Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

ph-pr-3

Proyecto Hardware

Autor 1:	Toral Pallás, Héctor - 798095
Autor 2:	Lahoz Bernad, Fernando - 800989
Grado:	Ingeniería Informática
Curso:	2022-2023

12 de Enero de 2023

Resumen

El siguiente documento recoge las ideas de un proyecto de diseño e implementación de un sistema capaz de ejecutar el juego Conecta 4 en un procesador ARM con microcontrolador LPC2105. Este proyecto ha sido realizado en el ámbito de la asignatura Proyecto Hardware, del grado de Ingeniería Informática impartido por la Universidad de Zaragoza. El objetivo era dar una solución a los requisitos de entrada y salida que puede llegar a tener cualquier sistema informático convencional y emplear el control sobre los periféricos para introducir interacción con el usuario al juego.

El entorno de depuración y ejecución utilizado ha sido el simulador Keil μVison versión 5. El código del proyecto está escrito en lenguaje C y ensamblador, haciendo uso exclusivo de la biblioteca «inttypes», para asegurar la representación esperada de los datos enteros, y el fichero «LPC210x.H», proporcionado por el compilador de Keil, que ofrece un conjunto de macros que facilitan el acceso a los registros de los periféricos sin sacrificar la legibilidad del código. Pese a que el simulador ofrece un editor de código, el entorno de desarrollo utilizado ha sido Visual Studio Code, el cual permite la edición conjunta de ficheros en remoto.

Pese a utilizar un lenguaje de programación imperativo, el funcionamiento del sistema ha sido diseñado aplicando programación orientada a eventos. Para ello se ha planteado un sistema modular en el que los principales módulos se comunican por medio de colas y la ejecución es controlada en todo momento por un planificador. Con esto se ha conseguido tratar tanto los mensajes que intercambian los módulos entre sí, como los eventos asíncronos generados por los periféricos. Una de las dificultades que implica es el hecho de tener que lidiar con condiciones de carrera provocadas por tratar estos eventos de forma concurrente.

El sistema cuenta con el gestor de entrada/salida y el gestor de serie para mostrar la información de juego al usuario a través de leds conectados a los pines del GPIO y la línea de serie utilizada como terminal (también usada como entrada de comandos). Un gestor de botones controla los eventos de pulsación, otro comprueba la situación para disminuir el uso energético. Las estadísticas de calidad de servicio también son controladas por gestor. Gracias al módulo de alarmas los gestores pueden enviar mensajes con retardo, y cancelarlos antes de su envío si es preciso. La lógica y las estructuras de memoria del juego están completamente aislados en un módulo, que también hace uso de las colas para recibir y enviar información del juego.

Para el manejo de los periféricos y su configuración al comienzo de una ejecución se han implementado varios módulos de bajo nivel, que interactúan directamente con los registros de los propios periféricos. En concreto, se ha desarrollado una interfaz de uso de los temporizadores 0 y 1, los contadores Real Time Clock (RTC) y Watchdog, la línea de serie UART0 y el puerto de entrada y salida del GPIO, además de unos módulos de configuración de las interrupciones externas y los modos de consumo del procesador. El tratamiento de los eventos producidos por los periféricos de forma asíncrona ha permitido eliminar bucles de espera activa y con ello introducir los modos de bajo consumo del procesador cuando no tiene la necesidad de ejecutar instrucciones. El resultado es que el procesador se encuentra detenido la mayor parte del tiempo, por lo que el ahorro energético es máximo.

Índice

1. Introducción

En este proyecto, se ha llevado a cabo la implementación del juego de Conecta 4 para el microcontrolador de ARM LPC2105.En una versión anterior se implementó y optimizó la lógica de comprobación del tablero. Esta versión incluye el desarrollo de una serie de módulos que permiten conectar los recursos hardware para la entrada y salida con nuestro juego, de manera que pueda ser jugado a través de los periféricos. Todo ello tratando de aprovechar al máximo las opciones del procesador disponibles para mejorar la experiencia de juego así como la eficiencia energética, que como se puede apreciar a día de hoy es un recurso cada vez más costoso.

El proyecto tiene implícito un objetivo de aprendizaje mezclando la modularización completa de las funcionalidades y la implementación de la lógica del sistema diseñada con programación orientada a eventos.

2. Comunicación del sistema

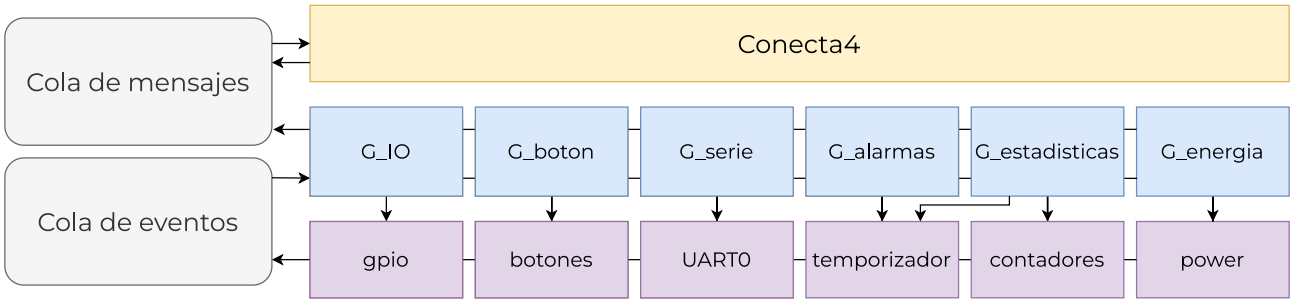

Figura 1: Estructura de comunicación del sistema.

El sistema está compuesto por un conjunto de módulos que gestionan las operaciones de entrada y salida. Cada uno hace uso de las funciones que le ofrece la interfaz del periférico que gestiona. De esta forma, el gestor puede funcionar de manera independiente del hardware utilizado y la implementación, siempre y cuando la interfaz ofrezca las operaciones necesarias. Además, existe un módulo dedicado exclusivamente a la lógica del juego.

La comunicación entre los gestores se realiza a través de una cola de mensajes. Esto significa que ninguno de ellos llama directamente a las funciones y procedimientos de los demás. En su lugar, es necesario declarar un mensaje que contenga la información necesaria para que el gestor al que va dirigido pueda reaccionar de manera adecuada. El mismo protocolo se utiliza para la comunicación entre los gestores y el módulo del juego. Los periféricos pueden generar interrupciones para reemplazar las esperas activas por eventos asíncronos. Estos eventos deben comunicarse al gestor correspondiente a través de una cola de eventos. Tanto los mensajes como los eventos están identificados con una constante entera, declaradas en los archivos «msg.h» y «eventos.h». La figura 1 muestra la estructura de comunicación descrita.

2.1. Planificador

El funcionamiento del programa principal consiste en inicializar todos los gestores y la lógica del juego, y luego ejecutar un bucle infinito que actúa como planificador. En cada iteración del bucle, se comprueba si hay un evento o mensaje disponibles en las colas. En ese caso, se transmiten el mensaje y/o evento a los diferentes módulos para que realicen la acción correspondiente. Para ello, tanto los gestores como el juego proporcionan tres funciones al planificador: iniciar, tratar mensaje y tratar evento. Para evitar que el programa se convierta en un bucle de espera activa, cada vez que no detecta eventos o mensajes en las colas, debe detener el avance del programa llamando a la función «idle» del gestor de energía (sección 5).

2.2. Colas de eventos y mensajes

El sistema utiliza dos colas circulares con política FIFO como estructuras de datos. Las operaciones que ofrecen son tres: encolar, desencolar y hay eventos/mensajes. Cada una tiene un tamaño fijo. Si se supera este tamaño al encolar una nueva entrada, se produce un evento o mensaje de overflow que se coloca en la primera posición para ser la próxima entrada a desencolar. Tanto el evento como el mensaje de overflow provocarán que el programa entre en un bucle infinito.

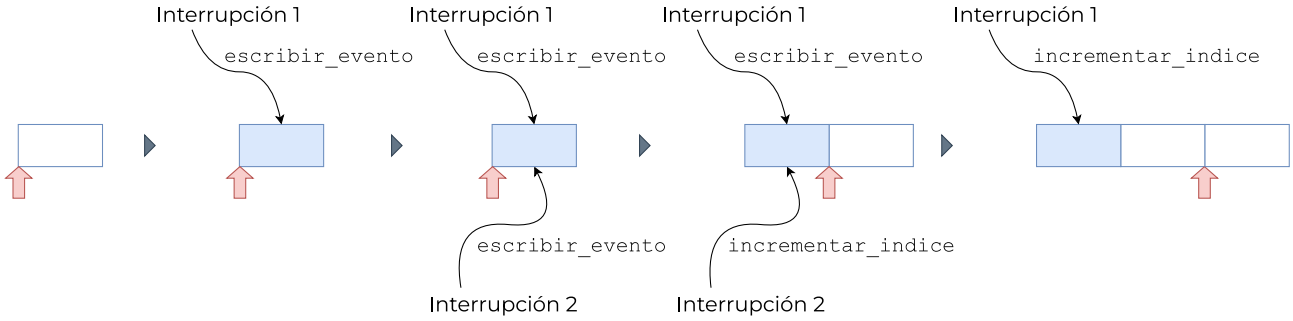

Figura 2: Condición de carrera producida cuando un periférico interrumpe la acción de encolar evento.

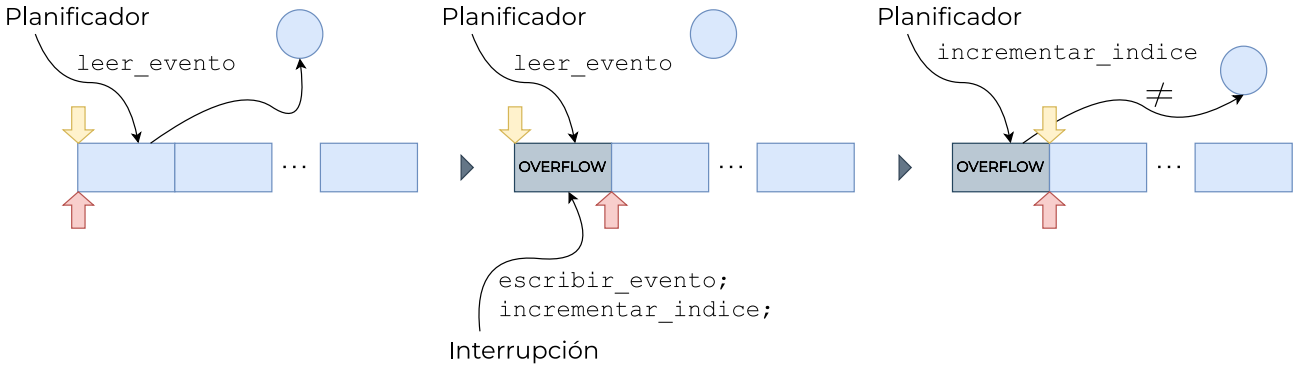

Figura 3: Condición de carrera producida cuando un periférico interrumpe la acción de desencolar evento en situación de overflow.

La cola de mensajes tiene un tamaño que permite almacenar 16 mensajes. La información disponible para cada mensaje incluye su identificador, una marca de tiempo en microsegundos de cuándo fue encolado y un campo auxiliar con información adicional. El tamaño de estos campos es de 8, 32 y 32 bits, respectivamente. Para optimizar el espacio utilizado por la cola y evitar que el compilador alinee el identificador, cada uno de los campos se almacena en un array por separado.

La cola de eventos almacena los últimos 32 eventos asíncronos generados por los periféricos. La información del evento que se almacena incluye su identificador, el número de veces que ha sido generado y un campo auxiliar con información adicional. El tamaño de estos campos es de 8, 32 y 32 bits, respectivamente, y, al igual que en la cola de mensajes, cada uno de los campos se almacena en un array por separado para evitar el alineamiento.

A diferencia de la cola de mensajes, la cola de eventos puede provocar comportamientos inesperados debido a que la acción de encolar se ejecuta al producirse una interrupción. Para evitar posibles condiciones de carrera, se ha implementado el módulo «semaforo_interrupciones» que permite bloquear las interrupciones a través de llamadas al sistema. Hay dos secciones de código que deben estar rodeadas por el bloqueo y liberación de las interrupciones:

- **Encolar un evento:** Si se produce una interrupción en algún momento mientras se está encolando un evento, la entrada correspondiente a ese evento puede ser sobrescrita por el evento entrante. Esta acción podría provocar que el índice de la cola aumente dos veces escribiendo en una única entrada, generando un evento indefinido (Ver 2).
- **Desencolar un evento:** Cuando la cola está llena, el primer y último índice coinciden. Si al desencolar un evento, justo después de leer el evento y antes de incrementar el primer índice, llega una interrupción, se genera un evento de overflow en la primera posición que debe ser leído antes que cualquier otro. Sin embargo, al volver a la operación de desencolar el evento de la posición sobrescrita ya ha sido leído, y ahora sería necesario leer todos los eventos anteriores al de overflow para poder gestionarlo (Ver 3).

2.3. Llamadas al sistema y semáforo de interrupciones

Para este proyecto ha sido necesario modificar el modo de ejecución del planificador para que el programa principal se ejecute en modo usuario. Como consecuencia, hay operaciones con cierto nivel de privilegios que no pueden ser ejecutadas a menos que sean implementadas como llamadas al sistema y estas se ejecuten en modo superusuario. Las operaciones que se ofrecen como llamadas al sistema incluyen leer y alterar los «flags» de interrupción del registro de estado y son necesarias para crear un semáforo que permita bloquear y desbloquear las interrupciones. Además, se incluyen las funciones «clock gettime» y «clocl get us» habilitadas para leer los valores de los registros contador de los relojes RTC y TIMER1, respectivamente.

Las llamadas al sistema están implementadas en ensamblador, en el archivo (SWI.s), junto con la rutina (SWI Handler). Esta se encarga de derivar la ejecución a la subrutina correspondiente según su número. Las llamadas al sistema con los números más bajos (0-3) se acceden a través de una tabla de traducción (un array), mientras que las últimas se decodifican mediante comprobaciones secuenciales. Los códigos y sus correspondencias se presentan en la siguiente tabla:

Código	**Función**	**Descripción**
0x00	clock_gettime	Devuelve el registro CTIME0.
0x01	clock_get_us	Devuelve el registro T1TC.
0x02	read_IRQ_bit	Devuelve el bit I del registro de estado.
0x03	read_FIQ_bit	Devuelve el bit F del registro de estado.
0xFA	enable_fiq	Desactiva el bit F del registro de estado.
0xFB	disable_fiq	Activa el bit F del registro de estado.
0xFC	enable_irq_fiq	Desactiva los bits F e I del registro de estado.
0xFD	disable_irq_fiq	Activa los bits F e I del registro de estado.
0xFE	enable_irq	Desactiva el bit I del registro de estado.
0xFF	disable_irq	Activa el bit I del registro de estado.

El semáforo de interrupciones ofrece las operaciones bloquear y liberar. La operación bloquear guarda el estado de las interrupciones leyendo los bits I y F y las desactiva. La operación liberar comprueba el estado previo para habilitar solamente aquellas interrupciones que estuvieran habilitadas. El estado no se almacena en una variable de tipo «static», sino que es devuelto como resultado de la función bloquear, y es necesario llamar a la función liberar con ese mismo valor. El motivo es que puede haber varias llamadas concurrentes a la función bloquear. Si después de haber consultado los «flags» y haberlos guardado en una variable de tipo «static» se produce una interrupción, la misma volverá a llamar a la función bloquear, pero el modo del procesador habrá cambiado y con él los «flags» de interrupción, por lo que el estado anterior se perdería. Si en cambio el estado es retornado como resultado de la función, este será apilado en caso de interrupción y se podrá recuperar.

3. Tiempos

Para poder cumplir con ciertos requisitos del sistemas se ha requerido de la implementación de diferentes módulos para la gestión de tiempos.

3.1. Relojes del procesador

El procesador empleado utiliza un oscilador de cristal para generar las señales de reloj. El sistema encargado de controlar la frecuencia de oscilación es el PLL y se configura en el archivo «Startup.s». La frecuencia de salida de la señal de reloj se denomina CCLK y es la que marca el tiempo por instrucción. En este proyecto se utiliza una frecuencia de procesador de 60 MHz. Los periféricos no funcionan a esta misma frecuencia, sino que utilizan una fracción del CCLK, denominada PCLK. En este caso, el valor de PCLK es 4 veces menor: 15 MHz.

3.2. Temporizadores y Contadores

Estos módulos implementan un conjunto de funciones que permiten trabajar con dos tipos de temporizadores: los de propósito general (Timer 0 y 1) y los de propósito específico (Real Time Clock y Watchdog).

Temporizadores de propósito general

Los temporizadores de propósito general tienen dos tareas asignadas: el timer 1 se encarga de medir con precisión el tiempo transcurrido en microsegundos, mientras que el timer 0 encola un evento de forma periódica según una cantidad indicada de milisegundos. Su configuración se encuentra en el archivo "temporizador.c", donde principalmente se modifica el valor de estos dos registros:

- **PR:** Indica cuántos ciclos de reloj deben transcurrir para que se incremente un contador. En el caso del timer 0, se desea que aumente cada milisegundo, mientras que en el del timer 1, cada microsegundo. Su valor se calcula mediante la fórmula: $PR = PCLK \cdot T - 1$, de modo que los valores utilizados son:

$$T0PR = PCLK \cdot 1ms - 1 = 15 \cdot 10^6 Hz \cdot 1 \cdot 10^{-3} s - 1 = 15000 - 1 = 14999$$

$$T1PR = PCLK \cdot 1\mu s - 1 = 15 \cdot 10^6 Hz \cdot 1 \cdot 10^{-6} s - 1 = 15 - 1 = 14$$

- **MR0:** Indica con cuantos incrementos el contador se debe resetear y, si está configurado para ello, generar una interrupción. En el caso del timer 1, se desea que se resetee lo más tarde posible, es decir, cuando alcance el máximo valor posible de representación con 32 bits, sin generar interrupciones. Por otro lado, con el timer 0, se desea que genere una interrupción en un período especificado con la función «temporizador_reloj».

El valor del contador del timer 1 se puede leer a través de la llamada al sistema «clock_get_us», explicada en la sección 2.3. El contador del timer 0 no es de interés ya que lo usamos para generar eventos asíncronos. Estos eventos son generados en una rutina de interrupción, en la que se encola un evento del tipo «TEMPORIZADOR». Las interrupciones del timer 0 están configuradas como FIQ.

Es importante tener en cuenta el uso de semáforos en las colas, como se menciona en los apartados 2.2 y 2.3, debido a que la función «temporizador leer» también hace uso de llamadas al sistema. Al realizar llamadas al sistema en la función de encolar, el procesador entra en modo supervisor. Si, mientras el procesador está en este modo, el temporizador genera una interrupción, la acción de encolar un evento volvería a ejecutar una llamada al sistema al intentar bloquear las interrupciones. Esto implicaría volver a entrar en modo supervisor, sobrescribiendo los registros previos y corrompiendo la ejecución del programa. Por eso, la rutina de interrupción del timer 0 no puede entrar en modo supervisor. Para solucionar este problema, se ha dividido la función para encolar eventos en dos: una que realice la acción de encolar y otra que la ejecute bloqueando las interrupciones. A partir de ahí, basta con que la rutina de interrupción llame a la versión «cruda».

Temporizadores de propósito específico

Además, hay temporizadores con propósitos específicos, configurados en el archivo «contadores.c».

El Real Time Clock (RTC) es el encargado de medir el tiempo de una partida en minutos y segundos. Se utiliza porque el procesador puede entrar en modo «power down». A diferencia de los timers 0 y 1, el RTC sigue contando incluso cuando el procesador está en este modo. Para que el RTC cuente los segundos correctamente, ha sido necesario configurar los registros PREINT y PREFRAC de acuerdo a las siguientes fórmulas:

$$PREINT = \left\lfloor \frac{PCLK}{32768} \right\rfloor - 1 = \left\lfloor \frac{15 \cdot 10^6}{32768} \right\rfloor - 1 = 456$$

$$PREFRAC = PCLK - ((PREINT + 1) \cdot 32768) = 10^6 - ((456 + 1) \cdot 32768) = 25024$$

Se puede leer los minutos y segundos transcurridos utilizando la función «RTC leer», que hace uso de la llamada al sistema «clock gettime» para obtener el valor del registro CTIME0. Los segundos se extraen de los primeros 6 bits y los minutos de los bits 8 al 13.

