PRÁCTICA 2: SISTEMA DE E/S Y DISPOSITIVOS BÁSICOS

En esta práctica vamos a aprender a trabajar con periféricos sencillos que se encuentran en todos los sistemas actuales: `timers` y `GPIO` (General-Purpose I/O). Los timers, o temporizadores, permiten generar retardos de forma precisa, realizar tareas periódicas, o medir tiempos. La GPIO nos permitirá emular botones, leds, o un teclado. Para gestionar estos elementos trabajaremos con un controlador de interrupciones (`VIC`). También configuraremos los pines de entrada del sistema para que reciba interrupciones externas que simularán botones. Además, vamos a aprender a dormir y a despertar al procesador. De esta forma cuando el procesador no está realizando cálculos (por ejemplo, porque está esperando a que el usuario realice su movimiento) podemos dormirlo, sin perder el estado, y reducir su consumo de energía.

La interacción con los periféricos se hará en C. Para ello diseñaremos un **código modular**, con interfaces claros que abstraerán los detalles de bajo nivel y permitirán a nuestro programa principal interaccionar con ellos de forma sencilla. Además, aprenderemos a depurar diversas fuentes concurrentes de interrupción y desarrollaremos un **planificador** que gestionará la respuesta a los distintos eventos que ocurran en el sistema, y que sea capaz de reducir el **consumo de energía** de nuestro procesador.

OBJETIVOS

- Gestionar la entrada/salida con dispositivos básicos, asignando valores a los registros internos de la placa desde un programa en C (utilizando las bibliotecas de la placa)
- Desarrollar en C las rutinas de tratamiento de interrupción
- Aprender a utilizar periféricos, como los temporizadores internos de la placa y General Purpose Input/Output (GPIO)
- Depurar eventos concurrentes asíncronos
- Entender los modos de ejecución del procesador y el tratamiento de excepciones
- Ser capaces de depurar un código con varias fuentes de interrupción activas
- Desarrollar un código modular con interfaces claros y robustos para que cualquier aplicación pueda utilizar los periféricos de forma sencilla
- Diseñar un planificador que monitoriza los eventos del sistema, gestiona la respuesta a estos y reduce el consumo de energía del procesador.

ENTORNO DE TRABAJO

Continuaremos con el mismo entorno de trabajo, ARM Keil 5, que en la práctica anterior y emplearemos funcionalidades que hasta ahora no manejábamos, sobre todo para trabajar con interrupciones.

Es imprescindible **leer el enunciado completo** de esta práctica antes de la primera sesión para hacer mejor uso de ella.

MATERIAL DISPONIBLE

Este guion sólo incluye algunas indicaciones, sin explicar en detalle el funcionamiento de cada periférico. Antes de utilizar cada uno de los periféricos debéis estudiar la documentación de la placa, forma parte de los objetivos de

la asignatura que aprendáis a localizar la información en las hojas técnicas. Allí encontraréis la descripción detallada del interfaz de cada periférico que vamos a utilizar. No hace falta que entendáis las cosas que no utilizamos, pero sí que debéis saber qué hace cada registro de entrada/salida que utilicéis.

En Moodle de la asignatura podéis encontrar el siguiente material de apoyo:

- Un proyecto que utiliza pulsadores virtuales y los timers de la placa. Antes de escribir una línea de código debéis entender a la perfección este proyecto.
- Documentación original de la placa (proporcionada por la empresa desarrolladora): muy útil para ver más detalles sobre la entrada / salida, los periféricos del System-on-Chip empleado (SoC) y su mapa de memoria.

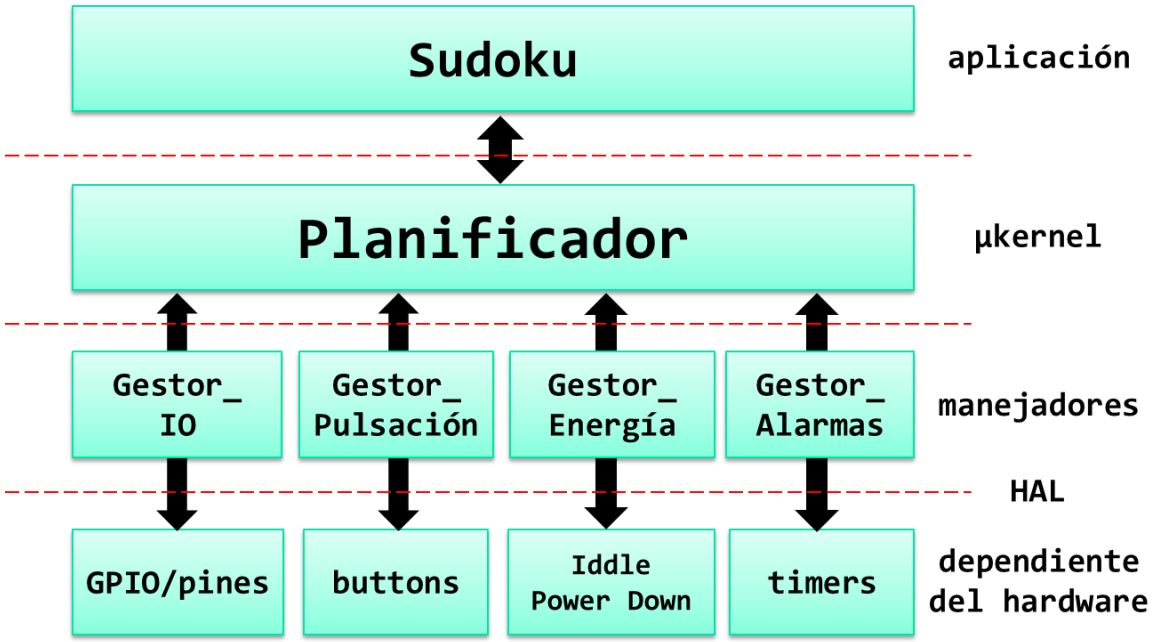
Además, disponéis de vuestros proyectos generados en la práctica 1.

