de reducir los tiempos de las alarmas y ejecutar sus correspondientes callbacks una vez finalizadas.

Otros aspectos relevantes son el lanzador, que se ejecuta una única vez pero cumple la función de gestionar todo el proceso, y el hecho de que el número de veces que se encola y se extrae un elemento de la cola es el mismo, lo cual demuestra el correcto funcionamiento y gestión de la cola.

6. Conclusión

Como conclusión, el desarrollo del juego Simón para los microcontroladores nRF52840 y LPC2105 conlleva una variedad de conceptos de programación y hardware, teniendo que conocer el funcionamiento interno de los microprocesadores que se vayan a emplear para poder exprimir todas sus funcionalidades.

A lo largo de las distintas etapas del proyecto, uno de los mayores objetivos era optimizar el consumo y se ha logrado optimizar de forma notable, obteniendo un juego completo al cual con una pila común de 1000 mAh podríamos estar jugando alrededor de 1361.53 horas sin que tuviéramos que cambiar las pilas y el juego diera ningún error..

Aunque el juego cumple con la funcionalidad básica, se podrían mejorar ciertos aspectos, debido a que la secuencia aleatoria de LEDs a encender, realmente es la misma todo el rato, de forma que habría que cambiar la semilla cada vez que se iniciara de nuevo el juego o se cometiera un fallo durante su desarrollo.

Queda pendiente también para futuras actualizaciones introducir una forma de concluir la partida, idea la cual se tenía prácticamente realizada en LPC 2105, de forma que si mantenías pulsados 2 botones a la vez, el juego terminaba. Sin embargo, por problema de tiempo y acceso limitado temporalmente a recursos de nRF52840, no se pudo poner a prueba para este microprocesador.

A pesar de esto, se consiguió una implementación modular y eficiente, que facilita la reutilización del código y la integración de funcionalidades comunes, a la vez que se mantiene la especificidad de cada plataforma. Los módulos clave, como la gestión de temporizadores, gestión de eventos, GPIOs, LEDs, y el consumo energético, fueron esenciales para el funcionamiento del juego, llegando así a cumplir así todos los objetivos propuestos inicialmente.

Este proyecto no solo permite la creación del Simón, sino que a partir de sus módulos es posible desarrollar cualquier juego en el que tengan que ver únicamente LEDs y botones, simplemente teniendo que realizar su mecánica de juego y adaptando tiempos y eventos.

7. Conclusión personal

Como conclusión personal como realizadores del proyecto, creemos que hemos sacado una gran cantidad de conocimientos acerca del hardware y cómo emplearlo en proyectos. Aunque no quita que haya sido un proceso costoso y duro, sobre todo al realizar cosas que no ibas a poder comprobar hasta pasada una semana, aunque a la vez era una gran sensación cuando consigues lograr que funcionase en la placa ya que llegabas a esas horas con cierta presión.

Por último, quiero concluir con que probablemente, el tener que realizar un proyecto como este durante todo un cuatrimestre, haga que aprendas más que tener que estudiarlo como teoría y realizar un examen, ya que de esta forma puedes comprobar las ideas y darte contra un muro las veces que haga falta que si invertias tiempo se podía sacar.

7.1. Horas invertidas aproximadas por persona

Práctica	Daniel Salas	Samuel Corpas
Práctica 2	10	8
Práctica 3	12	10
Práctica 4	30	25
Práctica 5	26	24
Memoria	6	4
Total	84	71

8. Referencias

- Manual de usuario LPC 2105: [user.manual.lpc2104.lpc2105.lpc2106-1.pdf](#)
- Manual de usuario nRF52840: [nRF52840_PS_v1.10.pdf](#)
- [PDF](#) de instrucciones para la redacción de una memoria técnica
- Moodle de Proyecto Hardware de Ingeniería Informática en UNIZAR: <https://moodle.unizar.es>
- Carpeta con ejemplos relacionados con el proyecto: <https://moodle.unizar.es>
- Páginas de consulta:
 - [Nordic Semiconductor](#)
 - <https://stackoverflow.com/>
 - <https://chatgpt.com/> (V. GPT-4o mini)

9. Anexos

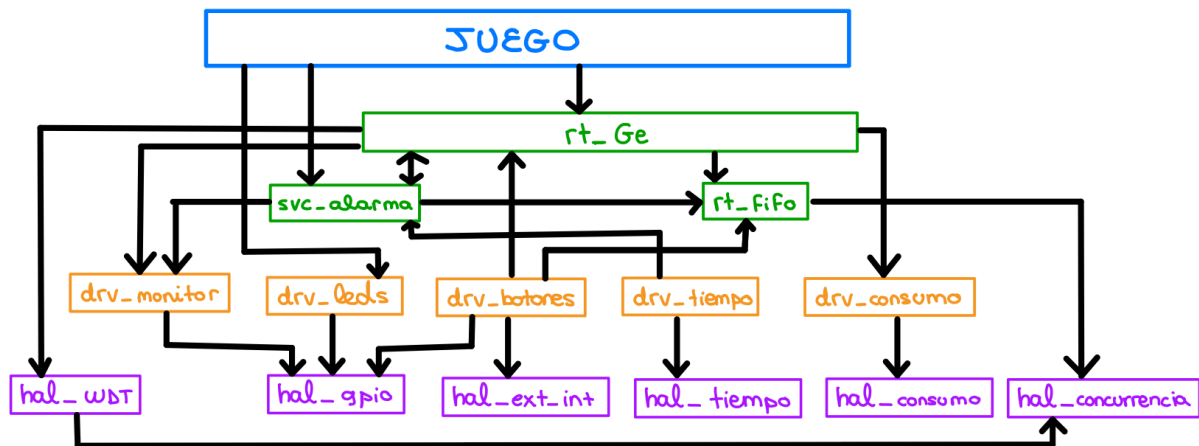


Imagen 1. Estructura del proyecto

ev_VOID				
ev_T_PERIODICO	svc_alarma_tratar			
ev_PULSAR_BOTON	rt_GE_tratar	drv_botones_tratar	simon	
ev_INACTIVIDAD	rt_GE_tratar			
ev_BOTON_RETARDO	drv_botones_tratar			
ev_ALARMA_BOTON	simon			
ev_SIMON	simon			
ev_PARPADEO	simon			
ev_ESPERA_LED	simon			

Imagen 2. Estructura interna del gestor de eventos, matriz de callbacks

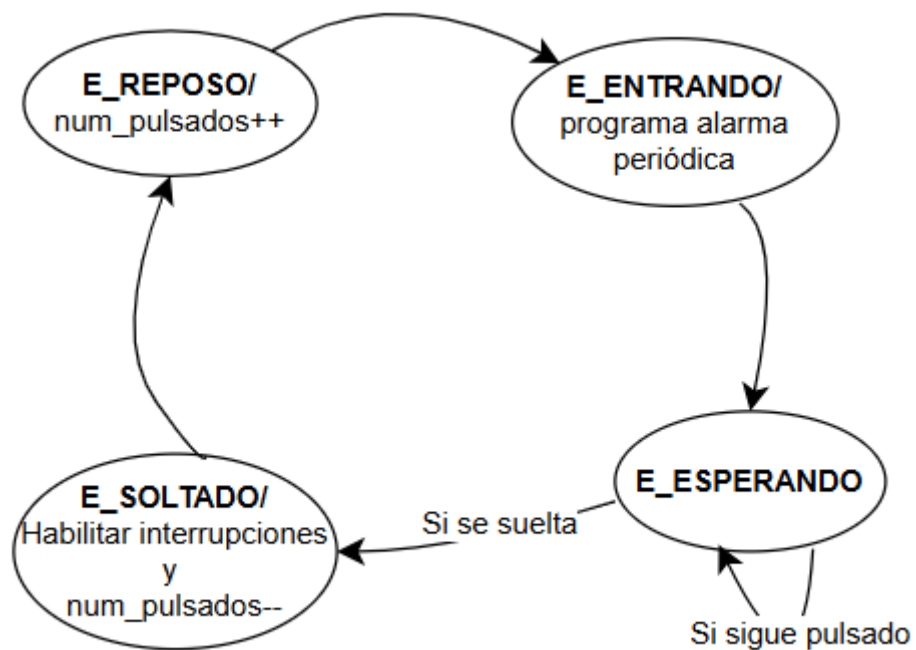


Imagen 3. Máquina de estados de los botones

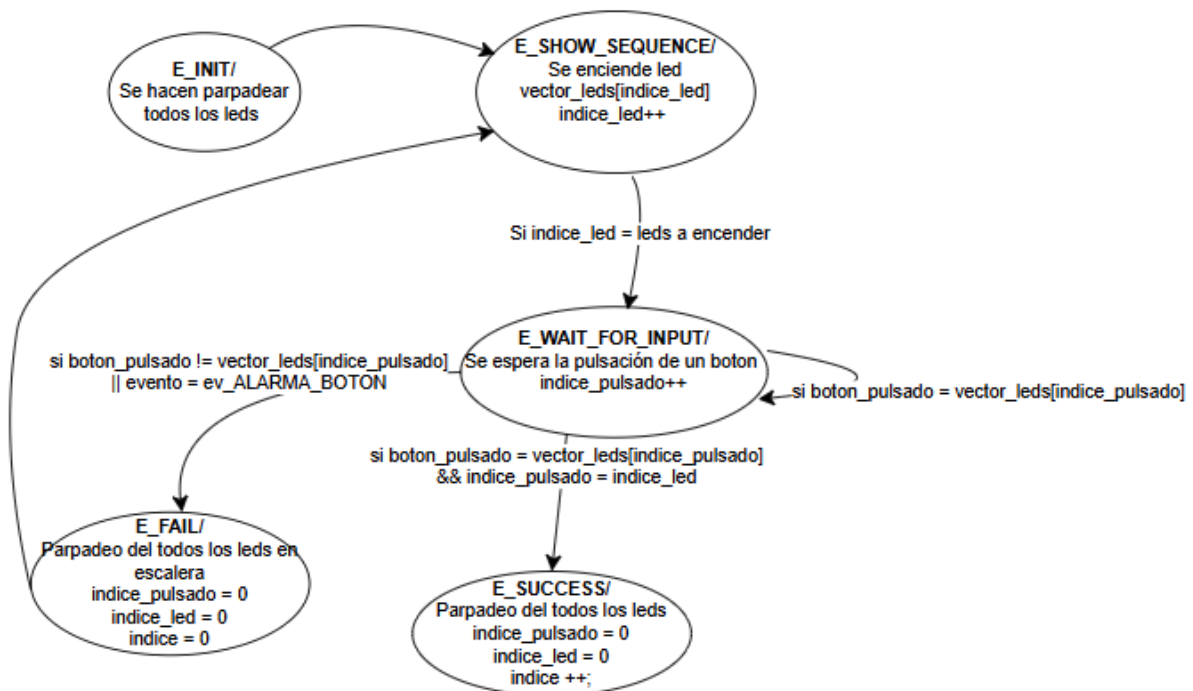


Imagen 4. Máquina de estados juego Simon

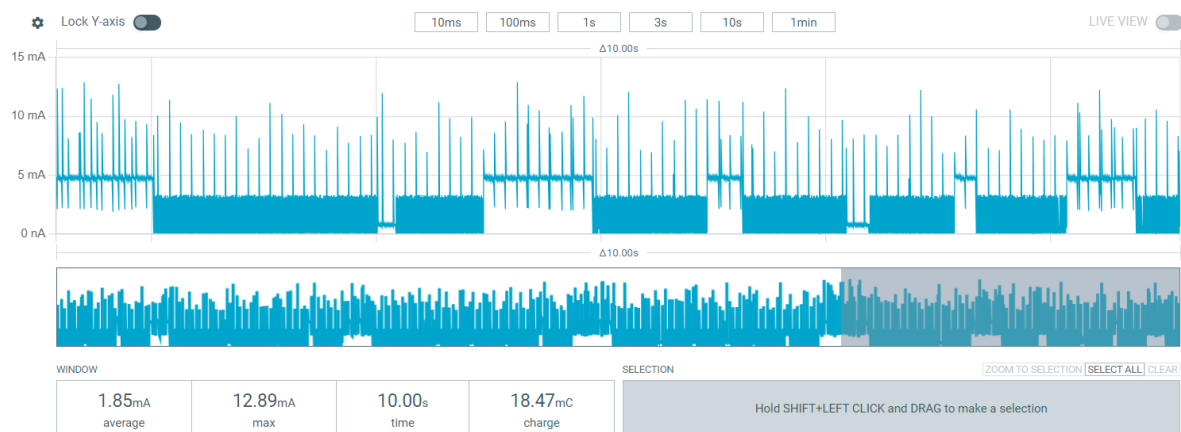


Imagen 6. Medidas de consumo versión de esperas activas, se puede observar el consumo mayor y el gran gasto provocado por las esperas activas, las zonas con gran acumulación de líneas.