El Watchdog tiene la función de reiniciar el sistema en caso de que el programa entre en un bucle infinito (situación de overflow explicada en la sección 2.2. Para detectar estas situaciones, es necesario ((alimentarlo)) antes de que transcurra un cierto número de segundos, establecidos en la función «WD init». Esta tarea es responsabilidad del planificador, que en cada iteración reseteará el contador del Watchdog llamando a la función

«WD feed». Para configurar el contador del Watchdog con un número de segundos «seg», se ha indicado en el registro WDTC el resultado de la siguiente fórmula, basada en la fórmula del cálculo del tiempo indicada en el manual.

$$WDTC \cdot \frac{1}{PCLK} \cdot 4 = sec$$

$$WDTC = sec \cdot PCLK \cdot \frac{1}{4} = sec \cdot 15 \cdot 10^6 \cdot \frac{1}{4} = sec \cdot 3750000$$

La operación de alimentar consiste en escribir los valores 0xAA y 0x55 en el registro WDFEED. Esta acción no debe ser interrumpida, por lo que es necesario bloquear las interrupciones mediante el uso del semáforo mencionado en la sección 2.3.

3.3. Gestor de alarmas

Se ha implementado un gestor de alarmas que permite a los módulos del sistema encolar un mensaje determinado transcurridos una cierta cantidad de microsegundos. Una alarma puede ser creada por cualquier módulo encolando el mensaje «SET ALARM», junto con una variable de tipo «alarma t» que contiene la información necesaria para crear o eliminar la alarma. El módulo proporciona una interfaz para facilitar el manejo de este tipo de datos sin tener que llamar directamente a las funciones que administran las alarmas. Esta interfaz incluye las siguientes funciones:

- **alarma_t g_alarma_crear(uint8_t ID_msg, int esPeriodica, uint32_t retardo):** Genera el campo auxiliar a encolar para programar una alarma, o reprogramar la que ya envía ese mensaje.
- **alarma_t g_alarma_borrar(uint8_t ID_msg):** Genera el campo auxiliar a encolar para eliminar la alarma con el mensaje introducido.
- **uint8_t g_alarma_id_msg(alarma_t alarma):** Devuelve el mensaje programado.
- **int g_alarma_es_periodica(alarma_t alarma):** Comprueba si la alarma es reprogramada una vez finaliza su tiempo.
- **alarma_t g_alarma_retardo(alarma_t alarma):** Devuelve el tiempo en milisegundos que tardará en saltar la alarma.

El gestor controla hasta 8 alarmas, almacenadas en un vector. Al recibir un mensaje «SET ALARM», se verifica si es necesario reprogramarla (o eliminarla en caso de que el retraso sea 0). Si no se ha encontrado ninguna alarma que encole el mismo mensaje, se verifica que el retraso sea mayor que 0 (no se pueden programar alarmas instantáneas) y se busca la primera posición del vector disponible para programarla. Además, en otro vector se almacenan los milisegundos restantes de cada alarma. Al recibir un evento «TEMPORIZADOR», generado por el timer 0 cada milisegundo, se recorre el vector de tiempos, comprobando qué posiciones están habilitadas para decrementar en uno los milisegundos restantes. Si alguna de las componentes llega a 0, el gestor encola el mensaje correspondiente. Si la alarma era periódica, el tiempo de la alarma se resetea a su retraso. Si no, la alarma queda eliminada

3.4. Gestor de estadísticas

Al finalizar una partida, se deben mostrar dos estadísticas: el tiempo de juego en minutos y segundos, y los microsegundos medios transcurridos desde que un mensaje es encolado hasta que es recibido por todos los módulos. Para ello, el gestor de estadísticas acumula los tiempos de gestión de mensajes y el número de mensajes tratados durante la partida, de manera que al finalizarla pueda calcular la media mediante una división. Es importante que el gestor de estadísticas sea el último en tratar el mensaje para obtener una visión más precisa de la calidad de servicio. El tiempo de juego se calcula simplemente como la diferencia entre el tiempo medido al comienzo y al final. La resta debe realizarse teniendo en cuenta que el tiempo está en minutos y segundos, es decir, si la diferencia de segundos es negativa, significa que hay que considerar un minuto menos y sumar a los segundos más recientes los que quedaban para alcanzar el minuto $(s_1 + (60 - s_0))$.

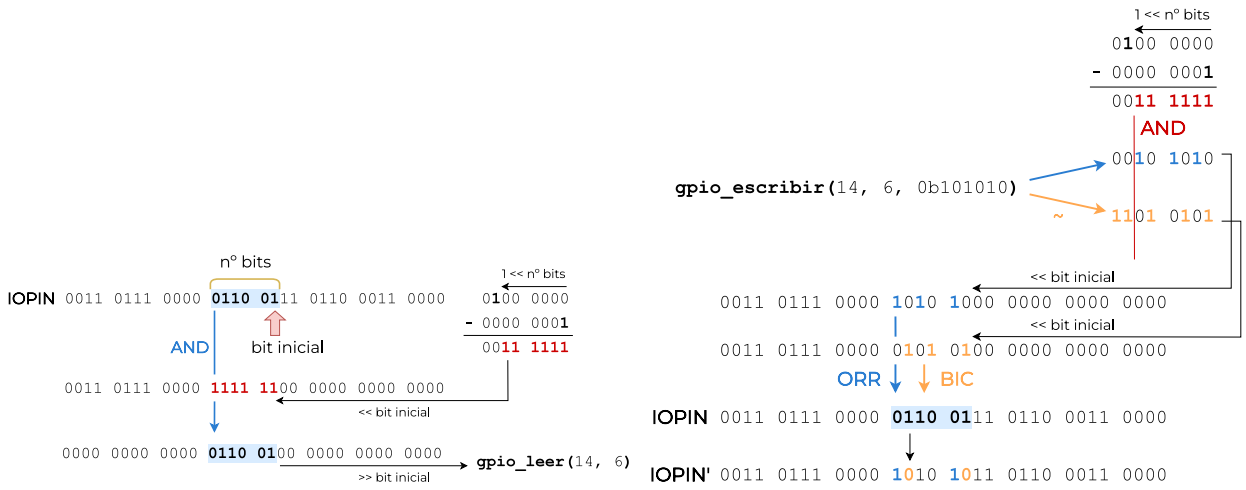

Figura 4: Operaciones a nivel de bits necesarias para leer y escribir en el registro IOPIN.

4. Interacción con el juego

El sistema se controla a través de tres periféricos: el GPIO, que se utiliza para controlar una serie de pines de propósito general; las interrupciones externas EXTINT, configuradas para ser activadas al pulsar un botón; y el transmisor asíncrono UART0, encargado de recibir y enviar caracteres. El usuario interactúa con el sistema a través de estos periféricos.

4.1. Control de pines y GPIO

El GPIO es un periférico que ofrece un conjunto de 32 pines de entrada y salida configurables (registro IOPIN). En la última versión del proyecto, solo se utilizan los pines configurados para salida. Sin embargo, la interfaz del GPIO debe permitir tanto la escritura como la lectura de los pines. En concreto, las operaciones disponibles son: leer un número determinado de bits a partir de un bit inicial, escribir un vector de bits a partir de un bit inicial (Ver 4), marcar los bits de entrada y marcar los bits de salida.

El gestor de entrada, «g_io», enciende los pines de salida según los mensajes recibidos, tal y como se indica en la siguiente tabla:

Pin	E/S	Descripción	On	Off	Deprecated
GPIO1	S	Indicador jugador 1	JUGADOR(1)	JUGADOR(2)	X
GPIO2	S	Indicador jugador 2	JUGADOR(2)	JUGADOR(1)	X
GPIO3-9	E	Columna seleccionada como jugada	LEER_ENTRADA		X
GPIO14	E	EINT1 (realizar la jugada)			X
GPIO14	E	EINT1 (cancelar jugada)			
GPIO15	E	EINT2 (reiniciar partida)			
GPIO16	S	Jugada realizada	JUGADA_REALIZADA	APAGAR_REALIZADA	
GPIO17	S	Jugada no válida (columna indicada llena o fuera de rango)	ENTRADA_VALIDADA	ENTRADA_VALIDADA	
GPIO18	S	Final de partida	FIN	CANCELAR/RESET	
GPIO19	S	Comando incorrecto, o recibe más de 5 caracteres	IGNORE_CMD	APAGAR_IGNORE_CMD	
GPIO30	S	Overflow	OVERFLOW_{M/E}		
GPIO31	S	Latido modo idle	LATIDO	LATIDO/APAGAR_LATIDO	

Aquellas filas marcadas como «deprecated» indican funcionalidades de una práctica anterior que han sido eliminadas porque generan conflictos con las añadidas más recientemente.

4.2. Botones

Actualmente el sistema cuenta con 2 botones activos: RESET y CANCELAR. Este último reemplaza a otro botón que implementaba la funcionalidad de efectuar una jugada. Para ello se han configurado las interrupciones externas EINT 1 y 2, generadas al modificar el valor de los pines 14 y 15 del GPIO, respectivamente; de tal manera que sirvan como entradas al usuario y simulen la funcionalidad de un botón. Cada una de ellas provoca que se genere un evento de tipo «PULSACION», el cual contendrá el número del botón pulsado.

Para evitar que se produzcan constantes interrupciones del botón antes de soltarlo, la rutina de interrupción deshabilita las interrupciones externas correspondientes al botón pulsado. Cuando el evento «PULSACION» es recibido por el gestor de alarmas, este encola un mensaje «EJECUTAR», que se detalla con mayor profundidad en el apartado 5, y configura una alarma periódica para dentro de 10 milisegundos con el mensaje «BAJAR_PULSACION_1/2». Al recibir ese mensaje, comprobará que el botón sigue pulsado. En caso de estarlo no hará nada. En caso contrario, elimina la alarma y habilita de nuevo las interrupciones para el correspondiente botón. Comprobar que un botón está pulsado consiste en bajar el flag de interrupción en el registro EXTINT y a continuación volverlo a leer. Si el flag vuelve a activarse, entonces el botón sigue pulsado.

4.3. Gestor serie y UART

En la primera versión de este proyecto, la interacción con el juego consistía en alterar manualmente los valores del tablero de juego en memoria. Más tarde se sustituyó por la entrada y salida a través de los pines del GPIO, aunque el tablero seguía siendo visualizado a través del mapa de memoria. En la versión actual, el tablero y la información del juego se muestran por medio de la línea de serie, a modo terminal. Este periférico también es utilizado para introducir comandos que ejecuten las jugadas de cada jugador, además de poder reiniciar y terminar una partida.

El periférico utilizado como línea de serie es UART0, el cual se configura en el módulo de C homónimo. Para habilitar tanto la entrada como salida de caracteres hay que configurar los bits 2 y 0 del registro PINSEL0. Esto provoca que los pines 0 y 1 del GPIO sean de uso exclusivo para la línea de serie, por lo que se ha eliminado la funcionalidad que permitía mostrar el turno, tal y como se indica en la sección 4.1.

Lo primero que se configura es la tasa de símbolos. Para ello se activa el bit DLAB del registro LCR y se fija el valor de los registros DLL y DLM, con el fin de obtener el estándar de 9666 baudios, según la siguiente fórmula, dónde DLM = 0 para simplificar el cálculo:

$$R_s = \frac{PCLK}{16 \cdot ((256 \cdot DLM) + DLL)} \overset{\text{DLM} = 0}{=\!=\!\Rightarrow} \frac{PCLK}{16 \cdot DLL}$$

$$DLL = \frac{PCLK}{R_s \cdot 16} = \frac{15 \cdot 10^6}{9666 \cdot 16} = 96{,}98 \approx 97$$

Una vez configurada la tasa de símbolos, se configura el registro LCR, deshabilitando el bit DLAB y determinando que el envío es de bytes completos, sin control de paridad. Con el registro IER habilitamos las interrupciones al recibir un carácter o completar el envío de un byte, y con el registro FCR se habilita el uso del buffer de recepción, además de determinar que un carácter basta para generar una interrupción. El periférico sólo tiene una rutina de interrupción, en la que se gestionan ambos eventos de recepción y envío. Cada uno provoca que se encole correspondiente evento «CARACTER_RECIBIDO», con el valor del carácter en el campo auxiliar, o «CARACTER_ENVIADO».

Entrada de comandos

La entrada de comandos se controla con una máquina de estados desde el gestor de serie, cuya entrada son los caracteres contenidos en el campo auxiliar de los eventos «CARACTER_RECIBIDO». Los comandos son de hasta 3 caracteres, por lo que los caracteres se van almacenando en un buffer de este mismo tamaño. Al recibir el carácter # el gestor comienza a almacenar los siguientes caracteres, hasta que se de una de estas situaciones:

- Se reciben más de 3 caracteres: el comando es descartado.
- Se recibe el carácter **!**: el comando es procesado y se ejecuta en caso de ser válido.
- Se recibe el carácter #: reinicia la recepción del comando.

Los comandos habilitados en el sistema son los siguientes:

- **END:** Provoca el fin de una partida. En caso de ser introducido el gestor encola el mensaje «FIN».
- **NEW:** Da comienzo a una nueva partida (o resetea la que está en curso). En caso de ser introducido el gestor encola el mensaje «RESET».
- **Cxx:** Realiza una jugada en la columna indicada con un valor no negativo en los caracteres **xx**. Si no corresponden con un número no negativo, el gestor determina que el comando es incorrecto. En caso contrario, el gestor encola el mensaje «JUGAR» La comprobación de si la columna es correcta la realiza el módulo del conecta 4.

Envío de cadenas

El envío de cadenas se realiza de forma asíncrona aprovechando las interrupciones que genera la línea de serie. El módulo del UART0 ofrece la función «uart0_enviar_array», que copia un string de tipo C (acabado en '\0') en su buffer interno e inicializa el índice de envío a la primera posición, y la función «uart0_continuar_envio», que envía el siguiente carácter de la cadena y devuelve false en caso de que no queden caracteres por enviar. No obstante, si no se controlan las llamadas a estas funciones se pueden solapar los envíos y sobreescribir el contenido del buffer antes de tiempo. Por ello, el gestor de serie maneja una cola de cadenas.

Cuando el gestor quiere enviar una cadena no llama a la función del módulo inferior. En su lugar encola el identificador de la cadena. La cola gestiona las llamadas a la función de envío, y envía una cadena si acaba de ser encolada y la cola está vacía o si es la siguiente cadena a desencolar. Cuando el gestor de serie reciba el evento «CARACTER_ENVIADO», continuará el envío; pero si no quedan caracteres por enviar de la anterior cadena, entonces la desencolará, provocando el envío de la siguiente sin solapamientos. La situación de overflow no provoca la detención del sistema, sino que se gestiona de tal manera que si la cola está llena ya no se permite añadir una nueva.

Almacenar todo el contenido de las cadenas en la cola resultaría en un uso muy elevado de memoria de lectura/escritura. La solución que se propone es declarar todas las cadenas en un mismo fichero «cadenas.h», donde se les asigna un identificador, que es lo que realmente almacena la cola. Existen dos tipos de cadenas: las constantes, situadas en zona de sólo lectura y accedidas a partir de su identificador en la tabla de cadenas (Ver 5), y las mutables, situadas en la pila y modificadas en tiempo de ejecución. Las cadenas constantes se identifican partiendo del valor 0 hacia arriba, mientras que las mutables adquieren los valores más altos, partiendo de 0xFF hacia abajo. Para aportar legibilidad al código, todos los identificadores se encuentran declarados en un tipo enumerado. El tamaño del buffer de envío y la cola de cadenas es de 32 bytes cada uno, y el de la tabla de traducción es de 132 bytes (33 punteros), por lo que podríamos llegar a tener 1KB de texto encolado para su envío utilizando tan solo el 20 % de la memoria de lectura/escritura necesaria.

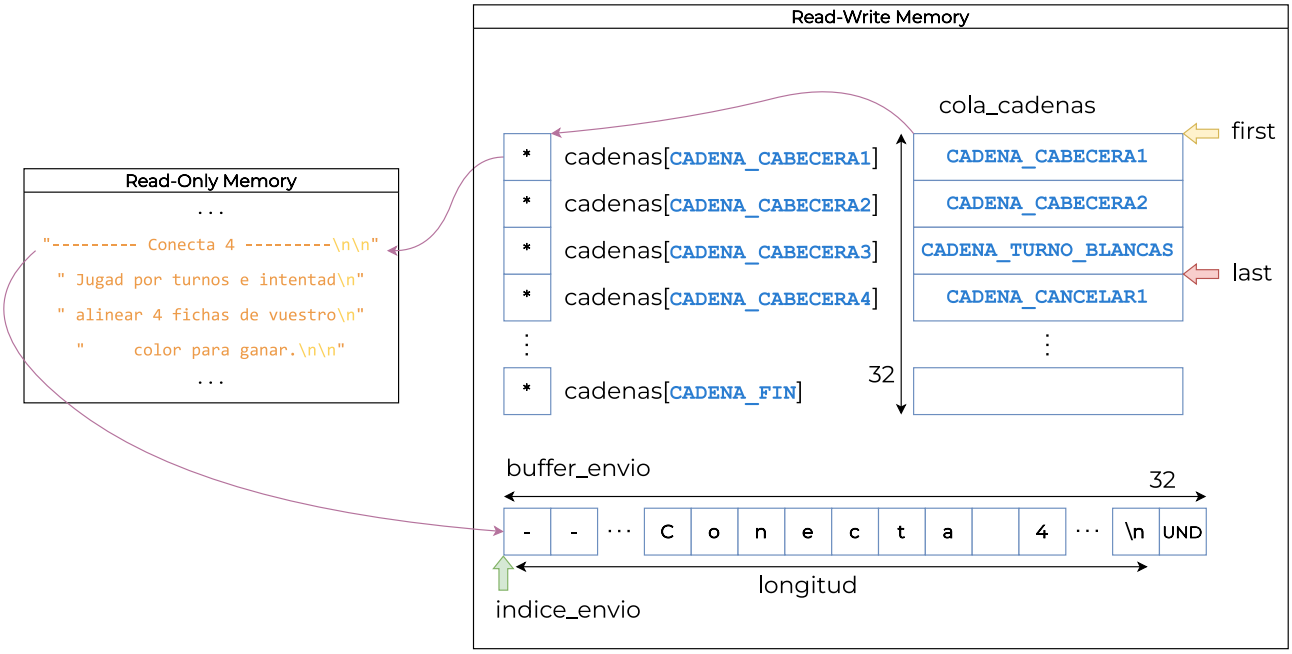

Figura 5: Esquema de acceso a las cadenas constantes.

Las cadenas mutables que maneja el gestor de serie requieren de datos pertenecientes a otros módulos gestores. Cuando llegue el momento de su envío el modulo encolará mensajes específicos a dichos módulos para que le respondan con la información precisa. Esa información consiste en valores enteros no negativos, por lo que ha sido preciso implementar una versión particular de la función «itoa», con el fin de traducirlos a texto. Las cadenas mutables son las siguientes:

- **CADENA_FILA1-6:** Esta cadena requiere enviar el mensaje «PEDIR_FILA» junto con el número de la fila solicitada. La respuesta la otorga el módulo del conecta 4, quien envía la fila en un mensaje «DEVOLVER_FILA» codificada en 32 bits, donde los 4 primeros bits corresponden con el número de fila, y el resto son los valores de las celdas de cada columna, separados cada 4 bits. Las celdas con color blanco se muestran con una B, las de negro con una N y las pendientes de confirmación con el símbolo *.
- **CADENA_CALIDAD_SERVICIO, CADENA_SEGUNDOS_JUGADOS y CADENA_MINUTOS_JUGADOS:** Al finalizar la partida se debe mostrar el tiempo medio de tratamiento de mensajes, que se pregunta al gestor de estadísticas con el mensaje «PEDIR_CALIDAD_SERVICIO». El tiempo de juego se solicita con los mensajes «PEDIR_SEGUNDOS_JUGADOS» y «PEDIR_MINUTOS_JUGADOS».

4.4. Toolbox en keil

El simulador Keil μVision, utilizado para ejecutar y depurar el código de este proyecto, permite configurar una «toolbox» con botones para que se ejecute un comando y simplificar la interacción con el usuario. Esta barra de herramientas incluye los botones CANCEL y RESET, que simulan una pulsación en los pines 14 y 15 del GPIO, además de los comandos habilitados para ser escritos a través de la línea de serie (NEW, END y C1-7). Aunque la interacción pase a ser mediante pulsaciones, el botón realmente introduce uno a uno los caracteres para ejecutar los comandos.