Nota: Keil te permite alterar el valor de los registros de entrada/salida escribiendo en ellos directamente a través de interfaces gráficos (tal y cómo hemos hecho en la P1 modificando la memoria para indicar la fila y la columna). Esto puede usarse para hacer pruebas y depurar. Pero eso sólo se puede hacer en el simulador, no en un procesador de verdad, por tanto, no sirve para desarrollar código. El código que entreguéis debe asignar los valores adecuados a los registros de entrada/salida. Es decir, debe funcionar sin necesidad de que asignemos ningún valor en los interfaces gráficos del simulador. La única excepción serán las entradas externas que simularán botones o teclados y se introducirán modificando los pines de entrada del sistema.

ARQUITECTURA DEL SISTEMA

El objetivo es tener un código modular, en el que no se mezcle la lógica del juego y la lógica que controla los periféricos, ni la de un periférico con la de otro a no ser que sea imprescindible. De esta forma tendremos un nivel dependiente del hardware que ofrece una serie de servicios a los niveles superiores. También tendremos manejadores o gestores de dispositivos que se encargarán de la lógica de los periféricos independizando su gestión de la aplicación concreta a ejecutar. En medio tendremos un planificador que se encargará del tratamiento de todos los eventos asíncronos.

En la siguiente figura os presentamos un esquema de los módulos principales a desarrollar:



El código que desarrolléis debe cumplir las siguientes reglas:

- Cada módulo tendrá su código separando las definiciones, fichero.h, de su implementación, fichero.c. La configuración de ese módulo sólo se realizará con las funciones que incluyáis en la biblioteca (API). Por ejemplo, si un módulo gestiona una variable interna, ésta no será visible de forma directa desde el exterior. La variable se encapsulara y se incluirán funciones visibles en el código para leerla o modificarla desde fuera si procede. Además, dichas variables no dependerán ni incluirán información de la aplicación (juego), sino información dependiente de ese módulo.
- La E/S es retardada, las interrupciones deben ser tan ligeras cómo se pueda. No se hacen otros cálculos, sólo se procesa el periférico y se informa que ha recibido una interrupción de un determinado tipo.

El código debe ser legible. En lugar de usar constantes directamente se recomienda definir etiquetas auto descriptivas. De la misma forma no debéis usar estados tipo *S0*, *S1* o *S2*, sino por ejemplo; e_inicial, e_*pulsado*, e_*no_pulsado*, e_*esperando_pulsacion*, e_esperando_fin_pulsacion….

ESTRUCTURA DE LA PRÁCTICA

Paso 0: Estudiar la documentación

Paso 1: Diseño de una cola de eventos

En un sistema con distintos eventos asíncronos, a la hora de depurar nos interesa no solo saber cómo hemos llegado a un punto, sino la secuencia previa de eventos; ya que el orden entre ellos o la producción de unos u otros puede condicionar la correcta ejecución del programa.

Para depurar la ejecución y la secuencia de eventos producida vais a crear una cola de eventos en la que se registren todos los eventos asíncronos del sistema.

La comunicación y sincronización entre los distintos módulos del diseño se realizará a través de una cola circular que almacenará la información de los últimos 32 eventos generados en el sistema. Un evento puede ser cualquier cosa que nos interese. Por ejemplo, una interrupción. Los eventos se guardarán con la función *cola_guardar_eventos(uint8_t ID_evento, uint32_t auxData)*. Esta función guardará en la cola dos palabras, 64 bits en total. La primera será el campo ID_evento (8 bits), que permita identificar el evento (p.e. qué interrupción ha saltado), y el campo *auxData* (que se puede usar para datos auxiliares sobre el evento ocurrido, por ejemplo, el botón pulsado). Como se piensa encolar eventos desde distintos módulos es conveniente definir adecuadamente los ID_evento de los eventos e incluirlos en cada módulo que los use con el fichero *eventos.h*. Deberéis definir un identificador único para cada posible tipo de evento que se pueda producir. Será necesario definir un nuevo módulo cola con ficheros *cola.h* y *cola.c*.