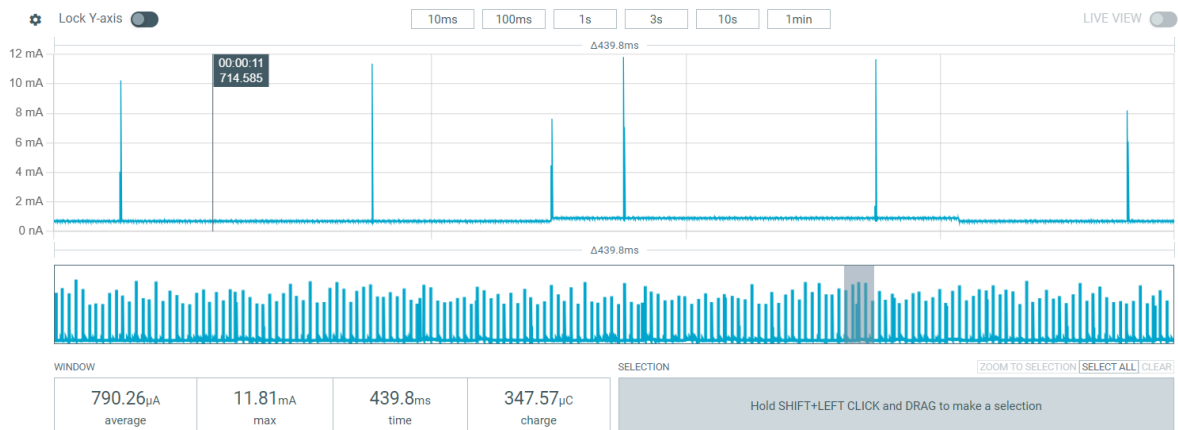


Imagen 7. Medidas finales agrandada, se puede ver que con el procesador en modo espera, el consumo es prácticamente nulo

Versión	Media Consumo
blink_v2 (parpadeo del led por interrupciones sin medidas de consumo)	5.78 mA
blink_v3 (parpadeo del led por interrupciones poniendo el procesador en modo espera hasta llegar la interrupción)	957 uA
blink_v3_bis (10 parpadeos de un led y se duerme el procesador)	461.79uA

Imagen 8. Resto de medidas tomadas a lo largo del desarrollo del proyecto. En esta tabla se puede observar claramente el progreso conforme se iban añadiendo medidas de consumo

```

void testCola_overflow(uint32_t id){
    rt_FIFO_inicializar(4);
    for (int i = 0; i<75; i++){
        rt_FIFO_encolar(ev_T_PERIODICO, i);
    }

}

void testCola_desencola(uint32_t id){
    rt_FIFO_inicializar(4);
    EVENTO_T evento;
    uint32_t dato;
    Tiempo_us_t tiempo;
    for (int i = 0; i<10; i++){
        rt_FIFO_encolar(ev_T_PERIODICO, i);
    }
    uint32_t datos0 = fifo.estadisticas[0];
    uint32_t datos1 = fifo.estadisticas[1];

    for (int i = 0; i<10; i++){
        rt_FIFO_extraer(& evento, & dato, & tiempo);
    }

}

```

Imagen 10. Código de test realizados para el módulo de la cola FIFO, en este test se observan una primera función que encola eventos sin extraer, superando límite de la cola, de esta forma se comprueba que salte overflow. En cuanto a la segunda función, se encolan 10 eventos periódicos con distintos auxiliares y se extraen, de forma que se podrá comprobar que se desencolan en el orden correcto.

```

void test_GE_overflow(uint32_t id){
    rt_GE_iniciar(id);
    for (int i = 0; i<(RT_GE_MAX_SUSCRITOS+1); i++){
        svc_GE_suscribir(ev_T_PERIODICO, rt_FIFO_encolar);
    }

}

```

Imagen 11. Código realizado como test del gestor de eventos para comprobar el funcionamiento del overflow, suscribiendo 1 evento mas del máximo, de forma que cuando se vaya a realizar la última subscripción, saltará overflow

```
void test_alarma_overflow(uint32_t id){
    svc_alarma_iniciar(4, rt_FIFO_encolar, ev_T_PERIODICO, rt_FIFO_encolar_USR);
    svc_alarma_activar(1000, ev_T_PERIODICO, 1);
    svc_alarma_activar(1000, ev_BOTON_RETARDO, 1);
    svc_alarma_activar(1000, ev_INACTIVIDAD, 1);
    svc_alarma_activar(1000, ev_VOID, 1);
    svc_alarma_activar(1000, ev_PULSAR_BOTON, 1);
}
```

Imagen 12. Código de test para comprobar el overflow en las alarmas, activando 5 alarmas, más de las permitidas, de forma que al activar la última saltará overflow.

main.c

```
/* *****  
 * P.H.2024: Driver/Manejador de los Leds  
 *  
 * blink practica 2 de proyecto hardware 2024  
 */  
  
#include <stdint.h>  
#include <stdbool.h>  
#include "ifdebug.h"  
#include "hal_gpio.h"  
#include "drv_leds.h"  
#include "drv_tiempo.h"  
#include "rt_GE.h"  
#include "rt_evento_t.h"  
#include "drv_consumo.h"  
#include "drv_botones.h"  
#include "hal_WDT.h"  
#include "rt_fifo.h"  
#include "simon.h"  
  
static uint32_t Num_leds;  
static uint32_t Num_botones;  
  
#ifdef STATS  
    Tiempo_ms_t tiempo_pulsacion;  
    Tiempo_ms_t tiempo2;  
    Tiempo_ms_t tiempo1;  
#endif  
  
#ifdef DEBUG  
void pruebaWDT(uint32_t id){  
    drv_led_encender(4);  
  
    hal_WDT_iniciar(10);  
  
    drv_led_encender(id);  
    uint32_t i = 0;  
    while (1) {  
        // El WDT no se alimentará, por lo tanto, el sistema  
        se reiniciará  
        i++;  
    }  
}
```

```

        if(i == 100){
            hal_WDT_feed();
        }
    }

}

#endif

/* *****
* MAIN, Programa principal.
* para la primera sesion se debe usar la funcion de blink_v1 sin temporizadores
* para la entrega final se debe incocar a blink_v2
*/
int main(void){
    //uint32_t Num_Leds, Num_Botones;

    hal_gpio_iniciar(); // llamamos a iniciar gpio antesde que lo hagan los
drivers

    /* Configure LED */
    Num_leds = drv_leds_iniciar();
    drv_tiempo_iniciar();
    rt_FIFO_inicializar(4);
    rt_GE_iniciar(4);
    svc_alarma_iniciar(4, rt_FIFO_encolar_USR, ev_T_PERIODICO,
rt_FIFO_encolar);
    hal_WDT_iniciar(6);
    Num_botones = drv_botones_iniciar(ev_PULSAR_BOTON,
ev_BOTON_RETARDO, rt_FIFO_encolar);
    if (Num_leds > 0){

        #ifdef DEBUG
            drv_led_encender(1);
            test_GE_overflow(4);
            drv_led_encender(2);
            testCola_desencola(4);
            testCola_overflow(4);
            pruebaWDT(3);
            test_alarma_overflow(4);

        #endif

        simon_iniciar(Num_botones, Num_leds);
    }
}

```



```

        #ifdef STATS
        uint32_t veces_encolado[EVENT_TYPES] = {};
        for (uint32_t i=0; i<EVENT_TYPES; i++){
            veces_encolado[i] = rt_FIFO_estadisticas(i);
        }
        uint32_t max_cola = maximo_cola();
        uint32_t alarmas_activas = maximo_alarmas_activas();
        uint32_t alarmas_totales = totales_alarmas();
        uint32_t eventos_suscritos[EVENT_TYPES] = {};
        for (uint32_t i=0; i<EVENT_TYPES; i++){
            eventos_suscritos[i] = eventos_encolados(i);
        }
        #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; //definido en board_xxx.h en cada placa...
}

/**
 * enciende el led id, si id es cero ... no enciende ninguno?, todos? decidis vosotros
 */
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 complementaria a encender
 */
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
}

```

```
#endif //LEDS_NUMBER > 0
}
```

```
//otras???
```

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>
```

```
/**
 * nicializa los led, los deja apagados y devuelve el numero de leds disponibles en la
plataforma
 */
```

```
uint32_t drv_leds_iniciar(void);
```

```
/**
 * enciende el led id, si id es cero ... no enciende ninguno?, todos? decidis vosotros
 */
```

```
void drv_led_encender(uint32_t id);
```

```
/**
 * funcion complementaria a encender
 */
```

```
void drv_led_apagar(uint32_t id);
```

```
/**
 * conmuta el led de on a off y viceversa
 */
```

```
void drv_led_conmutar(uint32_t id);
```

```
#if 0
```

```
/**
 * otras funciones de alto nivel que podrian sernos utiles
 * podeis realizarlas opcionalmente o si las usais en vuestra app
 */
```

```
uint32_t drv_led_estado(uint32_t id);
void drv_leds_encender_todos();
void drv_leds_apagar_todos();
```

```
#endif
```

```
#endif
```

drv_tiempo.c

```
/* *****
```

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

```
#include "drv_tiempo.h"
```

```
#include "hal_tiempo.h"
```

```
#define TODO 0    //pa que no de error de compilacion con el proyecto vacio,
modificar
```

```
static uint32_t hal_ticks2us = HAL_TICKS2US;
```

```
/**
```

```
 * nicializa 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){
    uint64_t ticks_actuales = hal_tiempo_actual_tick();
```

```
    // Convierte los ticks a microsegundos usando la constante hal_ticks2us
    Tiempo_us_t tiempo_us = ticks_actuales / hal_ticks2us;
```

```
    return tiempo_us;
}
```

```
/**
```

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

```

*/
Tiempo_ms_t drv_tiempo_actual_ms(void){
    Tiempo_us_t tiempo_us = drv_tiempo_actual_us();

    // Convierte el tiempo a milisegundos (1 ms = 1000 us)
    Tiempo_ms_t tiempo_ms = tiempo_us / 1000;

    return tiempo_ms;
}

/**
 * retardo: esperar un cierto tiempo en milisegundos
 */
void drv_tiempo_esperar_ms(Tiempo_ms_t ms){
    Tiempo_ms_t tiempo_inicio = drv_tiempo_actual_ms();
    while ((drv_tiempo_actual_ms() - tiempo_inicio) < ms)
    {
        // Espera activa
    }
}

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

    // Esperar hasta que el tiempo actual sea mayor o igual al tiempo especificado
    while (tiempo_actual < ms) {
        // Actualizamos el tiempo actual
        tiempo_actual = drv_tiempo_actual_ms();
    }

    // Devuelve el tiempo actual
    return tiempo_actual;
}

/* *****
 * Activacion Periodica contador de ticks
 */
static uint32_t callback_id;
static void(*f_callback)();

static void interna_callback(void){

```

```

        f_callback(callback_id, drv_tiempo_actual_ms());
    }

void drv_tiempo_periodico_ms(Tiempo_ms_t ms, void(*funcion_callback)(), uint32_t
id) {
    callback_id = id;
    f_callback = funcion_callback;

    if (ms == 0) {
        // Si el periodo es 0, detener el temporizador
        hal_tiempo_reloj_periodico_tick(0, interna_callback);
    } else {
        // Convertir el periodo de ms a ticks
        uint32_t periodo_en_ticks = ms * hal_ticks2us * 1000; // Conversión de ms a
ticks
        hal_tiempo_reloj_periodico_tick(periodo_en_ticks, interna_callback);
    }
}

```

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>

/**
* 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;

/**
* nicializa el reloj y empieza a contar
*/
void drv_tiempo_iniciar(void);

/**