A continuación se muestran las instrucciones introducidas en la terminal de Keil para la definición de los botones:

```
DEFINE BUTTON "RESET", "PORT = PORT & ~0X8000; PORT = PORT | 0X8000"
DEFINE BUTTON "CANCEL", "PORT = PORT & ~0X4000; PORT = PORT | 0X4000"

DEFINE BUTTON "NEW", "SOIN = '#'; SOIN = 'N'; SOIN = 'E'; SOIN = 'W'; SOIN = '!'"
DEFINE BUTTON "END", "SOIN = '#'; SOIN = 'E'; SOIN = 'N'; SOIN = 'D'; SOIN = '!'"
DEFINE BUTTON "C1", "SOIN = '#'; SOIN = 'C';  SOIN = '1'; SOIN = '!'"
...
DEFINE BUTTON "C7", "SOIN = '#'; SOIN = 'C';  SOIN = '7'; SOIN = '!'"
```

5. Gestor de energía

Tal y como se describe en el apartado 2.1, si el planificador no detecta ningún evento o mensaje disponible para ser desencolado, deberá bloquear el avance de las instrucciones entrando en modo «idle» para evitar hacer uso innecesario de la energía en un bucle de espera activa. Este modo bloquea la ejecución de instrucciones, pero los periféricos siguen funcionando. Además, al producirse una interrupción, el procesador vuelve a su modo normal de forma automática. Pero en este proyecto no sólo se busca eliminar las esperas activas, sino optimizar todo lo posible el uso de energía. Es por ello que, transcurridos 10 segundos sin recibir un nueva pulsación o un nuevo carácter por la línea de serie, el procesador deberá entrar en modo «power-down».

Al iniciar la ejecución del juego, el gestor de energía programa una alarma que encole el mensaje «POWER_DOWN» pasados 10 segundos. Cuando el planificador ejecuta la función «g_energia_idle()», el gestor pasa a estado «IDLE». Las interrupciones no alteran el estado del gestor, pero sí determinados mensajes. Con el fin de detectar desde el gestor de energía la entrada de un nuevo carácter, el gestor de la línea de serie debe encolar un mensaje «RESET_POWERDOWN» cada vez que recibe el evento «CARACTER_RECIBIDO». Si este mensaje o «EJECUTAR» son recibidos, el gestor reprograma la alarma de «power-down» y regresa al estado «NORMAL».

En el momento en el que salta la alarma, el gestor pasa al estado «POWERDOWN» y provoca que la CPU entre en el modo de más bajo consumo. Lo único capaz de despertar al procesador en este modo son las interrupciones externas. Sin embargo, no será hasta que llegue el mensaje «EJECUTAR» que el gestor volverá a su estado «NORMAL». El motivo es evitar que la pulsación que despierta al procesador provoque cambios en el juego. Si ese mensaje llega en modo «NORMAL», el gestor de energía encola el mensaje contenido en el campo «auxData».

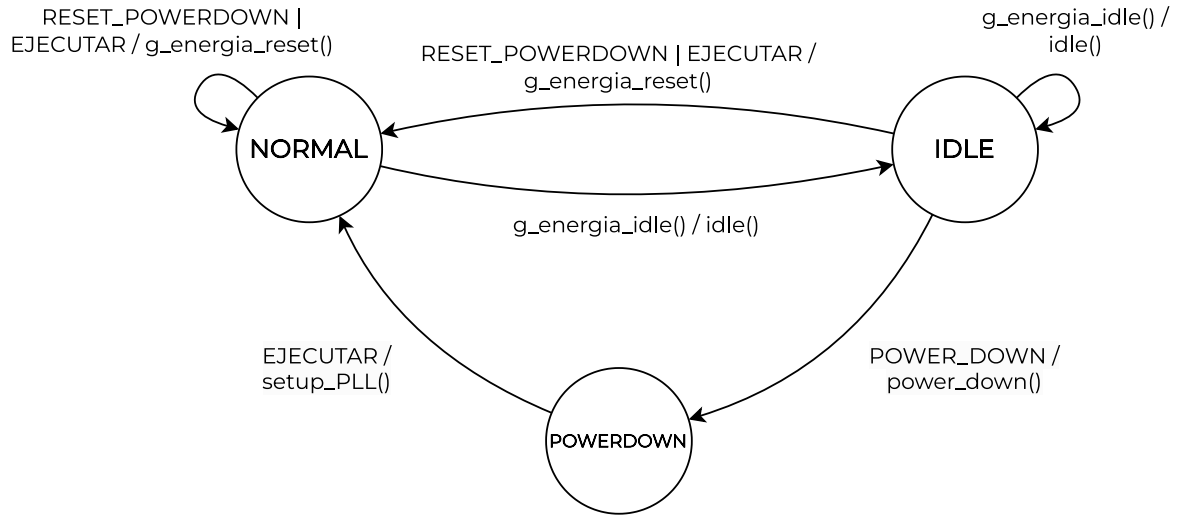

Figura 6: Máquina de estados del gestor de energía.

Si en cambio llega en modo «POWERDOWN», el gestor se limitará a cambiar de estado. La máquina de estados completa se puede visualizar en la figura 6.

Para provocar que el gestor I/O encienda y apague periódicamente el led indicador del modo «idle», cada vez que el gestor de energía cambia de estado «NORMAL» a «IDLE» programa una alarma periódica que encola el mensaje «LATIDO». Al regresar al estado «NORMAL», esta alarma es cancelada.

Las funciones utilizadas para cambiar el modo de energía de la CPU se encuentran implementadas en el módulo «power.c». Para cambiar al modo «idle» basta con activar el bit 0 del registro PCON; el bit 1 en caso del modo «power-down». Un punto importante es asegurarse de permitir al comienzo de la ejecución que las interrupciones externas despierten al procesador, activando los bits 1 y 2 del registro EXTWAKE.

La frecuencia del oscilador queda desconfigurada al entrar en modo «power-down». Por ello, el gestor de energía debe encargarse de reconfigurarla, restaurando el PLL. La solución más sencilla y por la que se optó fue exportar la rutina «setup_PLL» del fichero «Startup.s».

6. Lógica del juego

El juego comienza mostrando las cadenas de texto correspondientes a las instrucciones del juego y la información de uso. A continuación se muestra el mensaje para comenzar una partida, a la espera de la pulsación de uno de los botones o la introducción del comando NEW, para acto seguido mostrar el tablero de juego vacío y el turno. Cuando se introduce una ficha con el comando C, el juego comprueba si es válida. En caso afirmativo, el tablero se actualiza con la celda a completar marcada con una estrella. El jugador puede cancelar la jugada pulsando el botón 1 (CANCEL), pero si no lo hace en menos de un segundo la jugada será definitiva. En el caso de introducir una columna inválida, porque no corresponde con una columna del tablero o porque está llena, el juego lo comunica y vuelve a mostrar el mismo tablero.

En el momento en el que un jugador alinea 4 fichas de su color el juego termina con la victoria de ese jugador. Si el tablero se llena sin que ninguno de los jugadores consiga alinear 4 fichas, el juego termina con empate. Los jugadores también pueden introducir el comando END en cualquier momento de la partida, provocando que termine el juego de manera anticipada. En cualquiera de los casos de fin de partida el juego muestra el tiempo de juego en minutos y segundos y la calidad de servicio, mostrada como la media de tiempo de tratamiento de los mensajes, concluyendo con el mensaje de comienzo de una nueva partida.

Si alguno de los jugadores introduce el comando NEW o pulsa el botón 2 (RESET), dará comienzo una nueva partida sin mostrar el mensaje final y reseteando los tiempos de juego y calidad de servicio.

Estados de juego y gestión de mensajes

Para implementar la lógica del juego, el sistema tiene tres estados generales independientes de las máquinas de estado ya mencionadas: «INACTIVO», «ACTIVO» y «ESPERANDO» (los nombres y su gestión cambian según

el módulo). Al comenzar la ejecución el juego comienza en el estado «INACTIVO». En este estado sólo están habilitados los mensajes «RESET» y «CANCEL» (de aquellos que afectan directamente a la jugabilidad), correspondientes a los eventos de pulsación o el comando NEW. Al recibirlos, el sistema pasa al estado «ACTIVO» donde el único mensaje deshabilitado es «CANCEL». La introducción del comando END, una victoria o un empate provocarían que el sistema pase de nuevo a estado «INACTIVO».

Cuando una ficha es introducida con el comando C, el gestor de serie genera el mensaje «JUGAR» junto con la columna seleccionada. Si el sistema está en estado «INACTIVO» el mensaje sería ignorado, pero en estado «ACTIVO» es procesado por el módulo del conecta 4, el cual comprueba si la jugada es valida. El resultado de la comprobación se envía con el mensaje «ENTRADA_VALIDADA» para que el gestor de serie y el gestor I/O puedan mostrarlo. Si la jugada es válida se marca temporalmente en el tablero y se espera a la confirmación de la jugada. Previamente se encola el mensaje «CELDA_MARCADA» para que todo el sistema pase al estado «ESPERANDO». Además, se programa una alarma para que dentro de un segundo envíe el mensaje «CONFIRMAR_JUGADA». En el estado «ESPERANDO» el único mensaje habilitado es «CANCEL», lo que impide reiniciar o finalizar el juego, y tampoco permite ejecutar una segunda jugada antes de tiempo. Si se cancela la jugada, se elimina la alarma programada y el sistema vuelve al estado «ACTIVO». Si, por el contrario, el mensaje «CONFIRMAR_JUGADA» llega a ser encolado, el sistema vuelve al estado «ACTIVO» y el módulo del conecta 4 confirma la jugada:

1. Actualiza la celda marcada al color del jugador que realizó la jugada.
2. Encola el mensaje «JUGADA_REALIZADA» para que el gestor I/O pueda encender el led correspondiente durante un segundo (gestionado internamente con una alarma) y el gestor de serie muestre el tablero actualizado.
3. Comprueba si la partida debe finalizar por victoria o empate.

Si la partida continúa, el ciclo se repite a la espera de otra jugada. En caso de finalizar, el módulo del conecta 4 encola el mensaje «FIN» junto con la causa: el color del jugador victorioso o un valor especial para el empate. Si no hay contenido en el campo auxiliar se considera que la partida ha terminado por causa del comando END. Al procesar el mensaje, el sistema pasa a estado «INACTIVO» y el gestor de serie solicita toda la información necesaria al resto de módulos para mostrar el fin de partida.

7. Testing

Debido a que durante el curso normal del juego, el sistema no genera overflow, se ha implementado una función que comprueba el overflow en la cola de eventos y de mensajes, de tal manera que podemos probar, y con ello garantizar, que el sistema cumple con los requisitos planteados inicialmente al darse esta situación. Este test podemos encontrarlo en el fichero: «main.c» bajo el nombre de «test_overflow». La función requiere de un parámetro entero que podrá valer 1 para realizar la prueba sobre la cola de eventos, 2 para la cola de mensajes o cualquier otro número para ignorar el test.

Además de este test, cada funcionalidad fue probada de manera individual para comprobar su correcto funcionamiento antes de pasar a la siguiente.

8. Conclusiones

Entre las principales conclusiones de este proyecto destaca la concepción de la dificultad que reside en la implementación del software más básico de bajo nivel cuando se trata de realizar acciones de entrada y salida dentro del procesador. Normalmente se tiene asumido que uno siempre va a contar con un medio de depuración a través de una línea de serie, pero al momento de implementar la interacción esta ayuda no se encuentra disponible.

El proyecto se ha realizado por partes. Cada una eliminaba conceptos de la anterior e introducía otros nuevos, por lo que la modularidad jugaba un papel muy importante y se ha notado, sobre todo al introducir la interacción con los nuevos gestores.

En cuanto al esfuerzo requerido, pensamos que la carga de trabajo en esta asignatura, pese a ser muy elevada, está bien repartida a lo largo del cuatrimestre en comparación al resto de asignaturas, las cuales tienden a concentrar todo el trabajo en un periodo de tiempo muy reducido. Pensamos que el feedback presencial de los profesores no ha sido el esperado, aunque la información colgada en la wiki sí que ha sido de gran ayuda.

El modo de trabajo dentro del grupo ha sido conjunto la gran mayoría del tiempo, alternando los papeles de redactor y revisor de código. Creemos que de esta manera se detectan con mayor facilidad los errores de código. El trabajo de ambos miembros ha sido constante y consensuado. En ocasiones causadas por una mayor carga de trabajo procedente de otras asignaturas, el trabajo a tenido que ser relevado y ciertas partes del proyecto se realizaron individualmente con la posterior revisión del otro compañero. El resultado es un proyecto que cumple completamente con las funcionalidades indicadas y por ello valoramos nuestro trabajo con una puntuación 9.5.

9. Dedicación

Tarea	**Fernando**	**Héctor**	**Comentario**
Paso 0	12	12	Estudio de la documentación
Paso 1	4	4	Implementación de la cola de eventos
Paso 2	12	12	Configuración de los temporizadores
Paso 3	1	1	Gestor de alarmas
Paso 4	1	1	Configuración de GPIO y gestor I/O
Paso 5	4	4	Configuración y gestión de botones
Paso 6	3	3	Gestor de energía
Paso 7	12	12	Integración de la I/O en el juego
Paso D	2	2	Documentación y refactorización de los métodos implementados
Paso 8	2	2	Tratamiento de TIMER0 RSI como fast interrupt
Paso 9	4	4	Implementación de las llamadas al sistema
Paso 10	3	3	Configuración del Real Time Clock y Watchdog
Paso 11	2	1.5	Eliminación de las condiciones de carrera
Paso 12	13	11	Configuración de la línea serie y gestión de cadenas
Paso 13	2	2	Máquina de estados del juego completo en su implementación final
Paso 14	15	8	Depuración final
Paso D	2	2	Documentación y refactorización de los métodos implementados
Memoria	25	23	
Total	**119**	**107.5**	

10. Anexos

Startup

```
;
   /*****************************************************************************/

;/* STARTUP.S: Startup file for Blinky Example
   */
;
   /*****************************************************************************/

;/* <<< Use Configuration Wizard in Context Menu >>>
   */
;
   /*****************************************************************************/

;/* This file is part of the uVision/ARM development tools.
   */
;/* Copyright (c) 2005-2006 Keil Software. All rights reserved.
   */
;/* This software may only be used under the terms of a valid, current,
   */
;/* end user licence from KEIL for a compatible version of KEIL software
   */
;/* development tools. Nothing else gives you the right to use this software.
   */
;
   /*****************************************************************************/



;/*
; *  The STARTUP.S code is executed after CPU Reset. This file may be
; *  translated with the following SET symbols. In uVision these SET
; *  symbols are entered under Options - ASM - Define.
; *
; *  REMAP: when set the startup code initializes the register MEMMAP
; *  which overwrites the settings of the CPU configuration pins. The
; *  startup and interrupt vectors are remapped from:
; *     0x00000000  default setting (not remapped)
; *     0x40000000  when RAM_MODE is used
; *
; *  RAM_MODE: when set the device is configured for code execution
; *  from on-chip RAM starting at address 0x40000000.
; */

; Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs

Mode_USR        EQU     0x10
Mode_FIQ        EQU     0x11
Mode_IRQ        EQU     0x12
Mode_SVC        EQU     0x13
Mode_ABT        EQU     0x17
Mode_UND        EQU     0x1B
Mode_SYS        EQU     0x1F

I_Bit           EQU     0x80            ; when I bit is set, IRQ is disabled
```

```
F_Bit           EQU     0x40            ; when F bit is set, FIQ is disabled


;// <h> Stack Configuration (Stack Sizes in Bytes)
;//   <o0> Undefined Mode      <0x0-0xFFFFFFFF:8>
;//   <o1> Supervisor Mode     <0x0-0xFFFFFFFF:8>
;//   <o2> Abort Mode          <0x0-0xFFFFFFFF:8>
;//   <o3> Fast Interrupt Mode <0x0-0xFFFFFFFF:8>
;//   <o4> Interrupt Mode      <0x0-0xFFFFFFFF:8>
;//   <o5> User/System Mode    <0x0-0xFFFFFFFF:8>
;// </h>

UND_Stack_Size  EQU     0x00000000
SVC_Stack_Size  EQU     0x00000400
ABT_Stack_Size  EQU     0x00000000
FIQ_Stack_Size  EQU     0x00000080
IRQ_Stack_Size  EQU     0x00000080
USR_Stack_Size  EQU     0x00000400

Stack_Size      EQU     (UND_Stack_Size + SVC_Stack_Size + ABT_Stack_Size + \
                         FIQ_Stack_Size + IRQ_Stack_Size + USR_Stack_Size)

                AREA    STACK, NOINIT, READWRITE, ALIGN=3
            PRESERVE8 {TRUE}
Stack_Mem       SPACE   Stack_Size

Stack_Top       EQU     Stack_Mem + Stack_Size


;// <h> Heap Configuration
;//   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF>
;// </h>

Heap_Size       EQU     0x00000000

                AREA    HEAP, NOINIT, READWRITE, ALIGN=3
Heap_Mem        SPACE   Heap_Size


; VPBDIV definitions
VPBDIV          EQU     0xE01FC100      ; VPBDIV Address

;// <e> VPBDIV Setup
;// <i> Peripheral Bus Clock Rate
;//   <o1.0..1>   VPBDIV: VPB Clock
;//               <0=> VPB Clock = CPU Clock / 4
;//               <1=> VPB Clock = CPU Clock
;//               <2=> VPB Clock = CPU Clock / 2
;//   <o1.4..5>   XCLKDIV: XCLK Pin
;//               <0=> XCLK Pin = CPU Clock / 4
;//               <1=> XCLK Pin = CPU Clock
;//               <2=> XCLK Pin = CPU Clock / 2
;// </e>
VPBDIV_SETUP    EQU     0
VPBDIV_Val      EQU     0x00000000


; Phase Locked Loop (PLL) definitions
```

```
PLL_BASE        EQU     0xE01FC080      ; PLL Base Address
PLLCON_OFS      EQU     0x00            ; PLL Control Offset
PLLCFG_OFS      EQU     0x04            ; PLL Configuration Offset
PLLSTAT_OFS     EQU     0x08            ; PLL Status Offset
PLLFEED_OFS     EQU     0x0C            ; PLL Feed Offset
PLLCON_PLLE     EQU     (1<<0)          ; PLL Enable
PLLCON_PLLC     EQU     (1<<1)          ; PLL Connect
PLLCFG_MSEL     EQU     (0x1F<<0)       ; PLL Multiplier
PLLCFG_PSEL     EQU     (0x03<<5)       ; PLL Divider
PLLSTAT_PLOCK   EQU     (1<<10)         ; PLL Lock Status

;// <e> PLL Setup
;//   <o1.0..4>   MSEL: PLL Multiplier Selection
;//               <1-32><#-1>
;//               <i> M Value
;//   <o1.5..6>   PSEL: PLL Divider Selection
;//               <0=> 1   <1=> 2   <2=> 4   <3=> 8
;//               <i> P Value
;// </e>
PLL_SETUP       EQU     1
PLLCFG_Val      EQU     0x00000024


; Memory Accelerator Module (MAM) definitions
MAM_BASE        EQU     0xE01FC000      ; MAM Base Address
MAMCR_OFS       EQU     0x00            ; MAM Control Offset
MAMTIM_OFS      EQU     0x04            ; MAM Timing Offset

;// <e> MAM Setup
;//   <o1.0..1>   MAM Control
;//               <0=> Disabled
;//               <1=> Partially Enabled
;//               <2=> Fully Enabled
;//               <i> Mode
;//   <o2.0..2>   MAM Timing
;//               <0=> Reserved  <1=> 1   <2=> 2   <3=> 3
;//               <4=> 4         <5=> 5   <6=> 6   <7=> 7
;//               <i> Fetch Cycles
;// </e>
MAM_SETUP       EQU     1
MAMCR_Val       EQU     0x00000002
MAMTIM_Val      EQU     0x00000004


; Area Definition and Entry Point
;  Startup Code must be linked first at Address at which it expects to run.

                AREA    RESET, CODE, READONLY
                ARM

; Exception Vectors
;  Mapped to Address 0.
;  Absolute addressing mode must be used.
;  Dummy Handlers are implemented as infinite loops which can be modified.

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
```

```
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP                             ; Reserved Vector
;               LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0FF0]      ; Vector from VicVectAddr
                LDR     PC, FIQ_Addr

  IMPORT  SWI_Handler

Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
                DCD     0                       ; Reserved Address
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler

  EXTERN  timer0_IRC

Undef_Handler   B       Undef_Handler
; SWI_Handler     B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler
IRQ_Handler     B       IRQ_Handler
FIQ_Handler     B       timer0_IRC


; Reset Handler

                EXPORT  Reset_Handler
Reset_Handler


; Setup VPBDIV
                IF      VPBDIV_SETUP <> 0
                LDR     R0, =VPBDIV
                LDR     R1, =VPBDIV_Val
                STR     R1, [R0]
                ENDIF

; Setup PLL
                IF      PLL_SETUP <> 0
                LDR     R0, =PLL_BASE
                MOV     R1, #0xAA
                MOV     R2, #0x55

;  Configure and Enable PLL
                MOV     R3, #PLLCFG_Val
                STR     R3, [R0, #PLLCFG_OFS]
                MOV     R3, #PLLCON_PLLE
                STR     R3, [R0, #PLLCON_OFS]
                STR     R1, [R0, #PLLFEED_OFS]
                STR     R2, [R0, #PLLFEED_OFS]

;  Wait until PLL Locked
PLL_Loop        LDR     R3, [R0, #PLLSTAT_OFS]
                ANDS    R3, R3, #PLLSTAT_PLOCK
```

```
                BEQ     PLL_Loop

;  Switch to PLL Clock
                MOV     R3, #(PLLCON_PLLE:OR:PLLCON_PLLC)
                STR     R3, [R0, #PLLCON_OFS]
                STR     R1, [R0, #PLLFEED_OFS]
                STR     R2, [R0, #PLLFEED_OFS]
                ENDIF   ; PLL_SETUP

; Setup MAM
                IF      MAM_SETUP <> 0
                LDR     R0, =MAM_BASE
                MOV     R1, #MAMTIM_Val
                STR     R1, [R0, #MAMTIM_OFS]
                MOV     R1, #MAMCR_Val
                STR     R1, [R0, #MAMCR_OFS]
                ENDIF   ; MAM_SETUP


; Memory Mapping (when Interrupt Vectors are in RAM)
MEMMAP          EQU     0xE01FC040      ; Memory Mapping Control
                IF      :DEF:REMAP
                LDR     R0, =MEMMAP
                IF      :DEF:RAM_MODE
                MOV     R1, #2
                ELSE
                MOV     R1, #1
                ENDIF
                STR     R1, [R0]
                ENDIF


; Initialise Interrupt System
;  ...


; Setup Stack for each mode

                LDR     R0, =Stack_Top

;  Enter Undefined Instruction Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #UND_Stack_Size

;  Enter Abort Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_ABT:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #ABT_Stack_Size

;  Enter FIQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #FIQ_Stack_Size

;  Enter IRQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
```

```
                SUB     R0, R0, #IRQ_Stack_Size

;  Enter Supervisor Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #SVC_Stack_Size

;  Enter User Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_USR
                MOV     SP, R0
                SUB     SL, SP, #USR_Stack_Size
; Enter the C code
                IMPORT  __main
                LDR     R0, =__main
                BX      R0


; User Initial Stack & Heap
                AREA    |.text|, CODE, READONLY

                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap
__user_initial_stackheap

                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + USR_Stack_Size)
                LDR     R2, = (Heap_Mem +      Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR

                EXPORT setup_PLL
setup_PLL
; Setup PLL
                LDR     R0, =PLL_BASE
                MOV     R1, #0xAA
                MOV     R2, #0x55

;  Configure and Enable PLL
                MOV     R3, #PLLCFG_Val
                STR     R3, [R0, #PLLCFG_OFS]
                MOV     R3, #PLLCON_PLLE
                STR     R3, [R0, #PLLCON_OFS]
                STR     R1, [R0, #PLLFEED_OFS]
                STR     R2, [R0, #PLLFEED_OFS]

;  Wait until PLL Locked
PLL_Loop_        LDR     R3, [R0, #PLLSTAT_OFS]
                ANDS    R3, R3, #PLLSTAT_PLOCK
                BEQ     PLL_Loop_

;  Switch to PLL Clock
                MOV     R3, #(PLLCON_PLLE:OR:PLLCON_PLLC)
                STR     R3, [R0, #PLLCON_OFS]
                STR     R1, [R0, #PLLFEED_OFS]
                STR     R2, [R0, #PLLFEED_OFS]
                BX      LR
                END
```