Como la iteración con el usuario se realiza por entrada/salida (E/S), el programa principal estará pendiente de ella para realizar cálculos. Podemos hacer uso de la cola de eventos para convertir el programa en orientado a eventos añadiendo un **planificador**. Básicamente el programa estará pendiente de la cola de eventos. Cuando aparezca un evento nuevo no tratado, se procesará. De esa forma la comunicación entre la E/S y el programa principal siempre se realizará mediante la cola de eventos.

Deberéis diseñar una función que compruebe si la cola tiene nuevos eventos, y otra que lea el evento más antiguo sin procesar. Además, al ser una cola circular, podría pasar que se borrasen eventos que no han sido aún procesados (*overflow*). Si eso ocurriese se debe visualizar en el led de **`overflow`** y parar la ejecución (bucle infinito), más adelante se verá el módulo de visualización.

Paso 2: Utilización de los temporizadores

Nuestro SoC (System on Chip) LPC2105 incluye dos temporizadores de propósito general (`timers` 0 y 1, capítulo 14), junto con otros de propósito específico (`WatchDog` para resetear al procesador en caso de que malfuncionamiento, y `RTC` (Real Time Clock), para tener la hora y fecha).

Debéis aprender a utilizar estos temporizadores y realizar una pequeña biblioteca que los controle implementando las siguientes funciones o servicios al resto del software:

- temporizador_iniciar(): función que programa un contador para que pueda ser utilizado.
- temporizador_empezar(): función que inicia la cuenta de un contador de forma indefinida.
- temporizador_leer(): función que lee el tiempo que lleva contando el contador desde la última vez que se ejecutó temporizador_empezar y lo devuelve en microsegundos.

- temporizador_parar(): detiene el contador y devuelve el tiempo transcurrido desde temporizador_empezar.

Programa el ***timer1*** para que mida tiempos con la máxima precisión posible, pero interrumpiendo (y por tanto sobrecargando al programa principal) lo menos posible.

A la hora de planificar tareas a realizar es interesante poder programar un temporizador periódico para por ejemplo poder tener alarmas. Utilizando el ***timer0*** añadirás la siguiente función a la biblioteca del temporizador.

- temporizador_periodico(periodo): función que programa el temporizador para que encole un evento periódicamente. El periodo se indica en ms.

Como vais a utilizar interrupciones a parte de estudiar los timers deberéis estudiar el `VIC`.

- Para comprobar que vuestra librería funciona correctamente podéis utilizar la herramienta de `Keil Logic Analizer`. En moodle tenéis un video con un ejemplo de funcionamiento.

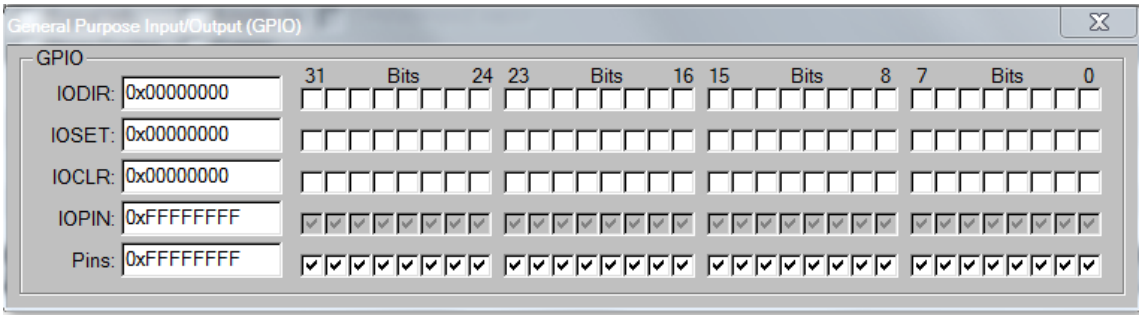
Necesitaremos planificar diversas tareas. Por ello necesitaremos un módulo que las maneje. El **`Gestor_Alarmas`** nos permitirá poder programar diversas alarmas que generen distintos eventos.

Cuando otro módulo quiera programar una nueva alarma para un evento concreto generará un evento de **`Set_Alarma`** que el planificador hará llegar al gestor. En el campo auxiliar se indicará en los 8 bits de mayor peso el evento a generar, el siguiente bit indica si es alarma periódica o no, y en los 23 restantes los milisegundos de retardo. Por ejemplo, el valor 1000 indicará un segundo. Si el retardo es cero se cancelará la alarma. Si se reprograma un determinado evento antes de que salte se reprogramará al nuevo valor.