```

```

* tiempo desde que se inicio el temporizador en microsegundos
*/
Tiempo_us_t drv_tiempo_actual_us(void);

/**
* tiempo desde que se inicio el temporizador en milisegundos
*/
Tiempo_ms_t drv_tiempo_actual_ms(void);

/**
* retardo: esperar un cierto tiempo en milisegundos
*/
void drv_tiempo_esperar_ms(Tiempo_ms_t ms);

/**
* esperar hasta un determinado tiempo (en ms), devuelve el tiempo actual
*/
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);

#endif

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

#ifndef HAL_GPIO
#define HAL_GPIO

#include <stdint.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;

/**
* 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);

/* *****
* Acceso a los GPIOs
*
* optimizado para campos/datos de mas de un bit
* gpio_inicial indica el primer bit con el que operar.
* num_bits indica cuántos bits con los que queremos operar.
*/

/**
* 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);

/**
* 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);

/**
* 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);

/* *****
 * Acceso a los GPIOs
 *
 * a gpio unico (optimizar accesos a un solo bit)
 */

/**
 * 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);

/**
 * La función devuelve un entero (bool), sera cero si el gpio es cero, sera distinto de
 * cero en caso contrario.
 */
uint32_t hal_gpio_leer(HAL_GPIO_PIN_T gpio);

/**
 * Escribe en el gpio el valor
 */
void hal_gpio_escribir(HAL_GPIO_PIN_T gpio, uint32_t valor);

#endif

```

hal_tiempo.h

```

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

#ifndef HAL_TIEMPO
#define HAL_TIEMPO
#define HAL_TICKS2US          15
    // funcionamos PCLK a 15 MHz de un total de 60 MHz CPU Clock

#include <stdint.h>

/**
 * 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);

/**
* nos devuelve el numero total de ticks desde que se inicio la cuenta
*/
uint64_t hal_tiempo_actual_tick(void);

/**
* TODO para la practica 3, no se si hacer hal propio...
* programa una activacion periodica cada periodo_en_tick ticks de la maquina
*/
void hal_tiempo_reloj_periodico_tick(uint32_t periodo_en_tick,
void(*funcion_callback)());

#endif

```

drv_consumo.c

```

#include "drv_consumo.h"
#include "hal_consumo.h"
#include "drv_monitor.h"

/* *****
* Inicialización del driver de consumo
* Por ahora, no requiere inicialización específica, pero mantenemos la función
* por consistencia con los demás módulos.
*/
void drv_consumo_iniciar(uint32_t monid) {
    // Llamamos a la función de inicialización del HAL si fuese necesario
    hal_consumo_iniciar();
    drv_monitor_iniciar();
}

/* *****
* Poner el procesador en modo de espera (WFI)
*/
void drv_consumo_esperar(void) {
    // Llama a la función del HAL para esperar
    drv_monitor_desmarcar(3);
    hal_consumo_esperar();
}

```

```

        drv_monitor_marcar(3);
    }

/* *****
* Poner el procesador en modo de dormir (futuro)
* Este modo aún no está implementado en la plataforma nRF, pero
* dejamos la estructura preparada.
*/
void drv_consumo_dormir(void) {
    // Llamará a una función del HAL cuando se implemente
    hal_consumo_dormir();
}

```

drv_consumo.h

```

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

#ifndef DRV_CONSUMO_H
#define DRV_CONSUMO_H

#include <stdint.h>

// Inicialización del driver de consumo
void drv_consumo_iniciar(uint32_t monid);

// Poner el procesador en modo de espera (WFI)
void drv_consumo_esperar(void);

// Poner el procesador en modo de dormir (función pendiente de implementación futura)
void drv_consumo_dormir(void);

#endif // DRV_CONSUMO_H

```

hal_consumo.h

```

#ifndef HAL_CONSUMO_NRF_H
#define HAL_CONSUMO_NRF_H

// Inicialización del hardware relacionado con el consumo
void hal_consumo_iniciar(void);

```

```

// Poner el procesador en modo de espera (WFI)
void hal_consumo_esperar(void);

// Poner el procesador en modo de dormir (función pendiente de implementación
futura)
void hal_consumo_dormir(void);

#endif // HAL_CONSUMO_NRF_H

```

drv_monitor.h

```

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

#ifndef DRV_MONITOR
#define DRV_MONITOR

#include <stdint.h>

/**
* inicializa los monitores, los deja desmarcados y devuelve el numero de monitores
disponibles en la plataforma
*/
uint32_t drv_monitor_iniciar(void);

/**
* marca el monitor
*/
void drv_monitor_marcar(uint32_t id);

/**
* desmarca el monitor
*/
void drv_monitor_desmarcar(uint32_t id);

#endif

```

drv_monitor.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_monitor.h"
#include "board.h"

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

/**
* inicializa los monitores, los deja desmarcados y devuelve el numero de monitores
disponibles en la plataforma
*/
uint32_t drv_monitor_iniciar(){
    #if MONITOR_NUMBER > 0
        for (uint32_t i = 0; i < MONITOR_NUMBER; ++i)
            {
                hal_gpio_sentido(monitor_list[i],
                HAL_GPIO_PIN_DIR_OUTPUT);
                drv_monitor_desmarcar(i+1);
            }
    #endif //MONITOR_NUMBER > 0

    return MONITOR_NUMBER; //definido en board_xxx.h en cada placa...
}

/**
* marca el monitor
*/
void drv_monitor_marcar(uint32_t id){

```

```

        #if MONITOR_NUMBER > 0
            if ((id <= MONITOR_NUMBER) && (id >0))
hal_gpio_escribir(monitor_list[id-1], MONITOR_ACTIVE_STATE);
        #endif //MONITOR_NUMBER > 0

    }

/**
 * desmarca el monitor
 */
void drv_monitor_desmarcar(uint32_t id){
    #if MONITOR_NUMBER > 0
        if ((id <= MONITOR_NUMBER) && (id >0))
hal_gpio_escribir(monitor_list[id-1], ~MONITOR_ACTIVE_STATE);
    #endif //MONITOR_NUMBER > 0
}

//otras???

```

rt_evento_t.h

```

#ifndef RT_EVENTO_T_H
#define RT_EVENTO_T_H

#include <stdint.h>
typedef enum {
    ev_VOID = 0, // default, vacio
    ev_T_PERIODICO = 1, //
    ev_PULSAR_BOTON = 2,
    ev_INACTIVIDAD = 3,
    ev_BOTON_RETARDO = 4,
    ev_ALARMA_BOTON = 5,
    ev_SIMON = 6,
    ev_PARPADEO = 7,
    ev_ESPERA_LED = 8
} EVENTO_T; //mapea a uint32_t

#define EVENT_TYPES 9
#define ev_NUM_EV_USUARIO 1
#define ev_USUARIO {ev_PULSAR_BOTON}

#endif

```

rt_fifo.c

```
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "hal_concurrencia.h"
#include "rt_fifo.h"

#define FIFO_SIZE 64 // Tamaño de la cola circular

typedef struct {
    EVENTO_T id_evento;
    uint32_t auxData;
    Tiempo_us_t tiempo_us; // Timestamp en microsegundos
} EVENTO;

typedef struct {
    EVENTO buffer[FIFO_SIZE]; // Almacena los eventos
    uint32_t head;           // Índice de la cabeza de la cola
    uint32_t tail;           // Índice de la cola
    uint32_t count;          // Número de elementos en la cola
    uint32_t overflow_monitor; // Monitor de overflow (ID del monitor)
    uint32_t estadisticas[EVENT_TYPES]; // Estadísticas de ocurrencia de eventos
} FIFO_T;

static FIFO_T fifo; // Cola FIFO
#ifdef STATS
static uint32_t maxCola = 0;
#endif

// Función de inicialización
void rt_FIFO_inicializar(uint32_t monitor_overflow) {
    fifo.head = 0;
    fifo.tail = 0;
    fifo.count = 0;
    fifo.overflow_monitor = monitor_overflow;
#ifdef STATS
    for (size_t i = 0; i < 3; ++i) {
        fifo.estadisticas[i] = 0; // Inicializa cada elemento a 0
    }
#endif
}
```

```

// Función para encolar un evento
void rt_FIFO_encolar(EVENTO_T ID_evento, uint32_t auxData) {
    // Verifica si la cola está llena
    if (fifo.count >= FIFO_SIZE) {
        drv_monitor_marcar(fifo.overflow_monitor);

        while (1) {
        }

    }
    fifo.count++;
    // Añade el evento a la cola y avanza el índice head
    fifo.buffer[fifo.head].id_evento = ID_evento;
    fifo.buffer[fifo.head].auxData = auxData;
    fifo.buffer[fifo.head].tiempo_us = drv_tiempo_actual_ms(); // Agrega timestamp

    fifo.head = (fifo.head + 1) % FIFO_SIZE;
    #ifdef STATS
        if (fifo.count > maxCola){
            maxCola = fifo.count;
        }
    // Actualiza estadísticas
    fifo.estadisticas[ID_evento]++;
    if (ID_evento != 0){
        fifo.estadisticas[0]++;
    }
    #endif
}

void rt_FIFO_encolar_USR(EVENTO_T ID_evento, uint32_t auxData) {
    SC_entrar_disable_irq();
    rt_FIFO_encolar(ID_evento, auxData);
    SC_salir_enable_irq();
}

uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t *auxData, Tiempo_us_t
*TS) {
    SC_entrar_disable_irq();
    if (fifo.count == 0) {
        SC_salir_enable_irq();
        return 0; // La cola está vacía
    }
}

```



```

// Extrae el evento más antiguo
*ID_evento = fifo.buffer[fifo.tail].id_evento;
*auxData = fifo.buffer[fifo.tail].auxData;
*TS = fifo.buffer[fifo.tail].tiempo_us;

// Avanza el índice tail y reduce el conteo de la cola
fifo.tail = (fifo.tail + 1) % FIFO_SIZE;
fifo.count--;

    SC_salir_enable_irq();

return fifo.count + 1; // Retorna el número de eventos sin procesar
}

#ifdef STATS
// Función para obtener estadísticas de un evento específico
uint32_t rt_FIFO_estadisticas(uint32_t ID_evento) {
    return fifo.estadisticas[ID_evento]; // Devuelve el conteo de ese evento
}

uint32_t maximoCola(void){
    return maxCola;
}
#endif

#ifdef DEBUG
void testColaOverflow(uint32_t id){
    rt_FIFO_inicializar(4);
    for (int i = 0; i<64; i++){
        rt_FIFO_encolar(ev_T_PERIODICO, i);
    }
}

void testColaDesencola(uint32_t id){
    rt_FIFO_inicializar(4);
    EVENTO_T evento;
    uint32_t dato;
    Tiempo_us_t tiempo;
    for (int i = 0; i<10; i++){
        rt_FIFO_encolar(ev_T_PERIODICO, i);
    }
    uint32_t datos0 = fifo.estadisticas[0];
    uint32_t datos1 = fifo.estadisticas[1];

```

```

        for (int i = 0; i<10; i++){
            rt_FIFO_extraer(& evento, & dato, & tiempo);
        }
    }

void eventos_encolados(void){
    for (uint32_t i=0; i<EVENT_TYPES; i++){

    }
}
#endif

```

rt_fifo.h

```

#ifndef RT_FIFO_H
#define RT_FIFO_H

#include "ifdebug.h"
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "rt_evento_t.h"
#include "drv_tiempo.h"
#include "drv_monitor.h"

// Función de inicialización
void rt_FIFO_inicializar(uint32_t monitor_overflow);
// Función para encolar un evento
void rt_FIFO_encolar(EVENTO_T ID_evento, uint32_t auxData);
//Funcion de encolar eventos para el procesador modo usuario
void rt_FIFO_encolar_USR(EVENTO_T ID_evento, uint32_t auxData);

// Función para extraer un evento de la cola
uint8_t rt_FIFO_extraer(EVENTO_T *ID_evento, uint32_t *auxData, Tiempo_us_t *TS);

#ifdef STATS
// Función para obtener estadísticas de un evento específico
uint32_t rt_FIFO_estadisticas(uint32_t ID_evento);

uint32_t maximoCola(void);

```

```
#endif
```

```
#ifdef DEBUG
```

```
void test_cola_overflow(uint32_t id);
```

```
void test_cola_desencola(uint32_t id);
```

```
#endif
```

```
#endif
```

rt_GE.h

```
#include <stdint.h>
```

```
#include <stdbool.h>
```

```
#include <string.h>
```

```
#include "rt_evento_t.h"
```

```
#include "drv_tiempo.h"
```

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