Listing 1: Startup.s

Programa principal

```
/**
 * @file main.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Programa principal del juego conecta 4
 */

#include "cola_asyn.h"
#include "cola_msg.h"
#include "conecta4_2022.h"
#include "contadores.h"
#include "g_alarmas.h"
#include "g_boton.h"
#include "g_energia.h"
#include "g_estadisticas.h"
#include "g_io.h"
#include "g_serie.h"
#include "llamadas_sistema.h"

/**
 * @brief Función que prueba la respuesta del sistema ante el overflow
 * @param case 1: Test Overflow para la cola de eventos
 *             2: Test Overflow para la cola de mensajes
 *             default: No se realiza ninguna prueba
 */
void test_overflow(int _case) {
  switch (_case) {
    case 1:
      for (int i = 0; i <= COLA_EVENTOS_SIZE; i++) {
        cola_encolar_eventos(PULSACION, i, 0);
      }
      break;
    case 2:
      for (int i = 0; i <= COLA_MSG_SIZE; i++) {
        cola_encolar_msg(LATIDO, 0);
      }
      break;
  }
}

void init(void) {
  g_io_iniciar();
  g_boton_iniciar();
  g_alarma_iniciar();
  g_energia_iniciar();
  g_serie_iniciar();

  WD_init(1);
  WD_feed();
}

int main(void) {
  int hay_evento, hay_msg;
  init();
  test_overflow(0);
  while (1) {
```

```
    hay_evento = cola_hay_eventos();
    if (hay_evento) {
      evento_t evento = cola_desencolar_eventos();
      g_alarma_tratar_evento(evento);
      g_io_tratar_evento(evento);
      g_serie_tratar_evento(evento);
      g_boton_tratar_evento(evento);
    }
    hay_msg = cola_hay_msg();
    if (hay_msg) {
      msg_t msg = cola_desencolar_msg();
      g_alarma_tratar_mensaje(msg);
      g_energia_tratar_mensaje(msg);
      g_io_tratar_mensaje(msg);
      g_serie_tratar_mensaje(msg);
      g_boton_tratar_mensaje(msg);
      conecta4_tratar_mensaje(msg);
      // El de estadísticas es el último para medir el
      // tiempo que tardan todos
      g_estadisticas_tratar_mensaje(msg);
    }
    if (!hay_evento && !hay_msg) g_energia_idle();
    WD_feed();
  }
}
```

Listing 2: main.c

Cola de eventos

```
/**
 * @file cola_asyn.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de la cola de eventos
 */

#ifndef COLA_ASYNC_H
#define COLA_ASYNC_H
#include <inttypes.h>

#include "eventos.h"
#include "msg.h"
#include "semaforo_interrupciones.h"
#include "utils.h"

enum { COLA_EVENTOS_SIZE = 32 };

typedef struct evento_t {
  uint32_t veces;
  uint32_t auxData;
  uint8_t ID_evento;
} evento_t;

/**
 * @brief  Función que encola un evento a la cola de eventos (sin bloquear
 * interrupciones)
 * @param ID_msg Identificador único del evento
```

```
 * @param veces Número de veces que ha aparecido un evento
 * @param auxData Información extra sobre el evento
 */
void cola_encolar_eventos_raw(uint8_t ID_evento, uint32_t veces,
                              uint32_t auxData);

/**
 * @brief  Función que encola un evento a la cola de eventos (bloqueando
 * interrupciones)
 * @param ID_msg Identificador único del evento
 * @param veces Número de veces que ha aparecido un evento
 * @param auxData Información extra sobre el evento
 */
void cola_encolar_eventos(uint8_t ID_evento, uint32_t veces, uint32_t auxData)
    ;

/**
 * @brief  Función que desencola un evento de la cola de eventos
 * @return Evento desencolado
 */
evento_t cola_desencolar_eventos(void);

/**
 * @brief  Función que comprueba si hay eventos
 * @return Número de eventos
 */
int cola_hay_eventos(void);

#endif
```

Listing 3: cola_asyn.h

```
/**
 * @file cola_asyn.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de la cola de
 * eventos
 */

#include "cola_asyn.h"

// Importante que las interrupciones lean exactamente el mismo que
// la ejecución en modo usuario
static volatile uint8_t first = 0, last = 0, full = FALSE;

static uint32_t colaVECES[COLA_EVENTOS_SIZE];
static uint32_t colaDATA[COLA_EVENTOS_SIZE];
static uint8_t colaID[COLA_EVENTOS_SIZE];

void cola_encolar_eventos(uint8_t ID_evento, uint32_t veces, uint32_t auxData)
    {
  uint32_t flags = bloquear_interrupciones(); /*LOCK*/
  cola_encolar_eventos_raw(ID_evento, veces, auxData);
  liberar_interrupciones(flags); /*UNLOCK*/
}

void cola_encolar_eventos_raw(uint8_t ID_evento, uint32_t veces,
```

```
                                uint32_t auxData) {
  if (full) {  // overflow
    // last == first -> escribe al comienzo de la cola
    colaID[last] = OVERFLOW_E;
  } else {
    colaVECES[last] = veces;
    colaDATA[last] = auxData;
    colaID[last] = ID_evento;

    last++;
    if (last == COLA_EVENTOS_SIZE) {
      last = 0;
    }
    if (last == first) {
      full = TRUE;
    }
  }
}

evento_t cola_desencolar_eventos(void) {
  evento_t evento;

  // No hace falta bloquear, porque el único evento que puede ser escrito en
  // esta posición es OVERFLOW, y solamente se sobreescribe el ID. El resto de
  // valores no pueden ser sobreescritos.
  evento.veces = colaVECES[first];
  evento.auxData = colaDATA[first];

  // La escritura ambas variables debe ser atómica, para evitar considerar que
  // la cola no está llena en caso de que una interrupción complete la cola.
  uint32_t flags = bloquear_interrupciones(); /*LOCK*/
  evento.ID_evento = colaID[first];
  first++;
  if (first == COLA_EVENTOS_SIZE) {
    first = 0;
  }
  full = FALSE;
  liberar_interrupciones(flags); /*UNLOCK*/

  return evento;
}

int cola_hay_eventos(void) {
  // Las interrupciones sólo pueden alterar el último,
  // por lo que no importa leer el primero y el último a la vez.
  return first != last || full;
}
```

Listing 4: cola_asyn.c

```
/**
 * @file eventos.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de los eventos
 */

#ifndef EVENTOS_H
```

```
#define EVENTOS_H

enum Eventos {
  TEMPORIZADOR,
  PULSACION,
  CARACTER_RECIBIDO,
  CARACTER_ENVIADO,
  OVERFLOW_E
};

#endif
```

Listing 5: eventos.h

Cola de mensajes

```
/**
 * @file cola_msg.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de la cola de mensajes
 */

#ifndef COLA_MSG_H
#define COLA_MSG_H
#include <inttypes.h>

#include "eventos.h"
#include "msg.h"
#include "temporizador.h"
#include "utils.h"

enum { COLA_MSG_SIZE = 16 };

typedef struct msg_t {
  uint32_t timestamp;
  uint32_t auxData;
  uint8_t ID_msg;
} msg_t;

/**
 * @brief  Función que encola un mensaje a la cola de mensajes
 * @param ID_msg Identificador único del mensaje
 * @param auxData Información extra sobre el mensaje
 */
void cola_encolar_msg(uint8_t ID_msg, uint32_t auxData);

/**
 * @brief  Función que desencola un mensaje de la cola de mensajes
 * @return Mensaje desencolado
 */
msg_t cola_desencolar_msg(void);

/**
 * @brief  Función que comprueba si hay mensajes
 * @return Número de mensajes
 */
int cola_hay_msg(void);
```

```
#endif
```

Listing 6: cola_msg.h

```
/**
 * @file cola_msg.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de la cola de
 * mensajes
 */

#include "cola_msg.h"

static uint8_t first = 0, last = 0, full = FALSE;
static uint32_t colaDATA[COLA_MSG_SIZE];
static uint32_t colaTIME[COLA_MSG_SIZE];
static uint8_t colaID[COLA_MSG_SIZE];

void cola_encolar_msg(uint8_t ID_msg, uint32_t auxData) {
  if (full) {  // overflow
    colaID[last] = OVERFLOW_M;
  } else {
    colaDATA[last] = auxData;
    colaTIME[last] = temporizador_leer();
    colaID[last] = ID_msg;
    last++;
    if (last == COLA_MSG_SIZE) {
      last = 0;
    }
    if (last == first) {
      full = TRUE;
    }
  }
}

msg_t cola_desencolar_msg(void) {
  msg_t msg;

  msg.timestamp = colaTIME[first];
  msg.auxData = colaDATA[first];
  msg.ID_msg = colaID[first];
  first++;
  if (first == COLA_MSG_SIZE) {
    first = 0;
  }
  full = FALSE;
  return msg;
}

int cola_hay_msg(void) { return first != last || full; }
```

Listing 7: cola_msg.c

```
/**
 * @file msg.h
```

```
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de los mensajes
 */

#ifndef MSG_H
#define MSG_H

enum Mensajes {
  SET_ALARM,  // auxData := 8 bits ID:1 bit esPeriodica:23 bits retardo
  BAJAR_PULSACION_1,
  BAJAR_PULSACION_2,
  POWER_DOWN,

  LATIDO,
  APAGAR_LATIDO,

  PEDIR_JUGADOR,
  JUGADOR,            // auxData := turno
  ENTRADA_VALIDADA,  // auxData := celda_valida

  CELDA_MARCADA,
  CONFIRMAR_JUGADA,
  JUGADA_REALIZADA,
  APAGAR_REALIZADA,
  IGNORE_CMD,
  APAGAR_IGNORE_CMD,

  JUGAR,     // auxData := columna
  EJECUTAR,  // auxData := mensaje a encolar
  RESET_POWERDOWN,

  PEDIR_FILA,
  DEVOLVER_FILA,

  PEDIR_CALIDAD_SERVICIO,
  PEDIR_SEGUNDOS_JUGADOS,
  PEDIR_MINUTOS_JUGADOS,

  CALIDAD_SERVICIO,  // auxData := tiempo
  SEGUNDOS_JUGADOS,  // auxData := segundos
  MINUTOS_JUGADOS,   // auxData := minutos

  CANCELAR,
  RESET,
  FIN,  // auxData := ganador, empate o nada
  OVERFLOW_M
};

#endif
```

Listing 8: msg.h

Llamadas al sistema

```
/**
 * @file llamadas_sistema.h
 * @authors: Fernando Lahoz & Héctor Toral
```

```
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de las llamadas al
 * sistema
 */

#ifndef LLAMADAS_SISTEMA_H
#define LLAMADAS_SISTEMA_H

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

#include "temporizador.h"
#include "utils.h"

/**
 * @brief Devuelve el registro del real-time clock
 */
uint32_t __swi(0) clock_gettime(void);

/**
 * @brief Lee el tiempo transcurrido en microsegundos utilizando el timer 1.
 */
uint32_t __swi(1) clock_get_us(void);

/**
 * @brief Lee el bit IRQ del registro de estado para saber si las
   interrupciones
 *        están habilitadas o deshabilitadas
 */
uint8_t __swi(2) read_IRQ_bit(void);

/**
 * @brief Lee el bit FIQ del registro de estado para saber si las
   interrupciones
 *        están habilitadas o deshabilitadas
 */
uint8_t __swi(3) read_FIQ_bit(void);

/**
 * @brief Activa sólo las interrupciones fiq en el registro de estado.
 */
void __swi(0xFF) enable_fiq(void);

/**
 * @brief Desactiva sólo las interrupciones fiq en el registro de estado.
 */
void __swi(0xFE) disable_fiq(void);

/**
 * @brief Activa interrupciones y fiq en el registro de estado.
 */
void __swi(0xFD) enable_irq_fiq(void);

/**
 * @brief Desactiva interrupciones y fiq en el registro de estado.
 */
void __swi(0xFC) disable_irq_fiq(void);
```

```
/**
 * @brief Activa sólo las interrupciones irq en el registro de estado.
 */
void __swi(0xFB) enable_irq(void);

/**
 * @brief Desactiva sólo las interrupciones irq en el registro de estado.
 */
void __swi(0xFA) disable_irq(void);

#endif  // LLAMADAS_SISTEMA_H
```

Listing 9: llamadas_sistema.h

```
/**
 * @file llamadas_sistema.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de las llamadas al
 * sistema.
 * @note: El resto de llamadas al sistema se encuentran en el archivo SWI.s
 */

#include "llamadas_sistema.h"

uint32_t __SWI_0() { return CTIME0; }

uint32_t __SWI_1(void) { return T1TC; }
```

Listing 10: llamadas_sistema.c

```
;
    /*****************************************************************************/

;/* SWI.S: SWI Handler
    */
;
    /*****************************************************************************/

;/* This file is part of the uVision/ARM development tools.
    */
;/* Copyright (c) 2005-2006 Keil Software. All rights reserved.
    */
;/* This software may only be used under the terms of a valid, current,
    */
;/* end user licence from KEIL for a compatible version of KEIL software
    */
;/* development tools. Nothing else gives you the right to use this software.
    */
;
    /*****************************************************************************/


; Standard definitions of Interrupt (I & F) flags and thumb bit in PSRs

I_Bit           EQU     0x80            ; when I bit is set, IRQ is disabled
F_Bit           EQU     0x40            ; when F bit is set, FIQ is disabled
```

```
T_Bit           EQU     0x20

                PRESERVE8                       ; 8-Byte aligned Stack
                AREA    SWI_Area, CODE, READONLY
                ARM

                EXPORT  SWI_Handler
SWI_Handler

                STMFD   SP!, {R12, LR}          ; Store R12, LR
                MRS     R12, SPSR               ; Get SPSR
                STMFD   SP!, {R8, R12}          ; Store R8, SPSR
                TST     R12, #T_Bit             ; Check Thumb Bit
                LDRNEH  R12, [LR,#-2]           ; Thumb: Load Halfword
                BICNE   R12, R12, #0xFF00       ; Extract SWI Number
                LDREQ   R12, [LR,#-4]           ; ARM:   Load Word
                BICEQ   R12, R12, #0xFF000000   ; Extract SWI Number

                CMP     R12, #0xFF
                BEQ     __enable_fiq
                CMP     R12, #0xFE
                BEQ     __disable_fiq
                CMP     R12, #0xFD
                BEQ     __enable_irq_fiq
                CMP     R12, #0xFC
                BEQ     __disable_irq_fiq
                CMP     R12, #0xFB
                BEQ     __enable_irq
                CMP     R12, #0xFA
                BEQ     __disable_irq

                LDR     R8, SWI_Count
                CMP     R12, R8
                BHS     SWI_Dead                ; Overflow
                ADR     R8, SWI_Table
                LDR     R12, [R8,R12,LSL #2]    ; Load SWI Function Address
                MOV     LR, PC                  ; Return Address
                BX      R12                     ; Call SWI Function

                LDMFD   SP!, {R8, R12}          ; Load R8, SPSR
                MSR     SPSR_cxsf, R12          ; Set SPSR
                LDMFD   SP!, {R12, PC}^         ; Restore R12 and Return

SWI_Dead        B       SWI_Dead                ; None Existing SWI

SWI_Cnt         EQU    (SWI_End-SWI_Table)/4
SWI_Count       DCD     SWI_Cnt

                IMPORT  __SWI_0
                IMPORT  __SWI_1
__SWI_2         EQU     read_IRQ_bit
__SWI_3         EQU     read_FIQ_bit

SWI_Table
                DCD     __SWI_0                 ; SWI 0 Function Entry
                DCD     __SWI_1                 ; SWI 1 Function Entry
                DCD     __SWI_2                 ; SWI 2 Function Entry
                DCD     __SWI_3                 ; SWI 3 Function Entry
```

```
SWI_End

;/**
; * @brief Lee el bit IRQ del registro de estado para saber si las
    interrupciones
; *        están habilitadas o deshabilitadas
; */
read_IRQ_bit
  LDMFD   SP!, {R8, R12}         ; Load R8, SPSR
  AND     R0, R12, #I_Bit
  MOV     R0, R0, LSR #7
  MSR     SPSR_cxsf, R12         ; Set SPSR
  LDMFD   SP!, {R12, PC}^        ; Restore R12 and Return

;/**
; * @brief Lee el bit FIQ del registro de estado para saber si las
    interrupciones
; *        están habilitadas o deshabilitadas
; */
read_FIQ_bit
  LDMFD   SP!, {R8, R12}         ; Load R8, SPSR
  AND     R0, R12, #F_Bit
  MOV     R0, R0, LSR #6
  MSR     SPSR_cxsf, R12         ; Set SPSR
  LDMFD   SP!, {R12, PC}^        ; Restore R12 and Return

;/**
; * @brief Activa sólo las interrupciones fiq en el registro de estado.
; */
__enable_fiq
  LDMFD   SP!, {R8, R12}         ; Load R8, SPSR
  BIC     R12, R12, #F_Bit       ; f bit = 0
  MSR     SPSR_cxsf, R12         ; Set SPSR
  LDMFD   SP!, {R12, PC}^        ; Restore R12 and Return

;/**
; * @brief Desactiva sólo las interrupciones fiq en el registro de estado.
; */
__disable_fiq
  LDMFD   SP!, {R8, R12}         ; Load R8, SPSR
  ORR     R12, R12, #F_Bit       ; f bit = 1
  MSR     SPSR_cxsf, R12         ; Set SPSR
  LDMFD   SP!, {R12, PC}^        ; Restore R12 and Return

;/**
; * @brief Activa interrupciones y fiq en el registro de estado.
; */
__enable_irq_fiq
  LDMFD   SP!, {R8, R12}           ; Load R8, SPSR
  BIC     R12, R12, #I_Bit:OR:F_Bit ; i bit = 0; f bit = 0
  MSR     SPSR_cxsf, R12           ; Set SPSR
  LDMFD   SP!, {R12, PC}^          ; Restore R12 and Return

;/**
; * @brief Desactiva interrupciones y fiq en el registro de estado.
; */
__disable_irq_fiq
```

```
  LDMFD   SP!, {R8, R12}            ; Load R8, SPSR
  ORR     R12, R12, #I_Bit:OR:F_Bit ; i bit = 1; f bit = 1
  MSR     SPSR_cxsf, R12            ; Set SPSR
  LDMFD   SP!, {R12, PC}^           ; Restore R12 and Return

;/**
; * @brief Activa sólo las interrupciones irq en el registro de estado.
; */
__enable_irq
  LDMFD   SP!, {R8, R12}            ; Load R8, SPSR
  BIC     R12, R12, #I_Bit          ; i bit = 0
  MSR     SPSR_cxsf, R12            ; Set SPSR
  LDMFD   SP!, {R12, PC}^           ; Restore R12 and Return

;/**
; * @brief Desactiva sólo las interrupciones irq en el registro de estado.
; */
__disable_irq
  LDMFD   SP!, {R8, R12}            ; Load R8, SPSR
  ORR     R12, R12, #I_Bit          ; i bit = 1
  MSR     SPSR_cxsf, R12            ; Set SPSR
  LDMFD   SP!, {R12, PC}^           ; Restore R12 and Return

                END
```

Listing 11: SWI.s

Semáforo de interrupciones

```
/**
 * @file semaforo_interrupciones.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definicion de funciones para el manejo de las
 * interrupciones
 */

#ifndef __SEMAFORO_INTERRUPCIONES__
#define __SEMAFORO_INTERRUPCIONES__

#include <LPC210x.H>
#include <inttypes.h>

#include "llamadas_sistema.h"

static uint32_t recover;

/**
 * @brief Bloquea las interrupciones
 * @retrun Flags de recuperación
 */
uint32_t bloquear_interrupciones(void);

/**
 * @brief Libera las interrupciones bloqueadas
 * @param flags Flags de recuperación
 */
void liberar_interrupciones(uint32_t flags);
```

```
#endif
```

Listing 12: semaforo_interrupciones.h

```
/**
 * @file semaforo_interrupciones.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementacion de funciones para el manejo de las
 * interrupciones
 */

#include "semaforo_interrupciones.h"

uint32_t bloquear_interrupciones() {
  uint32_t i_flag = read_IRQ_bit();
  uint32_t f_flag = read_FIQ_bit();

  disable_irq_fiq();
  return (i_flag << 1) | f_flag;
}

void liberar_interrupciones(uint32_t flags) {
  uint32_t i_flag = (flags & 0x2) >> 1;
  uint32_t f_flag = flags & 0x1;

  if (i_flag == 0 && f_flag == 0) {
    enable_irq_fiq();
  } else if (i_flag == 0 && f_flag == 1) {
    enable_irq();
  } else if (i_flag == 1 && f_flag == 0) {
    enable_fiq();
  }  // else {ya están desactivadas}
}
```

Listing 13: semaforo_interrupciones.c

Temporizadores

```
/**
 * @file temporizador.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definicion de funciones para el manejo del temporizador
 */

#ifndef TEMPORIZADOR_H
#define TEMPORIZADOR_H

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

#include "cola_asyn.h"
#include "eventos.h"
#include "msg.h"

/**
```

```
 * @brief Programa un contador para que pueda ser utilizado.
 */
void temporizador_iniciar(void);

/**
 * @brief Inicia la cuenta de un contador de forma indefinida.
 */
void temporizador_empezar(void);

/**
 * @brief Lee el tiempo que lleva contando el contador desde la última vez que
 * se ejecutó temporizador_empezar y lo devuelve en microsegundos (us).
 * @return Tiempo en microsegundos (us).
 */
uint32_t temporizador_leer(void);

/**
 * @brief Detiene el contador y devuelve el tiempo transcurrido desde
 * temporizador_empezar
 * @return Tiempo en microsegundos (us).
 */
uint32_t temporizador_parar(void);

/**
 * @brief Programa el reloj para que encole un evento periódicamente. El
   periodo
 * se indica en milisegundos (ms).
 * @param periodo Periodo en milisegundos (ms).
 */
void temporizador_reloj(int periodo);

#endif
```

Listing 14: temporizador.h

```
/**
 * @file temporizador.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementacion de funciones para el manejo del temporizador
 */

#include "temporizador.h"

void timer0_IRC(void) __irq {
  static int veces = 0;
  cola_encolar_eventos_raw(TEMPORIZADOR, ++veces, 0);
  T0IR = 1;         // Clear interrupt flag
  VICVectAddr = 0;  // Acknowledge Interrupt
}

void temporizador_iniciar() {
  T1PR = 14;           // Cuenta cada microsegundo: 15 clk = 1 us
  T1MR0 = UINT32_MAX;  // Para que cuente el máximo numero de microsegundos
                       // antes de reiniciarse

  T1MCR = 2;  // Reset on MR0
```

```
  T1TCR = T1TCR & ~0x1;
}

void temporizador_empezar() {
  T1PC = 0;
  T1TC = 0;
  T1TCR = T1TCR | 0x1;  // comienza a contar
}

uint32_t temporizador_leer() { return clock_get_us(); }

uint32_t temporizador_parar() {
  uint32_t time = T1TC;
  T1TCR = T1TCR & ~0x1;  // detiene el contador
  return time;
}

void temporizador_reloj(int periodo) {
  T0PR = 14999;  // Cuenta cada milisegundo: 15000 clk = 1 ms;
  T0MR0 = periodo - 1;

  T0MCR = 3;  // Interrumpe cada MR0 y resetea el contador

  VICIntSelect = VICIntSelect | 0x00000010;  // FIQ
  VICIntEnable = VICIntEnable | 0x00000010;  // Enable Timer0 Interrupt.

  T0TCR = T0TCR & ~0x1;
  T0PC = 0;
  T0TC = 0;
  T0TCR = T0TCR | 0x1;  // comienza a contar
}
```

Listing 15: temporizador.c

Contadores

```
/**
 * @file contadores.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de los contadores
 */

#ifndef CONTADORES_H
#define CONTADORES_H

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

#include "llamadas_sistema.h"

// ----- REAL TIME CLOCK -----

/**
 * @brief Inicializa el RTC, reseteando la cuenta, ajustando el reloj y
 *        activando el enable
 */
void RTC_init(void);
```

```
/**
 * @brief Devuelve los minutos y los segundos de juego (entre 0 y 59)
 *        en dos uint8
 * @param minutos puntero a variable dónde guardar los minutos
 * @param segundos puntero a variable dónde guardar los segundos
 */
void RTC_leer(uint32_t* minutos, uint32_t* segundos);

// ----- WATCHDOG TIMER -----

/**
 * @brief Inicializa el watchdog timer para que resetee el procesador dentro
   de
 *        sec segundos si no se le "alimenta".
 * @param sec segundos
 */
void WD_init(int sec);

/**
 * @brief Alimenta al watchdog timer
 */
void WD_feed(void);

#endif
```

Listing 16: contadores.h

```
/**
 * @file contadores.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de los contadores
 */

#include "contadores.h"

// ----- REAL TIME CLOCK -----

void RTC_init(void) {
  CCR = CCR & ~1;
                    // PCLK = 60E6 Hz / 4 = 15E6Hz
  PREINT = 456;     // PREINT = int (PCLK / 32768) - 1 = 456
  PREFRAC = 25024;  // PREFRAC = PCLK - ([PREINT + 1] * 32768) = 25024

  CCR = CCR | 3;  // Reset y enable
  CCR = CCR & ~2; // Empieza a contar
}

void RTC_leer(uint32_t* minutos, uint32_t* segundos) {
  uint32_t time = clock_gettime();
  *minutos = (time & 0x3F00) >> 8;
  *segundos = (time & 0x3F);
}

// --- WATCHDOG TIMER -----

void WD_init(int sec)  {
```

```
  WDTC = sec * 3750000;  // Set tiempo de watchdog
  // x * (1/15E6s) * 4 = sec <-> x = sec * 15E6 (1/4) = 3750000 * sec s^(-1)
  WDMOD = WDMOD | 3;  // Reset y watchdog)
}

void WD_feed() {
  uint32_t flags = bloquear_interrupciones(); /*LOCK*/
  WDFEED = 0xAA;
  WDFEED = 0x55;
  liberar_interrupciones(flags); /*UNLOCK*/
}
```

Listing 17: contadores.c

Gestor de alarmas

```
/**
 * @file g_alarmas.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de las alarmas
 */

#ifndef GESTOR_ALARMAS_H
#define GESTOR_ALARMAS_H
#include <inttypes.h>

#include "cola_msg.h"
#include "contadores.h"
#include "temporizador.h"
#include "utils.h"

enum { NUM_ALARMAS = 8 };

typedef uint32_t alarma_t;

/**
 * @brief Crea el campo auxiliar que hay que encolar junto a SET_ALARM
 *  para crear la alarma
 * @param ID_msg ID del mensaje a encolar cuando salte la alarma
 * @param esPeriodica Si TRUE la alarma se resetea cada vez que salta
 * @param retardo Tiempo en ms hasta que salte la alarma
 * @return Devuelve la información de los parámetros con el formato
 *  del campo auxiliar: 8 bits de ID : 1 bit de esPeriodica : 23 bits de
   retardo
 */
alarma_t g_alarma_crear(uint8_t ID_msg, int esPeriodica, uint32_t retardo);

/**
 * @brief Crea el campo auxiliar que hay que encolar junto a SET_ALARM
 *  para borrar la alarma
 * @param ID_msg ID del mensaje de la alarma a desprogramar
 * @return Devuelve el campo auxiliar de la alarma programado a 0ms
 */
alarma_t g_alarma_borrar(uint8_t ID_msg);

/**
 * @brief Obtiene el retardo de la alarma
```

```
 * @param alarma Alarma con formato de campo auxiliar
 * @return Devuelve el retardo de la alarma
 */
uint32_t g_alarma_retardo(alarma_t alarma);

/**
 * @brief Comprueba que la alarma es reiniciada cada vez que salta
 * @param alarma Alarma con formato de campo auxiliar
 * @return TRUE si la alarma es periódica
 */
int g_alarma_es_periodica(alarma_t alarma);

/**
 * @brief Obtiene el mensaje a encolar cuando salte la alarma
 * @param alarma Alarma con formato de campo auxiliar
 * @return Devuelve el identificador del mensaje a encolar cuando
 *  salte la alarma
 */
uint8_t g_alarma_id_msg(alarma_t alarma);

/**
 * @brief Iinicia el gestor de alarmas y los temporizadores
 */
void g_alarma_iniciar(void);

/**
 * @brief Si hay espacio, añade la alarma al conjunto de alarmas.
 *  Si la alarma ya estaba programada la reprograma con la nueva información.
 *  Si el retardo de la alarma es de 0ms la borra del conjunto.
 * @param alarma Alarma con formato de campo auxiliar
 */
void g_alarma_programar(alarma_t alarma);

/**
 * @brief Cuenta 1ms para cada alarma y comprueba las que puede disparar.
 *  Encola los mensajes de las alarmas disparadas.
 */
void g_alarma_comprobar_alarmas(void);

/**
 * @brief Tratamiento de eventos del módulo del gestor de alarmas
 * @param evento Evento a tratar
 */
void g_alarma_tratar_evento(evento_t evento);

/**
 * @brief Tratamiento de mensajes del módulo del gestor de alarmas
 * @param mensaje Mensaje a tratar
 */
void g_alarma_tratar_mensaje(msg_t mensaje);

#endif
```

Listing 18: g_alarmas.h

```
/**
 * @file g_alarmas.c
 * @authors: Fernando Lahoz & Héctor Toral
```

```
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de las alarmas
 */

#include "g_alarmas.h"

// Vector de alarmas
static alarma_t alarmas[NUM_ALARMAS];
static uint32_t tiempo[NUM_ALARMAS];

alarma_t g_alarma_borrar(uint8_t ID_msg) {
  return ID_msg << 24;  // El resto de campos a 0 para borrarla
}

alarma_t g_alarma_crear(uint8_t ID_msg, int esPeriodica, uint32_t retardo) {
  alarma_t alarma =
      ID_msg << 24;   // Los 8 bits más significativos son el msg asociado
  if (esPeriodica) {  // Activar bit 23
    alarma = alarma | 0x800000;
  }
  alarma = alarma | (retardo & 0x7FFFFF);
  return alarma;
}

alarma_t g_alarma_retardo(alarma_t alarma) { return alarma & 0x7FFFFF; }

int g_alarma_es_periodica(alarma_t alarma) { return (alarma & 0x800000) != 0;
   }

uint8_t g_alarma_id_msg(alarma_t alarma) { return alarma >> 24; }

void g_alarma_iniciar() {
  temporizador_reloj(1);
  temporizador_iniciar();
  temporizador_empezar();
}

void g_alarma_programar(alarma_t alarma) {
  // Buscar si hay que reprogramarla
  for (int i = 0; i < NUM_ALARMAS; i++) {
    if (alarmas[i] == 0)
      continue;
    else if ((g_alarma_id_msg(alarmas[i]) == g_alarma_id_msg(alarma))) {
      if (g_alarma_retardo(alarma) == 0) {
        alarmas[i] = 0;
        return;
      } else {
        alarmas[i] = alarma;
        tiempo[i] = g_alarma_retardo(alarma);
        return;
      }
    }
  }

  if (g_alarma_retardo(alarma) == 0) return;

  // Buscar el primer hueco libre
  for (int i = 0; i < NUM_ALARMAS; i++) {
```

```
    if (alarmas[i] == 0) {
      alarmas[i] = alarma;
      tiempo[i] = g_alarma_retardo(alarma);
      return;
    }
  }
}

void g_alarma_comprobar_alarmas() {
  for (int i = 0; i < NUM_ALARMAS; i++) {
    if (alarmas[i] == 0 || --tiempo[i] > 0) continue;
    cola_encolar_msg(g_alarma_id_msg(alarmas[i]), 0);
    if (!g_alarma_es_periodica(alarmas[i])) {
      alarmas[i] = 0;  // cancelar
    } else {
      tiempo[i] = g_alarma_retardo(alarmas[i]);  // reset
    }
  }
}

void g_alarma_tratar_evento(evento_t evento) {
  switch (evento.ID_evento) {
    case TEMPORIZADOR:
      g_alarma_comprobar_alarmas();
      break;
  }
}

void g_alarma_tratar_mensaje(msg_t mensaje) {
  switch (mensaje.ID_msg) {
    case RESET:
      temporizador_empezar();
      break;
    case SET_ALARM:
      g_alarma_programar(mensaje.auxData);
      break;
    case FIN:
      temporizador_parar();
      break;
  }
}
```

Listing 19: g_alarmas.c

Gestor de estadísticas

```
/**
 * @file g_estadisticas.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de las estadísticas
 */

#ifndef G_ESTADISTICAS_H
#define G_ESTADISTICAS_H

#include "cola_msg.h"
#include "contadores.h"
```

```h
#include "msg.h"
#include "temporizador.h"

enum {
  G_ESTADISTICAS_INACTIVO,
  G_ESTADISTICAS_ACTIVO,
  G_ESTADISTICAS_ESPERANDO
};

/**
 * @brief Función para el cálculo de las estadísticas de la partida
 * @param minutos_total Minutos totales acumulados en la partida
 * @param segundos_total Segundos totales acumulados en la partida
 * @param minutos_dif Minutos de diferencia entre el juego actual y el
   anterior
 * @param segundos_dif Segundos de diferencia entre el juego actual y el
 * anterior
 */
void g_estadisticas_partida(uint32_t *minutos_total, uint32_t *segundos_total,
                            int8_t *minutos_dif, int8_t *segundos_dif);

/**
 * @brief Función para el cálculo de las estadísticas de los mensajes
 * @param tiempo_total Tiempo total acumulado en la partida
 * @param num_mensajes Número de mensajes enviados en la partida
 * @param minutos_total Minutos totales acumulados en la partida
 * @param segundos_total Segundos totales acumulados en la partida
 */
void g_estadisticas_mensaje(uint32_t *tiempo_total, uint32_t *num_mensajes,
                            uint32_t *minutos_total, uint32_t *segundos_total)
   ;

/**
 * @brief Tratamiento de mensajes del módulo del gestor de estadísticas
 * @param mensaje Mensaje a tratar
 */
void g_estadisticas_tratar_mensaje(msg_t mensaje);

#endif
```

Listing 20: g_estadisticas.h

```c
/**
 * @file g_estadisticas.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de las estadí
   sticas
 */

#include "g_estadisticas.h"

void g_estadisticas_partida(uint32_t *minutos_total, uint32_t *segundos_total,
                            int8_t *minutos_dif, int8_t *segundos_dif) {
  int8_t minutos_aux = *minutos_total;
  int8_t segundos_aux = *segundos_total;
  RTC_leer(minutos_total, segundos_total);
  *minutos_dif = *minutos_total - minutos_aux;
```

```
  *segundos_dif = *segundos_total - segundos_aux;
  if (*segundos_dif < 0) {
    *segundos_dif = *segundos_total + (60 - segundos_aux);
    *minutos_dif--;
  }
}

void g_estadisticas_mensaje(uint32_t *tiempo_total, uint32_t *num_mensajes,
                            uint32_t *minutos_total, uint32_t *segundos_total)
    {
  RTC_leer(minutos_total, segundos_total);
  *tiempo_total = 0;
  *num_mensajes = 0;
  RTC_init();
}

void g_estadisticas_tratar_mensaje(msg_t mensaje) {
  static uint32_t tiempo_total = 0, num_mensajes = 0;
  static uint32_t minutos_total = 0, segundos_total = 0;
  static int8_t minutos_dif = 0, segundos_dif = 0;
  static uint8_t estado = G_ESTADISTICAS_INACTIVO;

  int tiempo_msg = temporizador_leer() - mensaje.timestamp;
  if (tiempo_msg > 0) {
    num_mensajes++;
    tiempo_total = tiempo_total + tiempo_msg;
  }

  switch (mensaje.ID_msg) {
    case FIN:
      if (estado == G_ESTADISTICAS_ACTIVO) {
        estado = G_ESTADISTICAS_INACTIVO;
        g_estadisticas_partida(&minutos_total, &segundos_total, &minutos_dif,
                               &segundos_dif);
      }
      break;
    case CELDA_MARCADA:
      estado = G_ESTADISTICAS_ESPERANDO;
      break;
    case JUGADA_REALIZADA:
      estado = G_ESTADISTICAS_ACTIVO;
      break;
    case CANCELAR:  // los botones y NEW comienzan el juego
      if (estado == G_ESTADISTICAS_ESPERANDO) {
        estado = G_ESTADISTICAS_ACTIVO;
      } else if (estado == G_ESTADISTICAS_INACTIVO) {
        g_estadisticas_mensaje(&tiempo_total, &num_mensajes, &minutos_total,
                               &segundos_total);
        estado = G_ESTADISTICAS_ACTIVO;
      }
      break;
    case RESET:
      if (estado != G_ESTADISTICAS_ESPERANDO) {
        g_estadisticas_mensaje(&tiempo_total, &num_mensajes, &minutos_total,
                               &segundos_total);
        estado = G_ESTADISTICAS_ACTIVO;
      }
      break;
```

```
    case PEDIR_SEGUNDOS_JUGADOS:
      cola_encolar_msg(SEGUNDOS_JUGADOS, segundos_dif);
      break;
    case PEDIR_MINUTOS_JUGADOS:
      cola_encolar_msg(MINUTOS_JUGADOS, minutos_dif);
      break;
    case PEDIR_CALIDAD_SERVICIO:
      cola_encolar_msg(CALIDAD_SERVICIO, tiempo_total / num_mensajes);
      break;
  }
}
```