```
#include "svc_alarma.h"
```

```
#include "drv_monitor.h"
```

```
#include "drv_consumo.h"
```

```
#include "ifdebug.h"
```

```
#define RT_GE_MAX_SUSCRITOS 4 // Número máximo de tareas por evento
```

```
//Función que inicializa el gestor de eventos
```

```
void rt_GE_iniciar(uint32_t monitor);
```

```
//Función que realiza toda la gestión de eventos
```

```
void rt_GE_lanzador(void);
```

```
//Función que subscribe un evento con su función callback asociada
```

```
void svc_GE_suscribir(uint32_t evento, void (*f_callback)());
```

```
//Función que cancela un evento suscrito
```

```
void svc_GE_cancelar(EVENTO_T evento, void (*f_callback)());
```

```
//Función que actuará de una forma u otra en cuestión del evento que se extrae
```

```
void rt_GE_tratar(EVENTO_T evento, uint32_t auxiliar);
```

```
#ifdef STATS
```

```
uint32_t eventos_encolados(uint32_t ID_evento);
```

```
#endif
```

```
#ifdef DEBUG
```

```
void test_GE_overflow(uint32_t id);
#endif
```

rt_GE.c

```
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "hal_WDT.h"
#include "rt_GE.h"

#ifdef STATS
static uint32_t max_eventos_subs[EVENT_TYPES] = {};
    Tiempo_ms_t tiempo_en_espera;
#endif

// Estructura de la tarea suscrita a un evento
typedef struct {
    void (*f_callback)(uint32_t, uint32_t); // Función de callback
    uint32_t auxData; // Datos auxiliares de la tarea suscrita
} rt_GE_tarea_t;

// Estructura del gestor de eventos
typedef struct {
    rt_GE_tarea_t tareas_suscritas[EVENT_TYPES][RT_GE_MAX_SUSCRITOS]; //
    Tareas suscritas por tipo de evento
    uint32_t num_tareas_suscritas[EVENT_TYPES]; // Número de tareas suscritas
    por evento
} rt_GE_t;

static rt_GE_t gestor_eventos; // Instancia del gestor de eventos
static uint32_t mon_overflow;

void rt_GE_iniciar(uint32_t monitor){
    //rt_FIFO_inicializar(3);
    for (int i = 0; i < EVENT_TYPES; ++i) {
        gestor_eventos.num_tareas_suscritas[i] = 0; // Ninguna tarea suscrita
        inicialmente
        for (int j = 0; j < RT_GE_MAX_SUSCRITOS; ++j) {
            gestor_eventos.tareas_suscritas[i][j].f_callback = NULL; // No hay callback
            asignado inicialmente
            gestor_eventos.tareas_suscritas[i][j].auxData = 0; // No hay datos auxiliares
```

```

    }
}

    svc_GE_suscribir(ev_INACTIVIDAD, rt_GE_tratar);
    svc_GE_suscribir(ev_PULSAR_BOTON, rt_GE_tratar);

}

void rt_GE_lanzador(){
    EVENTO_T evento;
    uint32_t auxData;
    Tiempo_us_t timestamp;

    svc_alarma_activar(200000, ev_INACTIVIDAD, 0); // 10 segundos
    como ejemplo de timeout
    // Intentamos extraer un evento de la cola FIFO
    while(1){
        hal_WDT_feed();
        if (rt_FIFO_extraer(&evento, &auxData, &timestamp) != 0) {
            // Si hay un evento, lo despachamos a las tareas
            suscritas
                                // Llamamos al callback de todas las
            tareas suscritas a este evento
                                for (int i = 0; i <
            gestor_eventos.num_tareas_suscritas[evento]; ++i) {
                                    if
            (gestor_eventos.tareas_suscritas[evento][i].f_callback != NULL) {
                                                //
            Llamamos al callback con el evento y los datos auxiliares

            gestor_eventos.tareas_suscritas[evento][i].f_callback(evento, auxData);
                                                }
                                    }
            } else {
                                // Si no hay eventos, ponemos el sistema en
            espera
                                #ifdef STATS
                                Tiempo_ms_t tiempo1 =
            drv_tiempo_actual_ms();
                                #endif
                                drv_consumo_esperar();
                                #ifdef STATS
                                Tiempo_ms_t tiempo2 =
            drv_tiempo_actual_ms();

```

```

                                tiempo_en_espera = tiempo_en_espera +
(tiempo2 - tiempo1);
                                #endif
                                }
                                }
}

void svc_GE_suscribir(uint32_t evento, void (*f_callback)()){

    if (gestor_eventos.num_tareas_suscritas[evento] <
RT_GE_MAX_SUSCRITOS) {
        int idx = gestor_eventos.num_tareas_suscritas[evento];
        gestor_eventos.tareas_suscritas[evento][idx].f_callback = f_callback;
        gestor_eventos.tareas_suscritas[evento][idx].auxData = 0;
        gestor_eventos.num_tareas_suscritas[evento]++;
        #ifdef STATS
            max_eventos_subs[evento]++;
        #endif
    } else {
        // Si no hay espacio para más tareas suscritas, gestionamos el overflow
        while (1) {
            drv_monitor_marcar(mon_overflow);
        }
    }
}

void svc_GE_cancelar(EVENTO_T evento, void (*f_callback)()){
    bool encontrado = false;
    for (int i = 0; i < gestor_eventos.num_tareas_suscritas[evento]; ++i) {
        if (gestor_eventos.tareas_suscritas[evento][i].f_callback == f_callback) {
            for (int j = i; j < gestor_eventos.num_tareas_suscritas[evento] - 1; ++j) {
                gestor_eventos.tareas_suscritas[evento][j] =
gestor_eventos.tareas_suscritas[evento][j + 1];
            }
            gestor_eventos.num_tareas_suscritas[evento]--;
            encontrado = true;
            break;
        }
    }
    if (!encontrado) {
        while (1) {
            drv_monitor_marcar(mon_overflow);
        }
    }
}

```

```

    }
void rt_GE_tratar(EVENTO_T evento, uint32_t auxiliar){
    if (evento == ev_PULSAR_BOTON) {
        // Si el evento es de usuario, reprogramamos la alarma de inactividad
        svc_alarma_activar(20000, ev_INACTIVIDAD, 0); // 20 segundos de tiempo de
espera
    } else if (evento == ev_INACTIVIDAD) {
        // Si la alarma de inactividad vence, ponemos el sistema en modo de bajo
consumo
        drv_consumo_dormir();
    }
}
}

```

```

#ifdef STATS
uint32_t eventos_encolados(uint32_t ID_evento) {
    return max_eventos_subs[ID_evento]; // Devuelve el conteo de ese evento
}
#endif

```

```

#ifdef DEBUG
void test_GE_overflow(uint32_t id){
    rt_GE_iniciar(id);
    for (int i = 0; i<(RT_GE_MAX_SUSCRITOS+1); i++){
        svc_GE_suscribir(ev_T_PERIODICO, rt_FIFO_encolar);
    }
}
#endif

```

svc_alarma.c

```

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

```

```

#include "svc_alarma.h"
#include "rt_GE.h"
#include "drv_tiempo.h"
#include "drv_monitor.h"
#include "ifdebug.h"

```

```

#define PERIODO_ALARMA_MS 100

```

```

#define MASK_PERIODIC_BIT 0x80000000 // Bit de mayor peso para definir
periodicidad
#define MASK_DELAY 0x7FFFFFFF // Mascara para el retardo en ms

static void(*fun_callback)(); //callback que sera
utilizado por interrupciones
static void(*fun_callback_USR)(); //callback que sera utilizado en modo
usuario
static uint32_t mon_overflow;

#ifdef STATS
static uint32_t num_activas_max = 0;
static uint32_t num_activas = 0;
static uint32_t num_totales = 0;
#endif

// Configuración de la cola y definición de tipos
#define svc_ALARMAS_MAX 4

typedef struct {
    uint32_t retardo_ms; // Tiempo en ms; el bit más significativo indica periodicidad
    uint32_t tiempo_inicial; // Almacenamos el valor original del retardo
    EVENTO_T ID_evento; // Identificador del evento
    uint32_t auxData; // Datos auxiliares a pasar al evento
    bool activa; // Estado de la alarma (activa/inactiva)
    bool periodica;
} ALARMA_T;

static ALARMA_T alarmas[svc_ALARMAS_MAX];

//static int num_alarmas_activas = 0; // Contador de alarmas activas

void svc_alarma_iniciar(uint32_t overflow, void (*f_callback_USR)(), uint32_t
ID_evento, void (*f_callback)()){
    fun_callback = f_callback;
    //se usara de callback la función encolar, Sera usada por las
interrupciones
    fun_callback_USR = f_callback_USR;
    //en caso de que la alarma se active desde modo usuario, como las
interrupciones estan activas, se desactivan en seccion critica
    mon_overflow = overflow;
    svc_GE_suscribir(ID_evento, svc_alarma_tratar);
    //num_alarmas_activas = 0;
    for (int i = 0; i < svc_ALARMAS_MAX; i++) {

```



```

        alarmas[i].activa = false;
    }
    drv_tiempo_periodico_ms(PERODO_ALARMA_MS, fun_callback,
ID_evento);

}
void svc_alarma_activar(uint32_t retardo_ms, uint32_t ID_evento, uint32_t
auxData){

    bool es_periodica = (retardo_ms & MASK_PERIODIC_BIT) != 0; // Verificar
bit de periodicidad
    uint32_t retardo_mask = retardo_ms & MASK_DELAY;          // Extraer el
retardo en ms
    uint32_t retardo = retardo_mask / 100;
    // Si el retardo es 0, cancelar alarma
    if (retardo == 0) {
        for (int i = 0; i < svc_ALARMAS_MAX; i++) {
            if (alarmas[i].activa && alarmas[i].ID_evento == ID_evento) {
                alarmas[i].activa = false; // Cancelar la alarma
                #ifdef STATS
                    num_activas--;
                #endif

                return; //Hacemos return porque ya hemos encontrado la alarma y hemos
hecho el objetivo que teniamos
            }
        }
        return; // Hacemos return porque quieres cancelar una alarma que no existe
    }

    //En caso de que la alarma para ese evento ya exista la
reprogramamos con el tiempo incial
    for (int i = 0; i < svc_ALARMAS_MAX; i++) {
        if (alarmas[i].activa && alarmas[i].ID_evento == ID_evento) {
            // Reprogramar alarma existente
            alarmas[i].retardo_ms = retardo;
            alarmas[i].periodica = es_periodica;
            alarmas[i].auxData = auxData;

            alarmas[i].tiempo_inicial = retardo;
            #ifdef STATS
                num_totales++;
            #endif

            return; //Hacemos return porque se ha encontrado la alarma que
se queria activar y se ha reprogramado
        }
    }
}

```

```

}

// En caso de que la alarma sea nueva, se busca el hueco y se añade
for (int i = 0; i < svc_ALARMAS_MAX; i++) {
    if (!alarmas[i].activa) {
        alarmas[i].activa = true;
        alarmas[i].periodica = es_periodica;
        alarmas[i].retardo_ms = retardo;
        alarmas[i].ID_evento = ID_evento;
        alarmas[i].auxData = auxData;

        alarmas[i].tiempo_inicial = retardo;
#ifdef STATS
        num_activas++;
        num_totales++;
        if (num_activas_max < num_activas){
            num_activas_max =
num_activas;
        }
#endif

        return;
    }
}

// Si no hay alarmas disponibles, indicar desbordamiento (overflow)
while (1) {
    drv_monitor_marcar(mon_overflow);
}

}

void svc_alarma_tratar(EVENTO_T evento, uint32_t aux){
    EVENTO_T id_evento;
    uint32_t auxData;
    if (evento == ev_T_PERIODICO){
        for (int i = 0; i < svc_ALARMAS_MAX; i++) {
            if (alarmas[i].activa) {
                if (alarmas[i].retardo_ms > 0) {

alarmas[i].retardo_ms--;

                }

                if (alarmas[i].retardo_ms == 0) {
                    id_evento =
alarmas[i].ID_evento;

```

```

alarmas[i].auxData;
asociado a la alarma

fun_callback_USR(id_evento, auxData);

es periódica
{
alarmas[i].retardo_ms = alarmas[i].tiempo_inicial;

alarmas[i].activa = false;

}
}
}

#ifdef STATS
uint32_t maximo_alarmas_activas(void){
    return num_activas_max;
}

uint32_t totales_alarmas(void){
    return num_totales;
}
#endif

#ifdef DEBUG
void test_alarma_overflow(uint32_t id){
    svc_alarma_iniciar(4, rt_FIFO_encolar, ev_T_PERIODICO,
rt_FIFO_encolar_USR);
    svc_alarma_activar(1000, ev_T_PERIODICO, 1);
    svc_alarma_activar(1000, ev_BOTON_RETARDO, 1);
    svc_alarma_activar(1000, ev_INACTIVIDAD, 1);
    svc_alarma_activar(1000, ev_VOID, 1);
    svc_alarma_activar(1000, ev_PULSAR_BOTON, 1);
}

```

auxData =

// LLamar callback

// Verificar si la alarma

if (alarmas[i].periodica)

} else {

}

}

}

}

}

}

#ifdef STATS

uint32_t maximo_alarmas_activas(void){

return num_activas_max;

}

uint32_t totales_alarmas(void){

return num_totales;

}

#endif

#ifdef DEBUG

void test_alarma_overflow(uint32_t id){

svc_alarma_iniciar(4, rt_FIFO_encolar, ev_T_PERIODICO,
rt_FIFO_encolar_USR);

svc_alarma_activar(1000, ev_T_PERIODICO, 1);

svc_alarma_activar(1000, ev_BOTON_RETARDO, 1);

svc_alarma_activar(1000, ev_INACTIVIDAD, 1);

svc_alarma_activar(1000, ev_VOID, 1);

svc_alarma_activar(1000, ev_PULSAR_BOTON, 1);

}

```
#endif
```

svc_alarma.h

```
#ifndef SVC_ALARMA_H // Nombre único para evitar colisiones  
#define SVC_ALARMA_H
```

```
#include <stdint.h>  
#include <stdbool.h>  
#include <string.h>  
#include "rt_evento_t.h"
```

```
//Iniciamos toda la estructura de las alarmas
```

```
void svc_alarma_iniciar(uint32_t overflow, void (*f_callback_USR)(), uint32_t  
ID_evento, void (*f_callback)());
```

```
//Activamos una alarma para dentro de retardo_ms ms con el evento ID_evento
```

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

```
//Funcion que actua de una forma u otra cuando le llega una alarma
```

```
void svc_alarma_tratar(EVENTO_T evento, uint32_t aux);
```

```
#endif // Cierra la guarda de inclusión
```

```
#ifdef STATS
```

```
uint32_t maximo_alarmas_activas(void);
```

```
uint32_t totales_alarmas(void);
```

```
#endif
```

```
#ifdef DEBUG
```

```
void test_alarma_overflow(uint32_t id);
```

```
#endif
```

hal_ext_int.h

```
#ifndef HAL_EXT_INT_H
```

```
#define HAL_EXT_INT_H
```

```
#include <stdint.h>
```

```
#include "rt_evento_t.h"
```

```

// Definiciones de las funciones a implementar por cada plataforma

//Iniciamos y habilitamos las interrupciones externas
void hal_ext_int_iniciar(uint32_t pin, uint32_t auxData, void (*callback)(uint32_t
auxData));

//Habilitamos las interrupciones externas para ese pin
void hal_ext_int_habilitar_int(uint32_t pin);

// Deshabilita la interrupción en el pin dado
void hal_ext_int_deshabilitar_int(uint32_t pin);

// Habilita el despertar del sueño profundo por un pin
void hal_ext_int_habilitar_despertar(uint32_t pin);

// Deshabilita el despertar del sueño profundo por un pin
void hal_ext_int_deshabilitar_despertar(uint32_t pin);

//Leemos si las interrupciones externas estan activas para ese determinado pin
uint32_t hal_ext_int_leerINT(uint32_t pin);

#endif // HAL_EXT_INT_H

```

drv_botones.c

```

#include "drv_botones.h"
#include "board.h"    // Código específico para la configuración de pines
#include "rt_GE.h"    // Módulo gestor de eventos
#include "hal_tiempo.h" // Módulo de alarmas para programar retardos
#include "hal_ext_int.h" // Interfaz para manejo de pines GPIO

static uint32_t tiempo_ret = 100;
static boton_t botones[BUTTONS_NUMBER];
static uint32_t num_pulsados=0;

// Callback para manejo de eventos de pulsación
static void (*callback_encolar)(EVENTO_T ID_evento, uint32_t auxData);

#if BUTTONS_NUMBER > 0
    static const uint8_t boton_list[BUTTONS_NUMBER] = BUTTONS_LIST;
#endif

static void drv_cb_desde_hal(uint32_t pin){//pin to boton

```

```

uint32_t botonPulsado;
    for (uint32_t i = 0; i < BUTTONS_NUMBER; i++){
        if (pin == boton_list[i]){
            botonPulsado = (i+1);
        }
    }
    callback_encolar(ev_PULSAR_BOTON, botonPulsado);
}

// Función de inicialización de los botones
uint32_t drv_botones_iniciar(uint32_t EV_pulsar, uint32_t EV_retardo, void
(*callback)(EVENTO_T ID_evento, uint32_t auxData)) {
    callback_encolar = callback;

    // Configuración de pines con la interfaz hal_gpio
    for (uint32_t i = 0; i < BUTTONS_NUMBER; i++){
        botones[i].pin = boton_list[i];
        botones[i].estado = E_REPOSO;
        hal_gpio_sentido(boton_list[i], HAL_GPIO_PIN_DIR_INPUT);
        hal_ext_int_iniciar(boton_list[i], (i+1), drv_cb_desde_hal);
        hal_ext_int_habilitar_despertar(boton_list[i]);
    }
    svc_GE_suscribir(EV_pulsar, drv_botones_tratar);
    svc_GE_suscribir(EV_retardo, drv_botones_tratar);

    return BUTTONS_NUMBER;
}

// Función de tratamiento de eventos en la FSM
void drv_botones_tratar(uint32_t evento, uint32_t aux) {
    uint32_t botonPulsado;
    uint32_t pinPulsado;

    botonPulsado = aux;
    pinPulsado = boton_list[(aux - 1)];
    EstadoBoton_t estado_actual=botones[(botonPulsado-1)].estado;
    switch (estado_actual) {
        case E_REPOSO:
            svc_alarma_activar(tiempo_ret, ev_BOTON_RETARDO, botonPulsado);
            botones[botonPulsado-1].estado = E_ENTRANDO;
            num_pulsados++;

            break;
    }
}

```

```

case E_ENTRANDO:
    if (evento == ev_PULSAR_BOTON) {
    }
    else if (evento == ev_BOTON_RETARDO) {
        uint32_t tiempo = tiempo_ret;
        tiempo |= 0x80000000;
        svc_alarma_activar(tiempo,
ev_BOTON_RETARDO, botonPulsado);
        botones[botonPulsado-1].estado = E_ESPERANDO;
    }
    break;

case E_ESPERANDO:
    if (evento == ev_PULSAR_BOTON) {
    }
    else if (evento == ev_BOTON_RETARDO) {
        uint32_t estado_boton =
hal_ext_int_leerINT(pinPulsado);
        //uint32_t estado_boton =
hal_gpio_leer(pinPulsado);

        botones[botonPulsado-1].estado = E_SOLTADO;

        svc_alarma_activar(tiempo_ret, ev_BOTON_RETARDO, botonPulsado);

        //} else {
        /*
if(num_pulsados > 1){

        svc_alarma_activar(1, ev_INACTIVIDAD, botonPulsado);

        } else {

        svc_alarma_activar(1, ev_BOTON_RETARDO, botonPulsado);

        }*/

        /*svc_alarma_activar(30, ev_BOTON_RETARDO, botonPulsado);

        botones[botonPulsado-1].estado = E_SOLTADO;*/

        //}

    }
    break;

case E_SOLTADO:
    if (evento == ev_PULSAR_BOTON) {

```

```

    }
    else if (evento == ev_BOTON_RETARDO) {

hal_ext_int_habilita_int(pinPulsado);

        botones[botonPulsado-1].estado = E_REPOSO;
                                                num_pulsados--;
    }
    break;

default:
    estado_actual = E_REPOSO;
    break;
}
}

```

drv_botones.h

```

#ifndef DRV_BOTONES_H
#define DRV_BOTONES_H

#include <stdint.h>
#include "hal_gpio.h" // Interfaz para manejo de pines GPIO
#include "rt_evento_t.h"

// Definición de los estados de la FSM
typedef enum {
    E_REPOSO,
    E_ENTRANDO,
    E_ESPERANDO,
    E_SOLTADO
} EstadoBoton_t;

typedef struct {
    HAL_GPIO_PIN_T pin;           // Pin del botón
    EstadoBoton_t estado;         // Estado actual del botón
    uint32_t ultimo_tiempo;       // Último tiempo de cambio de estado
    // void (*callbackStruct)(EVENTO_T evento, uint32_t aux); // Función
    callback a ejecutar
} boton_t;

// Inicialización de los botones

```



```
uint32_t drv_botones_iniciar(uint32_t evento1, uint32_t evento2, void
(*callback)(EVENTO_T ID_evento, uint32_t auxData));
```

```
// Tratamiento de eventos de los botones
void drv_botones_tratar(uint32_t evento, uint32_t aux);
```

```
#endif // DRV_BOTONES_H
```

hal_WDT.h

```
#ifndef HAL_WDT_H
#define HAL_WDT_H
```

```
#include "ifdebug.h"
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
```

```
//Función que inicializa el WDT
void hal_WDT_iniciar(uint32_t sec);
```

```
//Función para alimentar el WDT
void hal_WDT_feed(void);
#endif
```

random.c

```
#include "random.h"
```

```
// Variables globales para almacenar el estado del generador (semilla)
static uint32_t seed = 0;
```

```
// Función para inicializar la semilla del generador de números aleatorios
void random_init(uint32_t init_seed) {
    if (init_seed != 0) {
        seed = init_seed;
    } else {
        seed = 123456789;
    }
}
```

```
uint32_t random_generate(uint32_t min, uint32_t max) {
    seed = (1103515245 * seed + 12345) % (1 << 31); // Generador congruencial
    lineal (LCG)
```

```

        uint32_t prueba = min + (seed % (max - min + 1));
    // Escalar el número aleatorio generado dentro del rango [min, max]
    return prueba;
}

```

random.h

```

#ifndef RANDOM_H
#define RANDOM_H

#include <stdint.h>

// Función para inicializar la semilla del generador de números aleatorios
void random_init(uint32_t seed);

// Función para generar un número aleatorio en un rango específico
uint32_t random_generate(uint32_t min, uint32_t max);

#endif

```

simon.c

```

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "ifdebug.h"
#include "hal_gpio.h"
#include "drv_leds.h"
#include "drv_tiempo.h"
#include "rt_GE.h"
#include "rt_evento_t.h"
#include "drv_consumo.h"
#include "drv_botones.h"
#include "hal_WDT.h"
#include "rt_fifo.h"
#include "random.h"
#include "simon.h"
typedef enum {
    e_INIT,
    e_SHOW_SEQUENCE,
    e_WAIT_FOR_INPUT,
    e_SUCCESS,
    e_FAIL
}

```

```

} _GameState;

static uint32_t vector_leds[8] = {}; // Declaración e inicialización
static uint32_t indice;
static uint32_t indice_pulsado;
static uint32_t indice_led;
static _GameState Estado_Simon = e_INIT;
static uint32_t iter;
static uint32_t leds_encendidos = 0;

static uint32_t tiempo_sin_pulsar = 5000;
static uint32_t tiempo_espera_led = 800;
static uint32_t tiempo_espera_inicial = 800;
static uint32_t tiempo_simon = 100;
static uint32_t tiempo_espera_encender = 800;

static uint32_t Num_leds;
static uint32_t Num_botones;