Listing 21: g_estadisticas.c

Gestor de entrada y salida

```
/**
 * @file g_io.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de la entrada/salida
 */

#ifndef GESTOR_IO_H
#define GESTOR_IO_H

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

#include "cola_asyn.h"
#include "cola_msg.h"
#include "g_alarmas.h"
#include "gpio.h"
#include "utils.h"

enum { G_IO_FIN_BLOQUEADO, G_IO_FIN_LIBRE };

/**
 * @brief Inicializa el gestor de IO.
 */
void g_io_iniciar(void);

/**
 * @brief Indicar que la jugada ha sido realizada.
 */
void g_io_encender_realizada(void);

/**
 * @brief Apagar indicador de jugada realizada.
 */
void g_io_apagar_realizada(void);

/**
 * @brief Indicar jugada no válida.
 */
void g_io_mostrar_invalido(void);

/**
```

```
 * @brief Apagar indicador de jugada no válida.
 */
void g_io_apagar_invalido(void);

/**
 * @brief Indicar fin de partida.
 */
void g_io_mostrar_fin(void);

/**
 * @brief Apagar indicador fin de partida.
 */
void g_io_apagar_fin(void);

/**
 * @brief Indicar overflow
 */
void g_io_overflow(void);

/**
 * @brief Alternar led indicador de modo idle
 */
void g_io_latido(void);

/**
 * @brief Apagar indicador de modo idle
 */
void g_io_apagar_latido(void);

/**
 * @brief Tratamiento de eventos del módulo del gestor de entrada y salida
 * @param evento Evento a tratar
 */
void g_io_tratar_evento(evento_t evento);

/**
 * @brief Tratamiento de mensajes del módulo del gestor de entrada y salida
 * @param mensaje Mensaje a tratar
 */
void g_io_tratar_mensaje(msg_t mensaje);

#endif
```

Listing 22: g_io.h

```
/**
 * @file g_io.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de la entrada/
   salida
 */

#include "g_io.h"

void g_io_iniciar() {
  gpio_iniciar();
  gpio_marcar_salida(0, 32);
```

```
  // todos los pines a 0
  gpio_escribir(0, 32, 0);
  // menos los de los botones (evitar interrupción al comienzo)
  gpio_escribir(14, 2, 3);
  gpio_marcar_entrada(14, 2);
}

void g_io_encender_realizada() {
  gpio_escribir(16, 1, 1);
  // Al cabo de 2s apagar
  cola_encolar_msg(SET_ALARM, g_alarma_crear(APAGAR_REALIZADA, FALSE, 2000));
}

void g_io_apagar_realizada() { gpio_escribir(16, 1, 0); }

void g_io_mostrar_invalido() { gpio_escribir(17, 1, 1); }

void g_io_apagar_invalido() { gpio_escribir(17, 1, 0); }

void g_io_mostrar_fin() { gpio_escribir(18, 1, 1); }

void g_io_apagar_fin() { gpio_escribir(18, 1, 0); }

void g_io_mostrar_ignore_cmd() { gpio_escribir(19, 1, 1); }

void g_io_apagar_ignore_cmd() { gpio_escribir(19, 1, 0); }

void g_io_overflow() { gpio_escribir(30, 1, 1); }

void g_io_latido() {
  static uint8_t ON = 0;
  gpio_escribir(31, 1, ON);
  ON = 1 - ON;  // alternar estado
}

void g_io_apagar_latido() { gpio_escribir(31, 1, 0); }

void g_io_tratar_evento(evento_t evento) {
  switch (evento.ID_evento) {
    case OVERFLOW_E:
      g_io_overflow();
      while (1)
        ;
  }
}

void g_io_tratar_mensaje(msg_t mensaje) {
  static uint8_t estado = G_IO_FIN_LIBRE;
  switch (mensaje.ID_msg) {
    case JUGADA_REALIZADA:
      g_io_encender_realizada();
      estado = G_IO_FIN_LIBRE;
      break;
    case CANCELAR:
      estado = G_IO_FIN_LIBRE;
    case RESET:
      g_io_apagar_fin();
      break;
```

```
    case CELDA_MARCADA:
      estado = G_IO_FIN_BLOQUEADO;
      break;
    case LATIDO:
      g_io_latido();
      break;
    case APAGAR_LATIDO:
      g_io_apagar_latido();
      break;
    case ENTRADA_VALIDADA:
      if (mensaje.auxData)
        g_io_apagar_invalido();
      else
        g_io_mostrar_invalido();
      break;
    case APAGAR_REALIZADA:
      g_io_apagar_realizada();
      break;
    case IGNORE_CMD:
      g_io_mostrar_ignore_cmd();
      break;
    case APAGAR_IGNORE_CMD:
      g_io_apagar_ignore_cmd();
      break;
    case FIN:
      if (estado == G_IO_FIN_LIBRE) {
        g_io_mostrar_fin();
      }
      break;
    case OVERFLOW_M:
      g_io_overflow();
      while (1)
        ;
  }
}
```

Listing 23: g_io.c

Control de GPIO

```
/**
 * @file gpio.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo del GPIO
 */

#ifndef GPIO_H
#define GPIO_H

#include <LPC210X.H>
#include <inttypes.h>

/**
 * @brief Permite emplear el GPIO y debe ser invocada antes de poder llamar
 * al resto de funciones de la biblioteca.
 */
void gpio_iniciar(void);
```

```c
/**
 * @brief Leer del GPIO.
 * @param bit_inicial Indica el primer bit a leer.
 * @param num_bits    Indica cuántos bits queremos leer.
 * @return Entero con el valor de los bits indicados.
 */
int gpio_leer(int bit_inicial, int num_bits);

/**
 * @brief Escribir en el GPIO.
 * @param bit_inicial Indica el primer bit a escribir.
 * @param num_bits    Indica cuántos bits queremos escribir.
 */
void gpio_escribir(int bit_inicial, int num_bits, int valor);

/**
 * @brief Los bits indicados se utilizarán como pines de entrada.
 * @param bit_inicial Indica el primer bit de entrada.
 * @param num_bits    Indica cuántos bits queremos introducir.
 */
void gpio_marcar_entrada(int bit_inicial, int num_bits);

/**
 * @brief Los bits indicados se utilizarán como pines de salida.
 * @param bit_inicial Indica el primer bit de salida.
 * @param num_bits    Indica cuántos bits queremos introducir.
 */
void gpio_marcar_salida(int bit_inicial, int num_bits);

#endif
```

Listing 24: gpio.h

```c
/**
 * @file gpio.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo del GPIO
 */

#include "gpio.h"

void gpio_iniciar(void) {
  // Limpiamos los pines
  IODIR = 0xFFFFFFFF;
  IOCLR = 0xFFFFFFFF;
}

int gpio_leer(int bit_inicial, int num_bits) {
  uint32_t mask = ((1 << num_bits) - 1) << bit_inicial;
  return (IOPIN & mask) >> bit_inicial;
}

void gpio_escribir(int bit_inicial, int num_bits, int valor) {
  uint32_t maskON = (valor & ((1 << num_bits) - 1)) << bit_inicial;
  uint32_t maskOFF = (~valor & ((1 << num_bits) - 1)) << bit_inicial;
  IOSET = maskON;   // escribe los 1s
```

```
  IOCLR = maskOFF;  // escribe los 0s
}

void gpio_marcar_entrada(int bit_inicial, int num_bits) {
  uint32_t mask = ((1 << num_bits) - 1) << bit_inicial;
  IODIR = IODIR & ~mask;
}

void gpio_marcar_salida(int bit_inicial, int num_bits) {
  uint32_t mask = ((1 << num_bits) - 1) << bit_inicial;
  IODIR = IODIR | mask;
}
```

Listing 25: gpio.c

Botones

```
/**
 * @file botones.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de los botones
 */

#ifndef BOTONES_H
#define BOTONES_H

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

#include "cola_asyn.h"
#include "eventos.h"
#include "g_alarmas.h"
#include "msg.h"
#include "utils.h"

/**
 * @brief Prepara la configuración de las interrupciones para los
 *  botones 1 y 2.
 */
void botones_iniciar(void);

/**
 * @brief Resetea la configuracion de la interrupción para el boton 1.
 */
void boton1_reset(void);

/**
 * @brief Resetea la configuracion de la interrupción para el boton 2.
 */
void boton2_reset(void);

/**
 * @brief Indica si el boton 1 está pulsado.
 * @return true(1) si pulsado; false(0) en caso contrario.
 */
int boton1_pulsado(void);
```

```
/**
 * @brief Indica si el boton 2 está pulsado.
 * @return true(1) si pulsado; false(0) en caso contrario.
 */
int boton2_pulsado(void);

#endif
```

Listing 26: botones.h

```
/**
 * @file botones.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de los botones
 */

#include "botones.h"

void boton1_IRC(void) __irq {
  static int veces1 = 0;

  VICIntEnClr = 0x00008000;  // deshabilita eint1
  EXTINT = EXTINT | 0x2;     // borra el flag de interrupcion
  VICVectAddr = 0;           // Acknowledge Interrupt

  cola_encolar_eventos(PULSACION, ++veces1, 1);
}

void boton2_IRC(void) __irq {
  static int veces2 = 0;

  VICIntEnClr = 0x00010000;  // deshabilita eint2
  EXTINT = EXTINT | 0x4;     // borra el flag de interrupcion
  VICVectAddr = 0;           // Acknowledge Interrupt

  cola_encolar_eventos(PULSACION, ++veces2, 2);
}

void botones_iniciar(void) {
  PINSEL0 = PINSEL0 | ((uint32_t)0xA << 28);  // 10 10 ...

  VICVectAddr1 = (unsigned long)boton1_IRC;
  // bit 5 enable
  // bits 4:0 son el indice del dispositivo EINT1 (15)
  VICVectCntl1 = (VICVectCntl1 & ~0x1f) | 0x2F;

  VICVectAddr2 = (unsigned long)boton2_IRC;
  // bit 5 enable
  // bits 4:0 son el indice del dispositivo EINT2 (16)
  VICVectCntl2 = (VICVectCntl2 & ~0x1f) | 0x30;

  VICIntEnable = VICIntEnable | 0x18000;  // Enable EINT[1,2] Interrupt.
}

void boton1_reset(void) {
  EXTINT = EXTINT | 0x2;                    // borra el flag de interrupcion
  VICIntEnable = VICIntEnable | 0x00008000;  // Enable EINT1 Interrupt.
```

```
}

void boton2_reset(void) {
  EXTINT = EXTINT | 0x4;                    // borra el flag de interrupcion
  VICIntEnable = VICIntEnable | 0x00010000;  // Enable EINT2 Interrupt.
}

int boton1_pulsado(void) {
  EXTINT = EXTINT | 0x2;  // borra el flag de interrupcion
  // si aqui se vuelve a activar entonces el boton está pulsado
  return (EXTINT & 0x2) != 0;
}

int boton2_pulsado(void) {
  EXTINT = EXTINT | 0x4;  // borra el flag de interrupcion
  // si aqui se vuelve a activar entonces el boton está pulsado
  return (EXTINT & 0x4) != 0;
}
```

Listing 27: botones.c

Gestor de botones

```
/**
 * @file g_boton.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de los botones
 */

#ifndef G_BOTONES_H
#define G_BOTONES_H

#include "cola_asyn.h"
#include "cola_msg.h"
#include "eventos.h"
#include "msg.h"

/**
 * @brief Tratamiento de eventos del módulo del gestor de botones.
 * @param evento Evento a tratar.
 */
void g_boton_tratar_evento(evento_t evento);

/**
 * @brief Tratamiento de mensajes del módulo del gestor de botones.
 * @param mensaje Mensaje a tratar.
 */
void g_boton_tratar_mensaje(msg_t mensaje);

/**
 * @brief Inicialización del módulo del gestor de botones.
 */
void g_boton_iniciar(void);

#endif
```

Listing 28: g_boton.h

```c
/**
 * @file g_boton.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de los botones
 */

#include "botones.h"

void g_boton_tratar_evento(evento_t evento) {
  switch (evento.ID_evento) {
    case PULSACION:
      if (evento.auxData == 1) {
        cola_encolar_msg(SET_ALARM,
                         g_alarma_crear(BAJAR_PULSACION_1, TRUE, 10));
        cola_encolar_msg(EJECUTAR, CANCELAR);
      } else if (evento.auxData == 2) {
        cola_encolar_msg(SET_ALARM,
                         g_alarma_crear(BAJAR_PULSACION_2, TRUE, 10));
        cola_encolar_msg(EJECUTAR, RESET);
      }
      break;
  }
}

void g_boton_tratar_mensaje(msg_t mensaje) {
  switch (mensaje.ID_msg) {
    case BAJAR_PULSACION_1:
      if (!boton1_pulsado()) {
        boton1_reset();
        cola_encolar_msg(SET_ALARM, g_alarma_borrar(BAJAR_PULSACION_1));
      }
      break;
    case BAJAR_PULSACION_2:
      if (!boton2_pulsado()) {
        boton2_reset();
        cola_encolar_msg(SET_ALARM, g_alarma_borrar(BAJAR_PULSACION_2));
      }
      break;
  }
}

void g_boton_iniciar(void) { botones_iniciar(); }
```

Listing 29: g_boton.c

Gestor de línea serie

```c
/**
 * @file g_serie.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de la comunicación
   serie
 */

#ifndef G_SERIE_H
#define G_SERIE_H
```

```
#include <LPC210X.H>
#include <inttypes.h>

#include "UART0.h"
#include "cadenas.h"
#include "celda.h"
#include "cola_asyn.h"
#include "cola_msg.h"
#include "conecta4_2022.h"
#include "eventos.h"
#include "g_alarmas.h"
#include "msg.h"
#include "utils.h"

enum {
  COLA_CADENAS_SIZE = 32,

  BUFFER_SIZE = 3,
  NUM_COMANDOS = 2,

  G_SERIE_ACTIVO = 0,
  G_SERIE_INACTIVO = 1,
  G_SERIE_ESPERANDO = 2
};

/**
 * @brief Encola el identificador de una cadena para ser mostrada.
 * @param cadena Identificador de la cadena a mostrar.
 */
void g_serie_encolar_cadena(uint8_t cadena);

/**
 * @brief Desencola el identificador de una cadena para mostrar la siguiente.
 */
void g_serie_desencolar_cadena(void);

/**
 * @brief Muestra la cadena correspondiente al identificador.
 * @param cadena Identificador de la cadena a mostrar.
 */
void g_serie_mostrar_cadena(uint8_t cadena);

/**
 * @brief Comprueba que el comando del buffer corresponde a #Cx! y en
 *        caso afirmativo lo ejecuta.
 * @param buffer Comando a ejecutar
 */
int g_serie_check_c(char buffer[BUFFER_SIZE]);

/**
 * @brief Comprueba que el comando del buffer corresponde a alguno de los
 *        disponibles y en caso afirmativo lo ejecuta.
 * @param buffer Comando a ejecutar
 */
void g_serie_ejecutar_cmd(char buffer[BUFFER_SIZE]);

/**
```

```
 * @brief Sobreescribe todos los caracteres del buffer por '\0'.
 * @param buffer Buffer de almacenamiento de comandos.
 */
void g_serie_clean_buffer(char buffer[BUFFER_SIZE]);

/**
 * @brief Función llamada cuando se recibe un caracter. Lo añade
 *        al buffer de comandos.
 * @param c Caracter recibido.
 */
void g_serie_caracter_recibido(char c);

/**
 * @brief Codifica el valor de celda como un caracter.
 * @param celda Valor de celda a codificar.
 * @return Valor codificado como caracter.
 */
char g_serie_codificar_jugador(CELDA celda);

/**
 * @brief Muestra una fila con los datos de sus celdas.
 * @param datosFila Datos de las celdas de una fila.
 */
void g_serie_mostrar_fila(uint32_t datosFila);

/**
 * @brief Escribe el numero x como caracteres de texto en el array,
 *        alineado a la derecha a partir de la posición i.
 * @param array Cadena donde se escribe el número.
 * @param i Posición final (posición de las unidades).
 * @param x Número a escribir.
 */
void g_serie_itoa(char array[], uint32_t i, uint32_t x);

/**
 * @brief Muestra la latencia media de transmisión de mensajes.
 * @param latencia Tiempo medio de transmisión de mensajes (us).
 */
void g_serie_mostrar_qos(uint32_t latencia);

/**
 * @brief Muestra los minutos transcurridos de la partida.
 * @param minutos Minutos transcurridos en la partida.
 */
void g_serie_mostrar_minutos(uint32_t minutos);

/**
 * @brief Muestra los segundos transcurridos de la partida.
 * @param segundos Segundos transcurridos en la partida.
 */
void g_serie_mostrar_segundos(uint32_t segundos);

/**
 * @brief Encola los identificadores de las cadenas que componen el tablero.
 */
void g_serie_encolar_tablero(void);

/**
```

```
 * @brief Encola los identificadores de las cadenas que componen el
 *        mensaje de inicio.
 */
void g_serie_encolar_inicio(void);

/**
 * @brief Encola los identificadores de las cadenas que componen el
 *        mensaje de comienzo.
 */
void g_serie_encolar_comenzar(void);

/**
 * @brief Inicia el funcionamiento del gestor.
 */
void g_serie_iniciar(void);

/**
 * @brief Tratamiento de eventos del módulo del gestor serie
 * @param evento Evento a tratar
 */
void g_serie_tratar_evento(evento_t evento);

/**
 * @brief Tratamiento de mensajes del módulo del gestor serie
 * @param mensaje Mensaje a tratar
 */
void g_serie_tratar_mensaje(msg_t mensaje);

#endif
```