#ifdef STATS
    Tiempo_ms_t tiempo_pulsacion;
    Tiempo_ms_t tiempo2;
    Tiempo_ms_t tiempo1;
#endif

void simon_iniciar(uint32_t botones, uint32_t leds){
    Num_leds = leds;
    Num_botones = botones;
    random_init(123456789); // Usamos una semilla fija para obtener la misma
    secuencia cada vez
    uint32_t valor;
    for (uint32_t i = 0; i < 8; i++){
        valor = random_generate(1, Num_botones);
        vector_leds[i] = valor;
    }
    svc_GE_suscribir(ev_PULSAR_BOTON, simon);
    svc_GE_suscribir(ev_SIMON, simon);
    svc_GE_suscribir(ev_ALARMA_BOTON, simon);
    svc_GE_suscribir(ev_PARPADEO, simon);
    svc_GE_suscribir(ev_ESPERA_LED, simon);
    svc_alarma_activar(tiempo_simon, ev_SIMON, 5);
    rt_GE_lanzador();
}

```

//Esta funcion contiene el desarrollo del juego Simon

```
void simon(uint32_t evento, uint32_t boton){
    if(indice < 8){
        switch (Estado_Simon) {
        case e_INIT:
            if(evento == ev_SIMON){
                indice_pulsado=0;
                for (uint32_t i=0; i<=Num_leds; i++){
                    drv_led_encender(i);
                }
                svc_alarma_activar(tiempo_espera_led,
ev_PARPADEO, 6); //Activamos alarma de parpadeo de secuencia inicial
            }
            else if (evento == ev_PARPADEO){
                for (uint32_t i=0; i<=Num_leds; i++){
                    drv_led_apagar(i);
                }
                Estado_Simon = e_SHOW_SEQUENCE;
                svc_alarma_activar(tiempo_simon,
ev_SIMON, 5);
            }
            break;

        case e_SHOW_SEQUENCE:
            if(evento == ev_SIMON){
                iter = (indice + 1);
                leds_encendidos = 0;
                if (tiempo_espera_led > 200){
                    tiempo_espera_led =
tiempo_espera_led - 200;

                    tiempo_espera_encender =
tiempo_espera_encender - 200;
                } else {
                    tiempo_espera_led = 100;
                    tiempo_espera_encender = 100;
                }
                svc_alarma_activar(tiempo_espera_led,
ev_PARPADEO, 6);
            } else if (evento == ev_PARPADEO){
                drv_led_encender(vector_leds[indice_led]);
            }
        }
    }
}
```

```

        svc_alarma_activar(tiempo_espera_led,
ev_ESPERA_LED, 7);
    } else if (evento == ev_ESPERA_LED){
        drv_led_apagar(vector_leds[indice_led]);
        indice_led++;
        leds_encendidos++;
        if (leds_encendidos < iter){

svc_alarma_activar(tiempo_espera_led, ev_PARPADEO, 6);
        } else {

                                #ifdef STATS
                                tiempo1 =

drv_tiempo_actual_ms();

                                #endif

                                indice++;
                                Estado_Simon =

e_WAIT_FOR_INPUT;

svc_alarma_activar(tiempo_sin_pulsar, ev_ALARMA_BOTON, 0);
        }
    }
    break;

case e_WAIT_FOR_INPUT:
    #ifdef STATS
        tiempo2 = drv_tiempo_actual_ms();
        tiempo_pulsacion = tiempo2 - tiempo1;
    #endif

    if(boton == vector_leds[indice_pulsado]){
        indice_pulsado++;
        if(indice_pulsado == indice_led){
            Estado_Simon = e_SUCCESS;
            svc_alarma_activar(tiempo_simon,
ev_SIMON, 0);
        }else{

svc_alarma_activar(tiempo_sin_pulsar, ev_ALARMA_BOTON, 0);
        }
    }else{
        Estado_Simon = e_FAIL;

```

```

                                svc_alarma_activar(tiempo_simon,
ev_SIMON, 5);
                                }
                                break;

                                case e_SUCCESS:
                                if(evento == ev_SIMON){
                                svc_alarma_activar(0,
ev_ALARMA_BOTON, 5);
                                for (uint32_t i=0; i<=Num_leds; i++){
                                drv_led_encender(i);
                                }
                                svc_alarma_activar(tiempo_espera_inicial,
ev_PARPADEO, 6); //Activamos alarma de parpadeo de secuencia inicial
                                }
                                else if (evento == ev_PARPADEO){
                                for (uint32_t i=0; i<=Num_leds; i++){
                                drv_led_apagar(i);
                                }
                                indice_pulsado = 0;
                                indice_led = 0;
                                Estado_Simon = e_SHOW_SEQUENCE;
                                svc_alarma_activar(tiempo_simon,
ev_SIMON, 5);
                                }
                                break;

                                case e_FAIL:
                                if (evento == ev_SIMON){
                                leds_encendidos=0;
                                svc_alarma_activar(0,
ev_ALARMA_BOTON, 5);
                                svc_alarma_activar(tiempo_simon,
ev_PARPADEO, 6);
                                } else if(evento == ev_PARPADEO){
                                if (leds_encendidos<Num_botones){
                                drv_led_encender(leds_encendidos+1);
                                }
                                svc_alarma_activar(tiempo_espera_inicial,
ev_ESPERA_LED, 7);
                                } else if(evento == ev_ESPERA_LED){
                                drv_led_apagar(leds_encendidos+1);
                                leds_encendidos++;

```

```

        if (leds_encendidos < Num_botones){
            svc_alarma_activar(tiempo_simon,
ev_PARPADEO, 6);
        }
        else {
            indice_pulsado = 0;
            indice_led = 0;
            indice = 0;
            tiempo_espera_led =
tiempo_espera_inicial;
            tiempo_espera_encender =
tiempo_espera_inicial;
            Estado_Simon =
e_SHOW_SEQUENCE;
            svc_alarma_activar(tiempo_simon,
ev_SIMON, 5);
        }
    }
    break;
    default:
        break;
}
}
}

```

simon.h

```

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

void simon_iniciar(uint32_t botones, uint32_t leds);

void simon(uint32_t evento, uint32_t boton);

```

hal_concurrencia.h

```

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
//Funcion para entrar en seccion critica
uint32_t SC_entrar_disable_irq(void);

```

```
//Funcion para salir de seccion critica
void SC_salir_enable_irq(void);
```

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

#define MAX_COUNTER_VALUE 0xFFFFF000 // 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

/* *****
 * Timer0 contador de ticks
 */

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
 */
void timer0_ISR (void) __irq {
    timer0_int_count++;
    T0IR = 1; // Clear interrupt flag
    VICVectAddr = 0; // Acknowledge Interrupt
}

/* *****
 * Programa un contador de tick sobre Timer0, con maxima precisión y minimas
 interrupciones
 */
uint32_t hal_tiempo_iniciar_tick() {
```



```

        timer0_int_count = 0;
        T0MR0 = MAX_COUNTER_VALUE;           // Si TC = 1 interrupcion cada
us
        T0MCR = 3;                          // Generates an interrupt and resets the
count when the value of MR0 is reached

// configuration of the IRQ slot number 0 of the VIC for Timer 0 Interrupt
        VICVectAddr0 = (unsigned long)timer0_ISR; // set interrupt vector in 0
// 0x20 bit 5 enables vectored IRQs.
        // 4 is the number of the interrupt assigned. Number 4 is the Timer 0 (see
table 40 of the LPC2105 user manual
        VICVectCntl0 = 0x20 | 4;
        VICIntEnable = VICIntEnable | 0x00000010; // Enable Timer0 Interrupt

        T0TCR = 2; // Reinicia los contadores
        T0TCR = 1; // Empieza la cuenta

        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;
}

/* *****
 * Activacion periodica con timer 1
 */
static void(*f_callback)();                //puntero a funcion a llamar cuando salte la RSI
(en modo irq)

/* *****
 * Timer 1 Interrupt Service Routine
 * llama a la funcion de callback que se ejecutara en modo irq
 */
void timer1_ISR (void) __irq {
    f_callback();
    // Llamo a la función desde la RSI.

```

```

        T1IR = 1;                // Clear interrupt flag
        VICVectAddr = 0;        // Acknowledge Interrupt
    }

/* *****
* Dependiente del hardware Timer1
* 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)()){

    f_callback = 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
        T1MCR = 3;
        // Enable Timer1 Interrupt.
        VICVectAddr1 = (unsigned long)timer1_ISR;
        // 0x20 bit 5 enables vectored IRQs.
        // 5 is the number of the interrupt assigned. Number 5 is the Timer 1
        VICVectCntl1 = 0x20 | 5;
        VICIntEnable = VICIntEnable | 0x00000020;

        T1TCR = 3; // Reincia los contadores
        T1TCR = 1; // Empieza la cuenta
    } else {
        // Detiene el temporizador
        T1TCR = 0;
        VICIntEnClr = 0x20;        // Disable Timer1 Interrupt
    }
}

```

hal_gpio_lpc.c

```

/* *****

```

```

* P.H.2024: GPIOs en LPC2105, Timer 0 y Timer 1
* implementacion para cumplir el hal_gpio.h
* interrupciones externas para los botones lo dejamos para otro modulo aparte
*/

```

```

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

```

```

#include "hal_gpio.h"

```

```

/**
 * 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
}

```

```

/* *****
 * Acceso a los GPIOs
 *
 * gpio_inicial indica el primer bit con el que operar.
 * num_bits indica cuántos bits con los que queremos operar.
 */

```

```

/**
 * 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){

    uint32_t masc = ((1 << num_bits) - 1) << gpio_inicial;
    if (direccion == HAL_GPIO_PIN_DIR_INPUT){
        IODIR = IODIR & ~masc;
    }
    else if (direccion == HAL_GPIO_PIN_DIR_OUTPUT){
        IODIR = IODIR | masc;
    }
}

```

```

/**
 * 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){
    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){
    uint32_t masc_value = (valor & ((1 << num_bits) - 1)) << bit_inicial;
    uint32_t masc = ((1 << num_bits) - 1) << bit_inicial;
    uint32_t temp = IOPIN & ~masc;
    IOPIN = temp | masc_value;
    // limpia la mascara en el iopin y cambia sus bits de golpe
}

/* *****
 * Acceso a los GPIOs
 *
 * a gpio unico (optimizar accesos)
 */

/**
 * 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 un entero (bool) con el valor de los bits indicados.
 */
uint32_t hal_gpio_leer(HAL_GPIO_PIN_T gpio){
    uint32_t masc = (1UL << gpio);
    return ((IOPIN & masc)!=0); //version anterior
    //return ((IOPIN & masc)==0);
}

/**
 * Escribe en el gpio el valor
 */
void hal_gpio_escribir(HAL_GPIO_PIN_T gpio, uint32_t valor){
    uint32_t masc = (1UL << gpio);

    if ((valor & 0x01) == 0) IOCLR = masc;
    else IOSET = masc;
}

```

hal_consumo_lpc.c

```

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

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

/* inicia el hal de consumo */
void hal_consumo_iniciar(void) {
}

/* pone al procesador en estado de espera para reducir su consumo */
void hal_consumo_esperar(void) {
    EXTWAKE = 7; /* EXTINT0,1,2 will awake the processor */
    PCON |= 0x01;

}

/* duerme al procesador para minimizar su consumo */

```

```

void hal_consumo_dormir(void) {
    EXTWAKE = 7;           // EXTINT0,1,2 will awake the processor
    PCON |= 0x02;
    switch_to_PLL();      //PLL aranca a 12Mhz cuando volvemos de power down
    ?????????????
}

```

hal_ext_int_lpc.c

```

#include "hal_ext_int.h"
#include "board_lpc.h"
#include "drv_botones.h"
#include <LPC210x.h>

```

```

static void (*ext_int_callback)(uint32_t pin);
//ext_int_callback = NULL;
//static EVENTO_T Id_evento_cb;
//static uint32_t auxData_cb;

```

```

void eint0_ISR(void) __irq {
    hal_ext_int_deshabilitar_int(14);
    EXTINT = 0x07; // Activa los bits 0, 1 y 2
    ext_int_callback(16);
    VICVectAddr = 0;
}

```

```

void eint1_ISR(void) __irq {
    hal_ext_int_deshabilitar_int(15);
    EXTINT = 0x07; // Activa los bits 0, 1 y 2
    ext_int_callback(14);
    VICVectAddr = 0;
}

```

```

void eint2_ISR(void) __irq {
    hal_ext_int_deshabilitar_int(16);
    EXTINT = 0x07; // Activa los bits 0, 1 y 2
    ext_int_callback(15);
    VICVectAddr = 0;
}

```

```

void hal_ext_int_iniciar(uint32_t pin, uint32_t auxData, void (*callback)(uint32_t
auxData)) {
    ext_int_callback = callback;
}

```

```

    if (pin < 16){
        PINSEL0 = PINSEL0 | (0x02 << pin * 2);
        IOCLR = (1 << pin);
    } else {
        PINSEL1 = PINSEL1 | 0x01 << (pin - 16);
        IOCLR = (1 << pin);
    }
    switch(pin){
        case 14:
            VICVectAddr3 = (unsigned long)eint1_ISR; // Establece la
dirección de la ISR para EINT1
            VICVectCntl3 = 0x20 | 15;           // Habilita el slot IRQ
vectoreado para EINT1 (número 15)
            VICIntEnable |= (1 << 15);         // Habilita la
interrupción EINT1 en el VIC
            break;
        case 15:
            VICVectAddr2 = (unsigned long)eint2_ISR; // Establece la
dirección de la ISR para EINT2
            VICVectCntl2 = 0x20 | 16;           // Habilita el slot IRQ
vectoreado para EINT1 (número 16)
            VICIntEnable |= (1 << 16);         // Habilita la
interrupción EINT2 en el VIC
            break;
        case 16:
            VICVectAddr4 = (unsigned long)eint0_ISR; // Establece la
dirección de la ISR para EINT0
            VICVectCntl4 = 0x20 | 14;           // Habilita el slot IRQ
vectoreado para EINT0 (número 14)
            VICIntEnable |= (1 << 14);         // Habilita la
interrupción EINT0 en el VIC
            break;
        default:
            break;
    }
}

void hal_ext_int_habilitar_int(uint32_t pin) {
    EXTINT = 0x07;
    if (pin == 16){
        VICIntEnable = (1 << 14); // Habilita EINT1 (Interrupción
externa 0)
    } else if (pin == 15 || pin == 14){

```

```

        VICIntEnable = (1 << (pin + 1)); // Habilita EINT1 (Interrupción
externa 1, 2)
    } else {
        VICIntEnable = (1 << pin);
    }
}

void hal_ext_int_deshabilitar_int(uint32_t pin) {

    VICIntEnClr = (1 << pin);
}

void hal_ext_int_habilitar_despertar(uint32_t pin) {
    EXTWAKE |= (1 << pin);
}

void hal_ext_int_deshabilitar_despertar(uint32_t pin) {
    EXTWAKE &= ~(1 << pin);
}

void hal_ext_int_limpiarINT() {
    EXTINT = 0x0F;
}

uint32_t hal_ext_int_leerINT(uint32_t pin) {
    uint32_t estado = EXTINT; // Leer el registro EXTINT
    if (pin == 16){
        if(estado & (1 << 0)){
            return 1;
        } else {
            return 0;
        }
    } else if(pin == 14){
        if(estado & (1 << 1)){
            return 1;
        } else {
            return 0;
        }
    } else if(pin == 15){
        if(estado & (1 << 2)){
            return 1;
        } else {
            return 0;
        }
    }
}

```



```

        } else {
            return 0;
        }
    }
// Manejador de la interrupción para EINT1

```

hal_WDT_lpc.c

```

#include "ifdebug.h"
#include <LPC210x.H> /* LPC210x definitions */
#include "hal_WDT.h"
#include "concurrency.h"

void hal_WDT_iniciar(uint32_t sec) {
    WDTC = sec * (15000000 / 4); // time-out WatchDog.
    WDMOD = 0x03; // Habilito y configuro reinicio.
    hal_WDT_feed();
}

void hal_WDT_feed(void) { //ojo irq();
    SC_entrar_disable_irq();
    WDFEED = 0xAA; // Alimento el WatchDog
    WDFEED = 0x55;
    SC_salir_enable_irq();
}

```

hal_concurrency_lpc.c

```

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "hal_concurrency.h"

static uint32_t m_in_critical_region = 0;
//static uint32_t cpsr;

/*extern void __Disable_irq(void);
extern void __Enable_irq(void);*/

uint32_t SC_entrar_disable_irq(void) {
    if (m_in_critical_region == 0) {

```

```

        __disable_irq(); // Desactiva interrupciones solo si no está en una sección
crítica
    }

    return(m_in_critical_region++);
}

void SC_salir_enable_irq(void) {

    if (m_in_critical_region > 0) {
        m_in_critical_region--;
        if (m_in_critical_region == 0) {
            __enable_irq(); // Restaura el estado previo solo si ya no hay secciones
críticas activas
        }
    }
}

```

hal_gpio_nrf.c

```

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

#include "hal_gpio_nrf.h"
#include <nrf.h>
#include "stdint.h"
#include "hal_gpio.h"
#include "board_nrf52840dk.h"

/**
* 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){
    NRF_GPIO->DIR = 0;
}

/* *****
* Acceso a los GPIOs
*

```

```

* optimizado para campos/datos de mas de un bit
* gpio_inicial indica el primer bit con el que operar.
* num_bits indica cuántos bits con los que queremos operar.
*/

/**
* Los bits indicados se configuran como
* entrada o salida según la dirección.

*/
void hal_gpio_sentido_n(uint32_t gpio_inicial,
                        uint8_t num_bits, hal_gpio_pin_dir_t direccion){

    uint32_t i;
    for (i = 0; i < num_bits; i++) {
        uint32_t pin = gpio_inicial + i;

        if (direccion == HAL_GPIO_PIN_DIR_INPUT) {
            // Configura el pin como entrada sin pull (equivalente a nrf_gpio_cfg_input)
            NRF_GPIO->PIN_CNF[pin] = (GPIO_PIN_CNF_DIR_Input <<
GPIO_PIN_CNF_DIR_Pos) |
(GPIO_PIN_CNF_PULL_Disabled <<
GPIO_PIN_CNF_PULL_Pos) |
(GPIO_PIN_CNF_INPUT_Connect <<
GPIO_PIN_CNF_INPUT_Pos) |
(GPIO_PIN_CNF_SENSE_Disabled <<
GPIO_PIN_CNF_SENSE_Pos) |

(GPIO_PIN_CNF_DRIVE_S0S1 <<
GPIO_PIN_CNF_DRIVE_Pos) |

//(GPIO_PIN_CNF_SENSE_Low <<
GPIO_PIN_CNF_SENSE_Pos) |

(GPIO_PIN_CNF_PULL_Pullup <<
GPIO_PIN_CNF_PULL_Pos);
        } else if (direccion == HAL_GPIO_PIN_DIR_OUTPUT) {
            // Configura el pin como salida (equivalente a nrf_gpio_cfg_output)
            NRF_GPIO->PIN_CNF[pin] = (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);
    }
}
}

/**
 * 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(uint32_t gpio_inicial, uint8_t num_bits) {
    uint32_t valor_pines = 0;
    uint32_t i;

    // Iterar sobre el número de bits que se deben leer
    for (i = 0; i < num_bits; i++) {
        uint32_t pin = gpio_inicial + i;

        // Leer el valor del pin actual y ajustar la posición en valor_pines
        if (NRF_GPIO->IN & (1 << pin)) {
            valor_pines |= (1 << i); // Si el pin está en alto (1), ajusta el bit
correspondiente
        }
    }

    return valor_pines;
}

```

```

/**
 * 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(uint32_t gpio_inicial, uint8_t num_bits, uint32_t valor) {
    uint32_t i;

    for (i = 0; i < num_bits; i++) {
        uint32_t pin = gpio_inicial + i;

```

```

        // Verificar el valor del bit correspondiente en 'valor'
        if (valor & (1 << i)) {
            NRF_GPIO->OUTSET = (1 << pin); // Si el bit está en 1, pone el pin en alto
        } else {
            NRF_GPIO->OUTCLR = (1 << pin); // Si el bit está en 0, pone el pin en bajo
        }
    }
}

/* *****
* Acceso a los GPIOs
*
* a gpio unico (optimizar accesos)
*/

/**
* 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){
    if (direccion == HAL_GPIO_PIN_DIR_INPUT) {
        // Configura el pin como entrada
        NRF_GPIO->PIN_CNF[gpio] = (GPIO_PIN_CNF_DIR_Input <<
GPIO_PIN_CNF_DIR_Pos) |
            (GPIO_PIN_CNF_PULL_Disabled <<
GPIO_PIN_CNF_PULL_Pos) |
            (GPIO_PIN_CNF_INPUT_Connect <<
GPIO_PIN_CNF_INPUT_Pos) |
            (GPIO_PIN_CNF_SENSE_Disabled <<
GPIO_PIN_CNF_SENSE_Pos);
    } else if (direccion == HAL_GPIO_PIN_DIR_OUTPUT) {
        // Configura el pin como salida
        NRF_GPIO->PIN_CNF[gpio] = (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);
    }
}

```

```

/**
 * La función devuelve un entero (bool) con el valor de los bits indicados.
 */
uint32_t hal_gpio_leer(HAL_GPIO_PIN_T gpio){
    uint32_t x = (NRF_GPIO->IN >> gpio) & 0x01;
    return x ;// Desplazar y enmascarar para obtener el valor del pin específico
}

/**
 * Escribe en el gpio el valor
 */
void hal_gpio_escribir(HAL_GPIO_PIN_T gpio, uint32_t valor){
    if (valor & 0x1) {
        NRF_GPIO->OUTSET = (1UL << gpio); // Si el bit está en 1, pone el pin en
alto
    } else {
        NRF_GPIO->OUTCLR = (1UL << gpio); // Si el bit está en 0, pone el pin en
bajo
    }
}

```

hal_tiempo_nrf.c

```

/* *****
 * P.H.2024: TODO
 * implementacion para cumplir el hal_tiempo.h
 */

#include <nrf.h>
#include <stdint.h>

#define MAX_COUNTER_VALUE 0xFFFFFFFF // Maximo valor
del contador de 32 bits
#define HAL_TICKS2US 16
// funcionamos PCLK a 16 MHz
// #define US2MS 1000
//milisegundos por microsogundos

/* *****
 * Timer0 contador de ticks
 */

```

```
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
*/
void TIMER0_IRQHandler (void){
    volatile uint32_t dummy;
    if (NRF_TIMER0->EVENTS_COMPARE[0] == 1)
    {
        NRF_TIMER0->EVENTS_COMPARE[0] = 0;

        // Read back event register so ensure we have cleared it before exiting IRQ
        handler.
        dummy = NRF_TIMER0->EVENTS_COMPARE[0];
        dummy; // to get rid of set but not used warning
    }

    timer0_int_count++;
    // Limpiar la interrupción pendiente en el NVIC
    NVIC_ClearPendingIRQ(TIMER0_IRQn);          // Acknowledge Interrupt
}

/* *****
* Programa un contador de tick sobre Timer0, con maxima precisión y minimas
interrupciones
*/
uint32_t hal_tiempo_iniciar_tick() {
    // Reiniciar el contador de interrupciones del Timer 0
    timer0_int_count = 0;

    // Apagar el temporizador antes de configurarlo
    NRF_TIMER0->TASKS_STOP = 1;

    // Configurar el temporizador en modo TIMER (no en modo COUNTER)
    NRF_TIMER0->MODE = TIMER_MODE_MODE_Timer; // 0 = Timer Mode

    // Configurar el temporizador con una resolución de 32 bits
    NRF_TIMER0->BITMODE = TIMER_BITMODE_BITMODE_32Bit <<
    TIMER_BITMODE_BITMODE_Pos;

    // Configurar la frecuencia del temporizador (ejemplo: 1 MHz = 1 tick por
    microsegundo)
```

```

NRF_TIMER0->PRESCALER = 0; // 1 MHz (16 MHz / 2^0 = 16 MHz)

// Establecer el valor de comparación en el registro CC[0]
NRF_TIMER0->CC[0] = MAX_COUNTER_VALUE;

// Configurar el evento de comparación (Match Compare) para reiniciar y generar
interrupción
NRF_TIMER0->SHORTS = TIMER_SHORTS_COMPARE0_CLEAR_Enabled <<
TIMER_SHORTS_COMPARE0_CLEAR_Pos;
NRF_TIMER0->INTENSET = TIMER_INTENSET_COMPARE0_Enabled <<
TIMER_INTENSET_COMPARE0_Pos;

// Limpiar cualquier evento de comparación previo
NRF_TIMER0->EVENTS_COMPARE[0] = 0;

// Habilitar la interrupción del Timer 0 en el NVIC
NVIC_EnableIRQ(TIMER0_IRQn);

// Reiniciar y luego comenzar el temporizador
NRF_TIMER0->TASKS_CLEAR = 1;
NRF_TIMER0->TASKS_START = 1;

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;
    // Obtener el valor actual del contador del Timer 0
    NRF_TIMER0->TASKS_CAPTURE[1] = 1;
    uint32_t contador_actual = NRF_TIMER0->CC[1]; // Usamos el registro de
comparación CC[1] como referencia

    // Combinar las interrupciones y el valor actual del contador
    time = ((uint64_t)(MAX_COUNTER_VALUE) * (uint64_t)timer0_int_count) +
(uint64_t)contador_actual;

    return time;
}