Listing 30: g_serie.h

```
/**
 * @file g_serie.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de la comunicación
 * por serie
 */

#include "g_serie.h"

static uint8_t first = 0, last = 0, full = FALSE;
static uint8_t cola_cadenas[COLA_CADENAS_SIZE];

void g_serie_encolar_cadena(uint8_t cadena) {
  if (full) return;

  if (first == last) {  // Si está vacía antes de encolar se muestra
    g_serie_mostrar_cadena(cadena);
  }

  cola_cadenas[last] = cadena;

  last++;
  if (last == COLA_CADENAS_SIZE) {
    last = 0;
  }
```

```
  if (last == first) {
    full = TRUE;
  }
}

void g_serie_desencolar_cadena(void) {
  if (first == last && !full) return;  // Si está vacía no hace nada

  first++;
  if (first == COLA_CADENAS_SIZE) {
    first = 0;
  }
  full = FALSE;

  if (first != last) {  // Si no está vacía una vez desencolado se muestra
    uint8_t cadena = cola_cadenas[first];
    g_serie_mostrar_cadena(cadena);
  }
}

void g_serie_mostrar_cadena(uint8_t cadena) {
  if (cadena >= CADENA_FILA1 && cadena <= CADENA_FILA6) {  // Caso de una fila
    cola_encolar_msg(PEDIR_FILA, cadena - CADENA_FILA1 + 1);
  } else if (cadena == CADENA_CALIDAD_SERVICIO) {
    cola_encolar_msg(PEDIR_CALIDAD_SERVICIO, 0);
  } else if (cadena == CADENA_MINUTOS_JUGADOS) {
    cola_encolar_msg(PEDIR_MINUTOS_JUGADOS, 0);
  } else if (cadena == CADENA_SEGUNDOS_JUGADOS) {
    cola_encolar_msg(PEDIR_SEGUNDOS_JUGADOS, 0);
  } else {
    uart0_enviar_array(cadenas[cadena]);
  }
}

int g_serie_check_c(char buffer[BUFFER_SIZE]) {
  if (buffer[0] == 'C' && buffer[1] >= '0' && buffer[1] <= '9') {
    if (buffer[2] == '\0') {
      uint32_t unidades = buffer[1] - '0';
      cola_encolar_msg(JUGAR, unidades);
      return TRUE;
    } else if (buffer[2] >= '0' && buffer[2] <= '9') {
      uint32_t decenas = buffer[1] - '0';
      uint32_t decenasX10 = (decenas << 3) + (decenas << 1);  // 10x = 8x + 2x
      uint32_t unidades = buffer[2] - '0';
      cola_encolar_msg(JUGAR, decenasX10 + unidades);
      return TRUE;
    }
  }
  return FALSE;
}

void g_serie_ejecutar_cmd(char buffer[BUFFER_SIZE]) {
  const char* cmd[NUM_COMANDOS] = {"NEW", "END"};
  const uint8_t msg[NUM_COMANDOS] = {RESET, FIN};

  if (g_serie_check_c(buffer)) return;

  for (uint8_t i = 0; i < NUM_COMANDOS; i++) {
```

```
    int iguales = TRUE;
    for (uint8_t j = 0; j < BUFFER_SIZE && iguales; j++) {
      if (buffer[j] != cmd[i][j]) {
        iguales = FALSE;
      } else if (cmd[i][j] == '\0') {
        break;
      }
    }
    if (iguales) {
      cola_encolar_msg(msg[i], 0);
      return;
    }
  }
  cola_encolar_msg(IGNORE_CMD, 0);
}

void g_serie_clean_buffer(char buffer[BUFFER_SIZE]) {
  for (uint8_t i = 0; i < BUFFER_SIZE; i++) {
    buffer[i] = '\0';
  }
}

void g_serie_caracter_recibido(char c) {
  static char buffer[BUFFER_SIZE];
  static uint8_t i = 0, k = 0, leer = FALSE;

  if (c == '#') {  // Comienzo de comando
    cola_encolar_msg(APAGAR_IGNORE_CMD, 0);
    leer = TRUE;
    i = 0;
    k = 0;
    g_serie_clean_buffer(buffer);
  } else if (leer && c == '!') {  // Fin de comando
    leer = FALSE;
    k = 0;
    g_serie_ejecutar_cmd(buffer);
  } else if (leer) {  // Caracter del comando
    if (i >= 3)
      leer = FALSE;
    else
      buffer[i++] = c;
  }
  k++;
  if (k > 5) {
    cola_encolar_msg(IGNORE_CMD, 0);
  }
}

char g_serie_codificar_jugador(CELDA celda) {
  if (celda_vacia(celda)) {
    return ' ';
  } else if (celda_blanca(celda)) {
    return 'B';
  } else if (celda_negra(celda)) {
    return 'N';
  }  // celda_fijada(celda)
  return '*';
}
```

```
void g_serie_mostrar_fila(uint32_t datosFila) {
  char array_fila[BUFFER_ENVIO_SIZE + 1] = "      x| | | | | | | |\n";

  int fila = datosFila & 0xF;
  array_fila[6] = '0' + fila;

  datosFila = datosFila >> 4;

  for (int i = 8; i <= 20; i += 2) {
    array_fila[i] = g_serie_codificar_jugador(datosFila & 0xF);

    datosFila = datosFila >> 4;
  }

  uart0_enviar_array(array_fila);
}

void g_serie_itoa(char array[], uint32_t i, uint32_t x) {
  if (x == 0) {
    array[i] = '0';
  } else {
    while (x > 0) {
      array[i] = '0' + (x % 10);
      x = x / 10;
      i--;
    }
  }
}

void g_serie_mostrar_qos(uint32_t latencia) {
  char array[BUFFER_ENVIO_SIZE + 1] = "Latencia:                  Xus\n\n";
  g_serie_itoa(array, 26 /*Posicion de la X*/, latencia);
  uart0_enviar_array(array);
}

void g_serie_mostrar_minutos(uint32_t minutos) {
  char array[BUFFER_ENVIO_SIZE + 1] = "Tiempo jugado:         Xm";
  g_serie_itoa(array, 23 /*Posicion de la X*/, minutos);
  uart0_enviar_array(array);
}

void g_serie_mostrar_segundos(uint32_t segundos) {
  char array[BUFFER_ENVIO_SIZE + 1] = "  Ys\n";
  g_serie_itoa(array, 2 /*Posicion de la Y*/, segundos);
  uart0_enviar_array(array);
}

void g_serie_encolar_tablero(void) {
  for (uint8_t c = CADENA_FILA6; c >= CADENA_FILA1; c--) {
    g_serie_encolar_cadena(c);
  }
  g_serie_encolar_cadena(CADENA_BASE1);
  g_serie_encolar_cadena(CADENA_BASE2);
}

void g_serie_encolar_inicio(void) {
  for (uint8_t c = CADENA_CABECERA1; c <= CADENA_CABECERA20; c++) {
```

```
    g_serie_encolar_cadena(c);
  }
}

void g_serie_encolar_comenzar(void) {
  for (uint8_t c = CADENA_COMENZAR1; c <= CADENA_COMENZAR4; c++) {
    g_serie_encolar_cadena(c);
  }
}

void g_serie_iniciar(void) {
  uart0_iniciar();
  g_serie_encolar_inicio();
}

void g_serie_tratar_evento(evento_t evento) {
  switch (evento.ID_evento) {
    case CARACTER_RECIBIDO:
      g_serie_caracter_recibido(evento.auxData);
      // Avisamos al g_energia de que resetee la alarma de Power-Down
      cola_encolar_msg(RESET_POWERDOWN, 0);
      break;
    case CARACTER_ENVIADO:
      if (!uart0_continuar_envio()) {
        g_serie_desencolar_cadena();
      }
      break;
  }
}

void g_serie_tratar_mensaje(msg_t mensaje) {
  static uint8_t estado = G_SERIE_INACTIVO;
  switch (mensaje.ID_msg) {
    case DEVOLVER_FILA:
      g_serie_mostrar_fila(mensaje.auxData);
      break;
    case CELDA_MARCADA:
      estado = G_SERIE_ESPERANDO;
      g_serie_encolar_tablero();
      g_serie_encolar_cadena(CADENA_CANCELAR1);
      g_serie_encolar_cadena(CADENA_CANCELAR2);
      break;
    case JUGADA_REALIZADA:
      g_serie_encolar_tablero();
      cola_encolar_msg(PEDIR_JUGADOR, 0);
      estado = G_SERIE_ACTIVO;
      break;
    case JUGADOR:
      if (estado == G_SERIE_ACTIVO) {
        if (mensaje.auxData == FICHA_BLANCA)
          g_serie_encolar_cadena(CADENA_TURNO_BLANCAS);
        else if (mensaje.auxData == FICHA_NEGRA)
          g_serie_encolar_cadena(CADENA_TURNO_NEGRAS);
      }
      break;
    case CALIDAD_SERVICIO:
      g_serie_mostrar_qos(mensaje.auxData);
      break;
```

```
    case MINUTOS_JUGADOS:
      g_serie_mostrar_minutos(mensaje.auxData);
      break;
    case SEGUNDOS_JUGADOS:
      g_serie_mostrar_segundos(mensaje.auxData);
      break;
    case ENTRADA_VALIDADA:
      if (!mensaje.auxData) {  // Si no es valida
        g_serie_encolar_cadena(CADENA_COLUMNA_NO_VALIDA);
      }
      break;
    case CANCELAR:
      if (estado == G_SERIE_INACTIVO) {
        estado = G_SERIE_ACTIVO;
        g_serie_encolar_tablero();
        cola_encolar_msg(PEDIR_JUGADOR, 0);
      } else if (estado == G_SERIE_ESPERANDO) {
        estado = G_SERIE_ACTIVO;
        g_serie_encolar_cadena(CADENA_CANCELADO);
        g_serie_encolar_tablero();
        cola_encolar_msg(PEDIR_JUGADOR, 0);
      }
      break;
    case RESET:
      if (estado != G_SERIE_ESPERANDO) {
        if (estado == G_SERIE_ACTIVO) {
          g_serie_encolar_cadena(CADENA_RESET);
        }
        estado = G_SERIE_ACTIVO;
        g_serie_encolar_tablero();
        cola_encolar_msg(PEDIR_JUGADOR, 0);
      }
      break;
    case FIN:
      if (estado == G_SERIE_ACTIVO) {
        estado = G_SERIE_INACTIVO;
        g_serie_encolar_cadena(CADENA_FIN);
        if (mensaje.auxData == FICHA_BLANCA) {
          g_serie_encolar_cadena(CADENA_GANAN_BLANCAS);
        } else if (mensaje.auxData == FICHA_NEGRA) {
          g_serie_encolar_cadena(CADENA_GANAN_NEGRAS);
        } else if (mensaje.auxData == FICHA_FIJADA) {
          g_serie_encolar_cadena(CADENA_EMPATE);
        }
        g_serie_encolar_cadena(CADENA_MINUTOS_JUGADOS);
        g_serie_encolar_cadena(CADENA_SEGUNDOS_JUGADOS);
        g_serie_encolar_cadena(CADENA_CALIDAD_SERVICIO);
        g_serie_encolar_comenzar();
      }
      break;
  }
}
```

Listing 31: g_serie.c

```
/**
 * @file cadenas.h
 * @authors: Fernando Lahoz & Héctor Toral
```

```
 * @date: 22/09/2022
 * @description: Declara las cadenas de texto que se muestran en la pantalla
 */

#ifndef CADENAS_H
#define CADENAS_H

enum Cadenas {
  CADENA_FILA1 = 0xF1,
  CADENA_FILA2 = 0xF2,
  CADENA_FILA3 = 0xF3,
  CADENA_FILA4 = 0xF4,
  CADENA_FILA5 = 0xF5,
  CADENA_FILA6 = 0xF6,

  CADENA_CALIDAD_SERVICIO = 0xFF,
  CADENA_SEGUNDOS_JUGADOS = 0xFE,
  CADENA_MINUTOS_JUGADOS = 0xFD,

  CADENA_CABECERA1 = 0,
  CADENA_CABECERA2 = 1,
  CADENA_CABECERA3 = 2,
  CADENA_CABECERA4 = 3,
  CADENA_CABECERA5 = 4,
  CADENA_CABECERA6 = 5,
  CADENA_CABECERA7 = 6,
  CADENA_CABECERA8 = 7,
  CADENA_CABECERA9 = 8,
  CADENA_CABECERA10 = 9,
  CADENA_CABECERA11 = 10,
  CADENA_CABECERA12 = 11,
  CADENA_CABECERA13 = 12,
  CADENA_CABECERA14 = 13,
  CADENA_CABECERA15 = 14,
  CADENA_CABECERA16 = 15,
  CADENA_CABECERA17 = 16,
  CADENA_CABECERA18 = 17,
  CADENA_CABECERA19 = 18,
  CADENA_CABECERA20 = 19,

  // Alias 16 - 19
  CADENA_COMENZAR1 = 16,
  CADENA_COMENZAR2 = 17,
  CADENA_COMENZAR3 = 18,
  CADENA_COMENZAR4 = 19,

  CADENA_TURNO_BLANCAS = 20,
  CADENA_TURNO_NEGRAS = 21,

  CADENA_COLUMNA_NO_VALIDA = 22,

  CADENA_CANCELAR1 = 23,
  CADENA_CANCELAR2 = 24,
  CADENA_CANCELADO = 25,

  CADENA_BASE1 = 26,
  CADENA_BASE2 = 27,
```

```
  CADENA_GANAN_BLANCAS = 28,
  CADENA_GANAN_NEGRAS = 29,
  CADENA_EMPATE = 30,
  CADENA_RESET = 31,
  CADENA_FIN = 32,

  NUM_CADENAS = 33
};

static char *cadenas[NUM_CADENAS] = {
    "--------- Conecta 4 ---------\n\n",    //  0   = CADENA_CABECERA1
    " Jugad por turnos e intentad\n",       //  1   = CADENA_CABECERA2
    " alinear 4 fichas de vuestro\n",       //  2   = CADENA_CABECERA3
    "     color para ganar.\n\n",           //  3   = CADENA_CABECERA4
    "Usad el comando #C(1-7)! para\n",      //  4   = CADENA_CABECERA5
    " introducir una ficha en una\n",       //  5   = CADENA_CABECERA6
    "          columna.\n\n",               //  6   = CADENA_CABECERA7
    " El boton RESET reinicia la\n",        //  7   = CADENA_CABECERA8
    "  partida. El boton CANCEL\n",         //  8   = CADENA_CABECERA9
    " anula el ultimo movimiento\n",        //  9   = CADENA_CABECERA10
    "si se pulsa antes de que pase\n",      //  10  = CADENA_CABECERA11
    "un segundo. Los botones estan\n",      //  11  = CADENA_CABECERA12
    "  disponibles en la pestana\n",        //  12  = CADENA_CABECERA13
    "    \"View/Toolbox Window\".\n\n",      //  13  = CADENA_CABECERA14
    "  El comando #END! termina\n",         //  14  = CADENA_CABECERA15
    "         la partida.\n\n",             //  15  = CADENA_CABECERA16
    "-----------------------------\n",      //  16  = CADENA_CABECERA17
    "  Pulsa un boton o ejecuta\n",         //  17  = CADENA_CABECERA18
    " #NEW! para comenzar jugar.\n",        //  18  = CADENA_CABECERA19
    "-----------------------------\n\n",    //  19  = CADENA_CABECERA20
    "   || Turno de Blancas ||\n\n",        //  20  = CADENA_TURNO_BLANCAS
    "   || Turno de Negras  ||\n\n",        //  21  = CADENA_TURNO_NEGRAS
    "!! -- Columna no valida -- !!\n\n",    //  22  = CADENA_COLUMNA_NO_VALIDA
    "!! -- Pulsa CANCEL para -- !!\n",      //  23  = CADENA_CANCELAR1
    "           cancelar\n\n",              //  24  = CADENA_CANCELAR2
    "# -- Movimiento cancelado -- #\n\n",  //  25  = CADENA_CANCELAR3
    "      ----------------\n",            //  26  = CADENA_BASE1
    "       -|1|2|3|4|5|6|7|\n\n",         //  27  = CADENA_BASE2
    "===== Ganan las Blancas =====\n\n",    //  28  = CADENA_GANAN_BLANCAS
    "===== Ganan las Negras  =====\n\n",    //  29  = CADENA_GANAN_NEGRAS
    "========== Empate ==========\n\n",     //  30  = CADENA_EMPATE
    "---------- RESET ----------\n\n",      //  31  = CADENA_RESET
    "------ FIN DE PARTIDA -------\n\n"     //  32  = CADENA FIN
};

#endif
```

Listing 32: cadenas.h

Control de UART

```
/**
 * @file UART0.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definicion de funciones para el manejo de la UART0
 */
```

```c
#ifndef UART_ZERO_H
#define UART_ZERO_H

#include <LPC210X.H>
#include <inttypes.h>

#include "cola_asyn.h"
#include "utils.h"

enum { BUFFER_ENVIO_SIZE = 32 };

/**
 * @brief Configura el envío de caracteres a través del periférico UART0.
 */
void uart0_iniciar(void);

/**
 * @brief Inicia el envío de una cadena de caracteres tipo C, enviando el
 *        primer caracter.
 * @param array Cadena a enviar.
 */
void uart0_enviar_array(const char* array);

/**
 * @brief Envía el siguiente caracter de la cadena.
 * @return true si quedaban caracteres por enviar.
 */
int uart0_continuar_envio(void);

#endif
```

Listing 33: UART0.h

```c
/**
 * @file UART0.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementacion de funciones para el manejo de la UART0
 */

#include "UART0.h"

static volatile uint8_t indice_envio = 0, longitud;
static char buffer_envio[BUFFER_ENVIO_SIZE];

void uart0_IRC(void) __irq {
  static int vecesRecibido = 0;
  static int vecesEnviado = 0;

  int iir = U0IIR;

  if (iir & 0x04) {  // BRB
    cola_encolar_eventos(CARACTER_RECIBIDO, ++vecesRecibido, U0RBR);
  }
  if (iir & 0x2) {  // THRE
    cola_encolar_eventos(CARACTER_ENVIADO, ++vecesEnviado, 0);
  }
```

```
  VICVectAddr = 0;
}

void uart0_iniciar() {
  PINSEL0 = PINSEL0 | 0x5;  // Habilita RxD0 y TxD0 (UART 0)

  U0LCR = 0x80;  // DLAB = 1, para modificar la tasa de baudios

  // Para conseguir el estándar de 9600bd con PCLK =  15MHz
  U0DLM = 0;
  U0DLL = 97;
  // Cálculo:
  // Rs = PCLK / (16 * (256 * U0DLM + U0DLL)) = 15E6 / (16 * 97) = 9.664bd

  U0LCR = 0x03;  // 8 bits, sin paridad, 1 Stop bit, DLAB = 0

  // Habilita FIFO + Interrupciones al recibir 1 único caracter
  U0FCR = (U0FCR | 0x1) & ~0xC0;  // U0FCR =  '11xx xxx1'
  U0IER = U0IER | 0x3;            // Habilita interrupciones RBR y THRE

  VICVectAddr3 = (unsigned long)uart0_IRC;

  // Habilita la interrupción 6 (UART0)
  VICVectCntl3 = VICVectCntl3 | 0x26;
  VICIntEnable = VICIntEnable | 0x40;  // Enable UART0 Interrupt (bit 6)
}

void uart0_enviar_array(const char* array) {
  if (*array == '\0') return;  // Un array vacío no se envía

  int fin = FALSE;
  for (longitud = 0; longitud < BUFFER_ENVIO_SIZE && !fin; longitud++) {
    buffer_envio[longitud] = array[longitud];

    // No almacenar el caracter final permite enviar uno extra
    fin = (array[longitud + 1] == '\0');
  }
  indice_envio = 0;
  U0THR = buffer_envio[indice_envio];
  indice_envio++;
}

int uart0_continuar_envio(void) {
  if (indice_envio < longitud) {
    U0THR = buffer_envio[indice_envio];
    indice_envio++;
    return TRUE;
  }
  return FALSE;
}
```

Listing 34: UART0.c

Gestor de energía

```
/**
 * @file g_energia.h
 * @authors: Fernando Lahoz & Héctor Toral
```

```
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo de la energía
 */

#ifndef GESTOR_ENERGIA_H
#define GESTOR_ENERGIA_H

#include <inttypes.h>

#include "cola_asyn.h"
#include "cola_msg.h"
#include "eventos.h"
#include "g_alarmas.h"
#include "msg.h"
#include "power.h"
#include "utils.h"

enum estados { NORMAL, IDLE, POWERDOWN };

/**
 *  @brief Inicializa el gestor de energía.
 */
void g_energia_iniciar(void);

/**
 * @brief Pone al procesador a dormir.
 * En el estado power-down los periféricos también entran en bajo consumo y
 * dejan de funcionar pero se sigue manteniendo el estado.
 */
void g_energia_power_down(void);

/**
 * @brief Pone al procesador en modo normal.
 */
void g_energia_reset(void);

/**
 * @brief Pone el procesador en modo bajo consumo.
 * En g_energia_idle el procesador se para, pero los periféricos del chip,
   como
 * el temporizador, siguen activos y lo pueden despertar al realizar una
 * interrupción.
 */
void g_energia_idle(void);

/**
 * @brief Tratamiento de mensajes del módulo del gestor de energía
 * @param mensaje Mensaje a tratar
 */
void g_energia_tratar_mensaje(msg_t mensaje);

#endif
```