```



```

/* *****
* Activacion periodica con timer 1
*/
static void(*f_callback)();          //puntero a funcion a llamar cuando salte la RSI
(en modo irq)

/* *****
* Timer 1 Interrupt Service Routine
* llama a la funcion de callbak que se ejecutara en modo irq
*/
void TIMER1_IRQHandler(void) {
    volatile uint32_t dummy;
    if (NRF_TIMER1->EVENTS_COMPARE[0] != 0) { // Verifica si el evento de
comparación ocurrió
        NRF_TIMER1->EVENTS_COMPARE[0] = 0;    // Limpia el evento de
comparación
        dummy = NRF_TIMER1->EVENTS_COMPARE[0];
        dummy; // to get rid of set but not used warning

        f_callback(); // Llama a la función callback
    }
}

/* *****
* Dependiente del hardware Timer1
* 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)()){

    f_callback = funcion_callback;

    if (periodo_en_tick != 0) { //Si el periodo es cero solo se para el temporizador.
        // Apagar el temporizador antes de configurar
        NRF_TIMER1->TASKS_STOP = 1;

        // Resetear el contador
        NRF_TIMER1->TASKS_CLEAR = 1;

        // Configurar el modo de temporizador
        NRF_TIMER1->MODE = TIMER_MODE_MODE_Timer; // Modo Timer

```

```

    NRF_TIMER1->BITMODE = TIMER_BITMODE_BITMODE_32Bit; //
    Temporizador de 32 bits

    // Configurar la prescaler si es necesario (para ajustarse a la frecuencia
    deseada)
    // NRF_TIMER1->PRESCALER = 4; // Ejemplo de configuración de prescaler
    (usualmente 16 MHz / 2^(prescaler))

    // Configurar el valor de comparación (periodo en ticks)
    NRF_TIMER1->CC[0] = periodo_en_tick ;
    NRF_TIMER1->PRESCALER = 0;

    // Configurar para generar interrupciones en el evento de comparación
    NRF_TIMER1->SHORTS = TIMER_SHORTS_COMPARE0_CLEAR_Enabled;
    // Reinicia el contador al alcanzar CC[0]
    NRF_TIMER1->INTENSET = TIMER_INTENSET_COMPARE0_Msk; //
    Habilitar interrupciones en CC[0]

    // Habilitar la interrupción en el NVIC

    NVIC_EnableIRQ(TIMER1_IRQn);

    // Iniciar el temporizador
    NRF_TIMER1->TASKS_CLEAR = 1;
    NRF_TIMER1->TASKS_START = 1;
    } else {
        // Detiene el temporizador
        NRF_TIMER1->TASKS_STOP = 1;
        // NRF_TIMER1->INTENCLR =
    TIMER_INTENSET_COMPARE0_Msk;
    //NVIC_DisableIRQ(TIMER1_IRQn);
    }
}

```

hal_consumo_nrf.c

```

#include "hal_consumo_nrf.h"
#include <nrf.h>

/* *****
* Inicialización del hardware relacionado con el consumo
* En esta plataforma, no es necesaria una inicialización específica.
*/
void hal_consumo_iniciar(void) {

```

```

    // No se requiere inicialización específica por ahora
}

/* *****
* Poner el procesador en modo de espera utilizando WFI
*/
void hal_consumo_esperar(void) {
    // Instrucción CMSIS para poner el procesador en modo de espera hasta la
    // próxima interrupción
    __WFI();
}

/* *****
* Poner el procesador en modo de dormir (función pendiente de implementación)
* Este modo aún no es necesario en la plataforma nRF, se implementará en el
futuro.
*/
void hal_consumo_dormir(void) {
    NRF_POWER->SYSTEMOFF =
    POWER_SYSTEMOFF_SYSTEMOFF_Enter;
    while(1){
        __WFE();
    }
}

```

hal_ext_int_nrf.c

```

#include "hal_ext_int.h"
#include <nrf.h>           // Funciones específicas del microcontrolador nRF
#include "board.h"

//static void (*ext_int_callback)(EVENTO_T ID_evento, uint32_t auxData); // Puntero
//a la función de callback
//static EVENTO_T Id_evento_cb;
//static uint32_t pin_cb;
static uint32_t channel = 0; // Canal GPIOTE (0-7)

void (*callbacks[BUTTONS_NUMBER])(uint32_t auxData) = {};

typedef struct {
    uint32_t auxData;
    //uint32_t pin;
    uint32_t canal_pin;
}

```

```

    void (*callback)(uint32_t auxData);
        //void (*callback)(uint32_t pin);
} BotonCallbackInfo;

BotonCallbackInfo boton_callbacks[BUTTONS_NUMBER]; // Información de
callbacks por botón

// Función para inicializar los botones con GPIOTE
void hal_ext_int_iniciar(uint32_t pin, uint32_t auxData, void (*callback)(uint32_t
auxData)) {
    //static uint32_t channel = 0; // Canal GPIOTE (0-7)
    if (channel >= 8) {
        // No hay canales disponibles, manejar error aquí
        return;
    }

    NRF_GPIOTE->CONFIG[channel] = 0;
    NRF_GPIOTE->INTENCLR = (1 << channel);

    // Configurar el canal GPIOTE para detectar cambios en el pin
    NRF_GPIOTE->CONFIG[channel] = (GPIOTE_CONFIG_MODE_Event <<
GPIOTE_CONFIG_MODE_Pos) |
        (pin << GPIOTE_CONFIG_PSEL_Pos) |
        (GPIOTE_CONFIG_POLARITY_HiToLo <<
GPIOTE_CONFIG_POLARITY_Pos);

    // Habilitar la interrupción del canal GPIOT

    // Almacenar los datos del callback
    boton_callbacks[channel].auxData = pin;
    //boton_callbacks[channel].pin = pin;
        boton_callbacks[channel].canal_pin = channel;
        //habria que pasar el pin del boton que estamos pulsandi
    boton_callbacks[channel].callback = callback;
        hal_ext_int_habilitar_int(pin);

    // Incrementar el canal para el siguiente botón
    channel++;
}

// Manejador de interrupciones de GPIOTE
void GPIOTE_IRQHandler(void) {
    for (uint32_t channel = 0; channel < BUTTONS_NUMBER; channel++) {
        if (NRF_GPIOTE->EVENTS_IN[channel]) {
            NRF_GPIOTE->EVENTS_IN[channel] = 0; // Limpiar el evento

```

```

        // Llamar al callback asociado, si está definido
        if (boton_callbacks[channel].callback) {
            boton_callbacks[channel].callback(boton_callbacks[channel].auxData);

//boton_callbacks[channel].callback(boton_callbacks[channel].pin);

        hal_ext_int_deshabilitar_int(boton_callbacks[channel].auxData);

//hal_ext_int_deshabilitar_int(boton_callbacks[channel].pin);
    }
}
}

// Función para habilitar la interrupción externa para un pin específico
void hal_ext_int_habilitar_int(uint32_t channel) {

    for (uint32_t i = 0; i<BUTTONS_NUMBER; i++){
        if(boton_callbacks[i].auxData == channel){
            //if(boton_callbacks[i].pin == channel){

NRF_GPIOTE->EVENTS_IN[boton_callbacks[i].canal_pin] = 0; // Limpiar el evento
            uint32_t dummy =
NRF_GPIOTE->EVENTS_IN[boton_callbacks[i].canal_pin];

            // Habilitar la interrupción para este canal
            NRF_GPIOTE->INTENSET = (1 << boton_callbacks[i].canal_pin);
            NVIC_EnableIRQ(GPIOTE_IRQn);
        }
    }

    //

}

// Función para deshabilitar la interrupción externa para un pin específico
void hal_ext_int_deshabilitar_int(uint32_t channel) {
    NRF_GPIOTE->INTENCLR = (1 << channel);
    NVIC_DisableIRQ(GPIOTE_IRQn);

    /*for (uint32_t i = 0; i<BUTTONS_NUMBER; i++){
        if(boton_callbacks[i].auxData == channel){
            //if(boton_callbacks[i].pin == channel){

```

```

NRF_GPIOTE->EVENTS_IN[boton_callbacks[i].canal_pin] = 0; // Limpiar el evento
uint32_t dummy =
NRF_GPIOTE->EVENTS_IN[boton_callbacks[i].canal_pin];

    // Habilitar la interrupción para este canal
    NRF_GPIOTE->INTENCLR = (1 <<
boton_callbacks[i].canal_pin);
    NVIC_DisableIRQ(GPIOTE_IRQn);
    }
    */
}

// Función para configurar el pin para despertar del modo de bajo consumo
void hal_ext_int_habilitar_despertar(uint32_t pin) {
    // Configurar el pin para interrupciones externas (nivel bajo por defecto)
    NRF_GPIO->PIN_CNF[pin] = (GPIO_PIN_CNF_DRIVE_S0S1 <<
GPIO_PIN_CNF_DRIVE_Pos) |

    (GPIO_PIN_CNF_SENSE_Low << GPIO_PIN_CNF_SENSE_Pos) |

    (GPIO_PIN_CNF_PULL_Pullup << GPIO_PIN_CNF_PULL_Pos);
}

// Función para preparar el dispositivo para entrar en modo de bajo consumo
void hal_ext_int_preparar_dormir() {
    // Configura el sistema para entrar en modo de apagado (sueño profundo)
    NRF_POWER->SYSTEMOFF = 1; // Ordena al sistema entrar en modo de
apagado
}

// Función para despertar el sistema del modo de bajo consumo
void hal_ext_int_despertar() {
    // Despierta el sistema del sueño profundo
    NRF_POWER->SYSTEMOFF = 0; // Despierta el sistema
}

uint32_t hal_ext_int_leerINT(uint32_t gpio)
{
    uint32_t x = (NRF_GPIO->IN >> gpio) & 0x01;
    return x ;// Desplazar y enmascarar para obtener el valor del pin específico
}

```

```
}
```

hal_WDT_nrf.c

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

```
#include "hal_WDT.h"
```

```
void hal_WDT_iniciar(uint32_t sec) {  
    //NRF_WDT->TASKS_STOP = 1;
```

```
    NRF_WDT->CRV = 32768 * sec;  
    NRF_WDT->RREN |= WDT_RREN_RR0_Msk;
```

```
    // Configurar el comportamiento del WDT (reinicio del sistema cuando expire)  
    NRF_WDT->CONFIG = (WDT_CONFIG_HALT_Pause <<  
WDT_CONFIG_HALT_Pos) | ( WDT_CONFIG_SLEEP_Run <<  
WDT_CONFIG_SLEEP_Pos);
```

```
    NRF_WDT->TASKS_START = 1;  
    hal_WDT_feed();
```

```
}
```

```
void hal_WDT_feed(void) {  
    NRF_WDT->RR[0] = WDT_RR_RR_Reload;  
    NRF_WDT->RR[1] = WDT_RR_RR_Reload;  
}
```

hal_concurrencia_nrf.c

```
#include <stdint.h>
```

```
#include <stdbool.h>
```

```
#include <string.h>
```

```
#include "hal_concurrencia.h"
```

```
static uint32_t m_in_critical_region = 0;  
static uint32_t cpsr;
```

```
uint32_t SC_entrar_disable_irq(void) {  
    if (m_in_critical_region == 0) {  
        __disable_irq(); // Desactiva interrupciones solo si no est? en una secci?n  
cr?tica
```

```

    }
    return(m_in_critical_region++);
}

```

```

void SC_salir_enable_irq(void) {

    if (m_in_critical_region > 0) {
        m_in_critical_region--;
        if (m_in_critical_region == 0) {
            __enable_irq(); // Restaura el estado previo solo si ya no hay secciones
cr?ticas activas
        }
    }
}

```