Listing 35: g_energia.h

```
/**
 * @file g_energia.c
 * @authors: Fernando Lahoz & Héctor Toral
```

```
 * @date: 22/09/2022
 * @description: Implementación de funciones para el manejo de la energía
 */

#include "g_energia.h"

static int estado = NORMAL;

void setup_PLL(void);

void g_energia_iniciar() {
  power_iniciar();
  cola_encolar_msg(SET_ALARM, g_alarma_crear(POWER_DOWN, FALSE, 10000));
}

void g_energia_power_down() {
  estado = POWERDOWN;
  power_down();
  setup_PLL();
  cola_encolar_msg(SET_ALARM, g_alarma_crear(POWER_DOWN, FALSE, 10000));
}

void g_energia_reset() {
  if (estado == IDLE) {
    // Cancelar LATIDO
    cola_encolar_msg(SET_ALARM, g_alarma_crear(LATIDO, TRUE, 0));
    cola_encolar_msg(APAGAR_LATIDO, 0);
    estado = NORMAL;
  }
  // Resetear alarma de POWER_DOWN
  cola_encolar_msg(SET_ALARM, g_alarma_crear(POWER_DOWN, FALSE, 10000));
}

void g_energia_idle() {
  if (estado == NORMAL) {
    // Activar LATIDO
    cola_encolar_msg(SET_ALARM, g_alarma_crear(LATIDO, TRUE, 250));
    estado = IDLE;
  }
  idle();
}

void g_energia_tratar_mensaje(msg_t mensaje) {
  switch (mensaje.ID_msg) {
    case POWER_DOWN:
      g_energia_power_down();
      break;
    case EJECUTAR:
      // No hace falta reiniciar la alarma
      // porque reinicia al salir de powerdown
      if (estado == POWERDOWN) {
        estado = NORMAL;
      } else {
        cola_encolar_msg(mensaje.auxData, 0);
      }
    case RESET_POWERDOWN:
      g_energia_reset();
      break;
```

```
    }
}
```

Listing 36: g_energia.c

Power

```
/**
 * @file power.h
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Definición de funciones para el manejo del bajo consumo
 */

#ifndef POWER_H
#define POWER_H

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

/**
 * @brief Permite que el procesador se despierte de PD.
 */
void power_iniciar(void);

/**
 * @brief LLama a función ASM establecida en Startup.s para restablecer
 * la configuración del PLL.
 */
void setup_PLL(void);

/**
 * @brief Pone al procesador a dormir.
 * En el estado power-down los periféricos también entran en bajo consumo y
 * dejan de funcionar pero se sigue manteniendo el estado.
 */
void power_down(void);

/**
 * @brief Pone el procesador en modo bajo consumo.
 * En g_energia_idle el procesador se para, pero los periféricos del chip,
    como
 * el temporizador, siguen activos y lo pueden despertar al realizar una
 * interrupción.
 */
void idle(void);

#endif
```

Listing 37: power.h

```
/**
 * @file power.c
 * @authors: Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Implementacion de funciones para el manejo del bajo consumo
 */

#include "power.h"
```

```
void power_iniciar() { EXTWAKE = 6; }

void power_down() { PCON = PCON | 0x2; }

void idle() { PCON = PCON | 0x1; }
```

Listing 38: power.c

Conecta 4

```
/**
 * @file celda.h
 * @authors: Profesores de la asignatura &
 *           Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Tipo Abstracto de Datos. Encapsula el formato de celda
 */

#ifndef CELDA_H
#define CELDA_H

#include <inttypes.h>

/**
 * @brief Cada celda se codifica en 8 bits. Sólo se usan los 3 menos
 *        significativos
 *        bit 0 -> 1: ficha blanca, 0 : no hay ficha blanca
 *        bit 1 -> 1: ficha negra,  0 : no hay ficha negra
 *        bit 2 -> 1: ocupada,      0 : posición vacía
 */
typedef uint8_t CELDA;

/**
 * @brief Modifica el valor almacenado en la celda indicada.
 *        Valores válidos:
 *          1: ficha blanca
 *          2: ficha negra
 */
enum { FICHA_BLANCA = 1, FICHA_NEGRA = 2, FICHA_FIJADA = 3 };

__inline static void celda_poner_valor(CELDA *celdaptr, uint8_t val) {
  if ((val == FICHA_BLANCA) || (val == FICHA_NEGRA) || (val == FICHA_FIJADA))
    {
     *celdaptr = 0x04 /*ocupado*/ + val;
  }
}

__inline static void celda_vaciar(CELDA *celdaptr) { *celdaptr = 0; }

/**
 * @brief Devuelve 1 si la celda está vacía
 */
__inline static uint8_t celda_vacia(CELDA celda) { return (celda & 0x4) == 0;
    }

/**
 * @brief Devuelve color celda y si vacia 0
```

```
 */
__inline static uint8_t celda_color(CELDA celda) { return (celda & 0x03); }

/**
 * @brief Devuelve 1 si la celda es blanca y valida
 */
__inline static uint8_t celda_blanca(CELDA celda) { return celda == 0x05; }

/**
 * @brief Devuelve 1 si la celda es negra y valida
 */
__inline static uint8_t celda_negra(CELDA celda) { return celda == 0x06; }

/**
 * @brief Devuelve 1 si la celda es fijada y valida
 */
__inline static uint8_t celda_fijada(CELDA celda) { return celda == 0x07; }
#endif  // CELDA_H
```

Listing 39: celda.h

```
/**
 * @file celda.h
 * @authors: Profesores de la asignatura &
 *           Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Tipo Abstracto de Datos. Encapsula el formato de celda
 */

#ifndef CONECTA4_H_2022
#define CONECTA4_H_2022

#include <inttypes.h>

#include "celda.h"
#include "cola_msg.h"
#include "g_alarmas.h"
#include "msg.h"
#include "utils.h"

/* Tamaños de la cuadricula */
enum {
  NUM_FILAS = 6,
  PADDING_FIL = 1,
  TAM_FILS = PADDING_FIL + NUM_FILAS,
  NUM_COLUMNAS = 7,
  PADDING_COL = 1,
  TAM_COLS = PADDING_COL + NUM_COLUMNAS
};

enum C4_estado { C4_ACTIVO, C4_ESPERANDO, C4_INACTIVO };

/**
 * @brief Devuelve la longitud de la línea más larga
 */
uint8_t conecta4_buscar_alineamiento_c(CELDA cuadricula[TAM_FILS][TAM_COLS],
                                       uint8_t fila, uint8_t columna,
                                       uint8_t color, int8_t delta_fila,
```

```
                                        int8_t delta_columna);

/**
 * @brief Devuelve la longitud de la línea más larga
 */
uint8_t conecta4_buscar_alineamiento_arm(CELDA cuadricula[TAM_FILS][TAM_COLS],
                                         uint8_t fila, uint8_t columna,
                                         uint8_t color, int8_t delta_fila,
                                         int8_t delta_columna);

/**
 * @brief Devuelve true si encuentra una línea de longitud mayor o igual a 4
 */
uint8_t conecta4_hay_linea_c_c(CELDA cuadricula[TAM_FILS][TAM_COLS],
                               uint8_t fila, uint8_t columna, uint8_t color);

/**
 * @brief Devuelve true si encuentra una línea de longitud mayor o igual a 4
 */
uint8_t conecta4_hay_linea_arm_c(CELDA cuadricula[TAM_FILS][TAM_COLS],
                                 uint8_t fila, uint8_t columna, uint8_t color)
    ;

/**
 * @brief Devuelve true si encuentra una línea de longitud mayor o igual a 4
 */
uint8_t conecta4_hay_linea_arm_arm(CELDA cuadricula[TAM_FILS][TAM_COLS],
                                   uint8_t fila, uint8_t columna,
                                   uint8_t color);

/**
 * @brief Tratamiento de mensajes del módulo de conecta 4.
 */
void conecta4_tratar_mensaje(msg_t mensaje);

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

uint8_t C4_calcular_fila(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t columna
    );

void C4_actualizar_tablero(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila,
                           uint8_t columna, uint8_t val);

void C4_vaciar_celda_tablero(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t
    fila,
                             uint8_t columna);

int C4_comprobar_empate(CELDA cuadricula[TAM_FILS][TAM_COLS]);

int C4_verificar_4_en_linea(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila
    ,
                            uint8_t columna, uint8_t color, int *fail);

static inline uint8_t C4_alternar_color(uint8_t colour) {
  return colour == 1 ? 2 : 1;  // jugador 1 o jugador 2
}

static inline uint8_t C4_columna_valida(uint8_t columna) {
```

```
  return (columna >= 1) && (columna <= NUM_COLUMNAS);
}

static inline uint8_t C4_fila_valida(uint8_t fila) {
  return (fila >= 1) && (fila <= NUM_FILAS);
}

/**
 * @brief Inicializa el módulo de juego de conecta4
 */
void conecta4_iniciar(CELDA tablero[TAM_FILS][TAM_COLS]);

/**
 * @brief Reacción del juego ante un evento JUGAR
 */
void C4_jugar(CELDA tablero[TAM_FILS][TAM_COLS], uint8_t *estado, uint8_t *
    fila,
              uint8_t *columna);

/**
 * @brief Reacción del juego ante un evento CONFIRMAR_JUGADA
 */
void C4_confirmar_jugada(CELDA tablero[TAM_FILS][TAM_COLS], uint8_t *estado,
                         uint8_t *fila, uint8_t *columna, uint8_t *color);

/**
 * @brief Reacción del juego ante un evento PEDIR_FILA
 */
void C4_devolver_fila(CELDA tablero[TAM_FILS][TAM_COLS], uint32_t fila);

#endif /* CONECTA4_H_2022 */
```

Listing 40: conecta4_2022.h

```
/**
 * @file celda.h
 * @authors: Profesores de la asignatura &
 *           Fernando Lahoz & Héctor Toral
 * @date: 22/09/2022
 * @description: Tipo Abstracto de Datos. Encapsula el formato de celda
 */

#include "conecta4_2022.h"

uint8_t C4_calcular_fila(CELDA cuadricula[TAM_FILS][TAM_COLS],
                         uint8_t columna) {
  uint8_t fila = 1;

  if ((columna < 1) || (columna > NUM_COLUMNAS)) {
    return ERROR;
  }

  while ((fila <= NUM_FILAS) &&
         (celda_vacia(cuadricula[fila][columna]) == FALSE)) {
    fila++;
  }
  return fila <= NUM_FILAS ? fila : ERROR;
};
```

```
__attribute__((noinline)) uint8_t conecta4_buscar_alineamiento_c(
    CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila, uint8_t columna,
    uint8_t color, int8_t delta_fila, int8_t delta_columna) {
  // avanzar hasta que celda esté vacía, sea distinto color o lleguemos al
   borde
  if (!C4_fila_valida(fila) || !C4_columna_valida(columna)) {
    return 0;
  }

  // posicion valida y mismo color
  if (celda_vacia(cuadricula[fila][columna]) ||
      (celda_color(cuadricula[fila][columna]) != color)) {
    return 0;
  }

  // avanzar índices
  uint8_t nueva_fila = fila + delta_fila;
  uint8_t nueva_columna = columna + delta_columna;

  // incrementar longitud y visitar celda vecina
  return 1 + conecta4_buscar_alineamiento_c(cuadricula, nueva_fila,
                                            nueva_columna, color, delta_fila,
                                            delta_columna);
}

__attribute__((noinline)) uint8_t conecta4_hay_linea_c_c(
    CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila, uint8_t columna,
    uint8_t color) {
  int8_t deltas_fila[4] = {0, -1, -1, 1};
  int8_t deltas_columna[4] = {-1, 0, -1, -1};
  unsigned int i = 0;
  uint8_t linea = FALSE;
  uint8_t long_linea = 0;

  // buscar linea en fila, columna y 2 diagonales
  for (i = 0; (i < 4) && (linea == FALSE); ++i) {
    // buscar sentido
    long_linea = conecta4_buscar_alineamiento_c(
        cuadricula, fila, columna, color, deltas_fila[i], deltas_columna[i]);
    linea = long_linea >= 4;
    if (linea) {
      continue;
    }
    // buscar sentido inverso
    long_linea += conecta4_buscar_alineamiento_c(
        cuadricula, fila - deltas_fila[i], columna - deltas_columna[i], color,
        -deltas_fila[i], -deltas_columna[i]);
    linea = long_linea >= 4;
  }
  return linea;
}

uint8_t conecta4_hay_linea_arm_arm_c(CELDA cuadricula[TAM_FILS][TAM_COLS],
                                     uint8_t fila, uint8_t columna,
                                     uint8_t color) {
  int8_t deltas_fila[4] = {0, -1, -1, 1};
  int8_t deltas_columna[4] = {-1, 0, -1, -1};
```

```
  unsigned int i = 0;
  uint8_t long_linea = 0;
  uint8_t fila_aux = fila;
  uint8_t columna_aux = columna;

  if (!C4_fila_valida(fila) || !C4_columna_valida(columna) ||
      celda_vacia(cuadricula[fila][columna]) ||
      (celda_color(cuadricula[fila][columna]) != color)) {
    return FALSE;
  }

  // buscar linea en fila, columna y 2 diagonales
  for (i = 0; i < 4; ++i) {
    long_linea = 1;
    fila += deltas_fila[i];
    columna += deltas_columna[i];
    // buscar sentido
    while (!(!C4_fila_valida(fila) || !C4_columna_valida(columna) ||
             celda_vacia(cuadricula[fila][columna]) ||
             (celda_color(cuadricula[fila][columna]) != color))) {
      fila += deltas_fila[i];
      columna += deltas_columna[i];
      long_linea++;

      if (long_linea == 4) return TRUE;
    }

    fila = fila_aux - deltas_fila[i];
    columna = columna_aux - deltas_columna[i];
    // buscar sentido inverso
    while (!(!C4_fila_valida(fila) || !C4_columna_valida(columna) ||
             celda_vacia(cuadricula[fila][columna]) ||
             (celda_color(cuadricula[fila][columna]) != color))) {
      fila -= deltas_fila[i];
      columna -= deltas_columna[i];
      long_linea++;
      if (long_linea == 4) {
        return TRUE;
      }
    }
    fila = fila_aux;
    columna = columna_aux;
  }
  return FALSE;
}

__attribute__((noinline)) uint8_t conecta4_hay_linea_c_arm(
    CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila, uint8_t columna,
    uint8_t color) {
  int8_t deltas_fila[4] = {0, -1, -1, 1};
  int8_t deltas_columna[4] = {-1, 0, -1, -1};
  unsigned int i = 0;
  uint8_t linea = FALSE;
  uint8_t long_linea = 0;

  for (i = 0; (i < 4) && (linea == FALSE); ++i) {
    long_linea = conecta4_buscar_alineamiento_arm(
        cuadricula, fila, columna, color, deltas_fila[i], deltas_columna[i]);
```

```
    linea = long_linea >= 4;
    if (linea) {
      continue;
    }
    long_linea += conecta4_buscar_alineamiento_arm(
        cuadricula, fila - deltas_fila[i], columna - deltas_columna[i], color,
        -deltas_fila[i], -deltas_columna[i]);
    linea = long_linea >= 4;
  }
  return linea;
}

void C4_actualizar_tablero(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila,
                           uint8_t columna, uint8_t val) {
  celda_poner_valor(&cuadricula[(fila)][(columna)], val);
}

void C4_vaciar_celda_tablero(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t
    fila,
                             uint8_t columna) {
  celda_vaciar(&cuadricula[(fila)][(columna)]);
}

// comprueba si esta jugada llena todo el tablero y hay empate
int C4_comprobar_empate(CELDA cuadricula[TAM_FILS][TAM_COLS]) {
  for (int i = 1; i < TAM_COLS; i++) {
    if (celda_vacia(cuadricula[TAM_FILS - 1][i])) {
      return FALSE;
    }
  }
  return TRUE;
}

int C4_verificar_4_en_linea(CELDA cuadricula[TAM_FILS][TAM_COLS], uint8_t fila
    ,
                            uint8_t columna, uint8_t color, int *fail) {
  uint8_t resultado_c_c =
      conecta4_hay_linea_c_c(cuadricula, fila, columna, color);
  uint8_t resultado_c_arm =
      conecta4_hay_linea_c_arm(cuadricula, fila, columna, color);
  uint8_t resultado_arm_c =
      conecta4_hay_linea_arm_c(cuadricula, fila, columna, color);
  uint8_t resultado_arm_arm =
      conecta4_hay_linea_arm_arm(cuadricula, fila, columna, color);
  if (fail != 0)
    *fail = resultado_c_c != resultado_c_arm ||
            resultado_c_c != resultado_arm_c ||
            resultado_c_c != resultado_arm_arm;
  return resultado_c_c && resultado_c_arm && resultado_arm_c &&
         resultado_arm_arm;
}

void conecta4_iniciar(CELDA tablero[TAM_FILS][TAM_COLS]) {
  for (int i = 1; i <= NUM_FILAS; i++) {
    for (int j = 1; j <= NUM_COLUMNAS; j++) {
      tablero[i][j] = 0;
    }
  }
```

```
}

void C4_jugar(CELDA tablero[TAM_FILS][TAM_COLS], uint8_t *estado, uint8_t *
    fila,
              uint8_t *columna) {
  *fila = 0;
  int ok = C4_columna_valida(*columna);
  if (ok) *fila = C4_calcular_fila(tablero, *columna);
  ok = ok && C4_fila_valida(*fila);

  cola_encolar_msg(ENTRADA_VALIDADA, ok);

  if (ok) {
    *estado = C4_ESPERANDO;
    C4_actualizar_tablero(tablero, *fila, *columna, FICHA_FIJADA);
    cola_encolar_msg(CELDA_MARCADA, 0);
    cola_encolar_msg(SET_ALARM, g_alarma_crear(CONFIRMAR_JUGADA, FALSE, 1000))
    ;
  }
}

void C4_confirmar_jugada(CELDA tablero[TAM_FILS][TAM_COLS], uint8_t *estado,
                         uint8_t *fila, uint8_t *columna, uint8_t *color) {
  C4_actualizar_tablero(tablero, *fila, *columna, *color);
  cola_encolar_msg(JUGADA_REALIZADA, 0);
  int ganador = conecta4_hay_linea_arm_arm(tablero, *fila, *columna, *color);
  int empate = C4_comprobar_empate(tablero);
  if (ganador) {
    cola_encolar_msg(FIN, *color);
    *estado = C4_INACTIVO;
  } else if (empate) {
    cola_encolar_msg(FIN, FICHA_FIJADA);  // empate
    *estado = C4_INACTIVO;
  } else {
    *color = C4_alternar_color(*color);
    *estado = C4_ACTIVO;
  }
}

void C4_devolver_fila(CELDA tablero[TAM_FILS][TAM_COLS], uint32_t fila) {
  uint32_t result = fila;
  for (int col = 1; col <= NUM_COLUMNAS; col++) {
    int desplazamiento = ((col - 1) << 2) + 4;  //(col - 1) * 4 + 4
    result = result | (tablero[fila][col] << desplazamiento);
  }
  cola_encolar_msg(DEVOLVER_FILA, result);
}

void conecta4_tratar_mensaje(msg_t mensaje) {
  static uint8_t estado = C4_INACTIVO, fila, columna, color;
  static CELDA tablero[7][8] = {
      0,   0XC1, 0XC2, 0XC3, 0XC4, 0XC5, 0XC6, 0XC7,
      0XF1,   0,    0,    0,    0,    0,    0,    0,
      0XF2,   0,    0,    0,    0,    0,    0,    0,
      0XF3,   0,    0,    0,    0,    0,    0,    0,
      0XF4,   0,    0,    0,    0,    0,    0,    0,
      0XF5,   0,    0,    0,    0,    0,    0,    0,
      0XF6,   0,    0,    0,    0,    0,    0,    0
```

```
  };

  switch (mensaje.ID_msg) {
    case RESET:
      if (estado != C4_ESPERANDO) {
        estado = C4_ACTIVO;
        color = FICHA_BLANCA;
        conecta4_iniciar(tablero);
      }
      break;
    case FIN:
      if (estado == C4_ACTIVO) {
        estado = C4_INACTIVO;
      }
      break;
    case CANCELAR:
      if (estado == C4_INACTIVO) {
        color = FICHA_BLANCA;
        conecta4_iniciar(tablero);
      } else if (estado == C4_ESPERANDO) {
        C4_vaciar_celda_tablero(tablero, fila, columna);
        cola_encolar_msg(SET_ALARM, g_alarma_crear(CONFIRMAR_JUGADA, FALSE, 0)
   );
      }
      estado = C4_ACTIVO;
      break;
    case JUGAR:
      if (estado == C4_ACTIVO) {
        columna = mensaje.auxData;
        C4_jugar(tablero, &estado, &fila, &columna);
      }
      break;
    case CONFIRMAR_JUGADA:
      C4_confirmar_jugada(tablero, &estado, &fila, &columna, &color);
      break;
    case PEDIR_FILA:
      C4_devolver_fila(tablero, mensaje.auxData);
      break;
    case PEDIR_JUGADOR:
      cola_encolar_msg(JUGADOR, color);
      break;
  }
}
```

Listing 41: conecta4_2022.c