El gestor tendrá 8 posibles alarmas pendientes asociadas a 8 posibles eventos distintos.

Cada vez que le llegue el evento del temporizador periódico se deberá comprobar si hay que disparar el evento asociado a alguna de las alarmas programadas pendientes. Si lo hay se dispararán y se cancelará la alarma si no es periódica.

Paso 3: Programación de GPIO



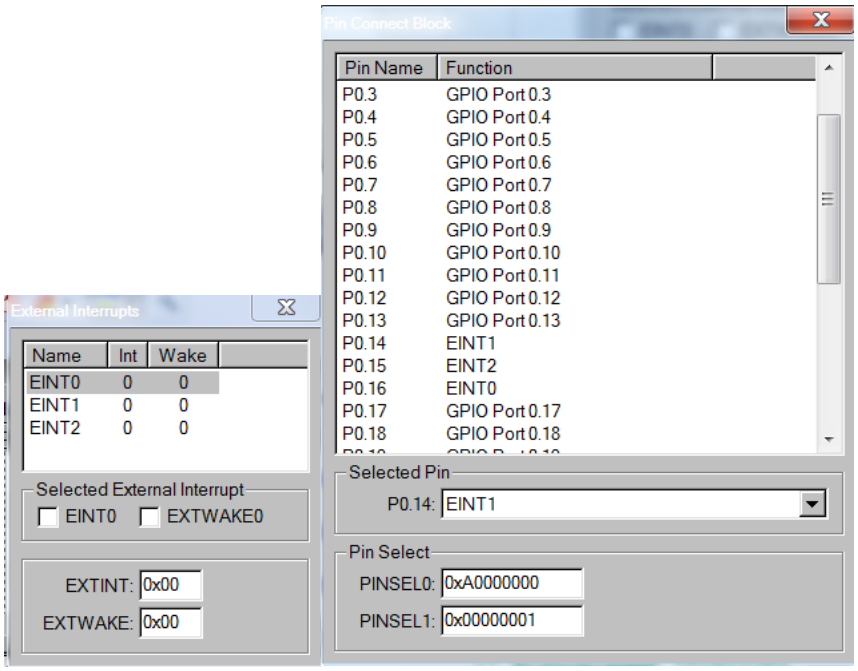
Ventana de Keil para visualizar la GPIO, e introducir entradas en los pines del procesador. Se puede encontrar en *Peripherals>>GPIO*.

El `GPIO` es un periférico que se puede utilizar para conectar al chip elementos de entrada salida como leds o entradas digitales. Hay disponible un puerto de 32 bits que se pueden utilizar como entrada o salida.
Debéis realizar una pequeña biblioteca que interactúe con la GPIO con las siguientes funciones

- GPIO_iniciar(): Permite emplear el GPIO y debe ser invocada antes de poder llamar al resto de funciones de la biblioteca.
- GPIO_leer(bit_inicial, num_bits): bit_inicial indica el primer bit a leer. num_bits indica cuántos bits queremos leer. La función devuelve un entero con el valor de los bits indicados. Ejemplo:
  - valor de los pines: `0x0F0FAFF0`
  - bit_inicial: 12 num_bits: 4
  - valor que retorna la función: `10` (lee los 4 bits 12-15)
- GPIO_escribir(bit_inicial, num_bits, valor): similar al anterior, pero en lugar de leer escribe en los bits indicados el valor (si valor no puede representarse en los bits indicados se escribirá los num_bits menos significativos a partir del inicial)
- GPIO_marcar_entrada(bit_inicial, num_bits): los bits indicados se utilizarán como pines de entrada.
- GPIO_marcar_salida(bit_inicial, num_bits): los bits indicados se utilizarán como pines de salida.

Paso 4: Interrupciones externas

Nuestro SoC dispone de tres líneas de interrupción externas que permiten interaccionar al procesador con dispositivos de entrada salida que están fuera del chip. Vamos a utilizar dos de ellas `EINT1` y `EINT2` para simular botones.



Ventanas de Keil para visualizar las interrupciones externas (*Peripherals>>System Control Block>> External Interrupts*) y las conexión de los pines *(Peripherals>> Pin Connect Block)*. En este ejemplo se han configurado los pines 14, 15 y 16 para las interrupciones externas.

Para usarlas hay que conectarlas a los pines del sistema. Como hay muchas más posibles conexiones que pines, estos incluyen multiplexores que se configuran con los registros `PINSEL0` y `PINSEL1` (ver manual, capítulos 6 y 7).

Según el manual las interrupciones se deberían poder configurar para trabajar por flanco o nivel, y activas a alta o a baja (capítulo 3.6 del manual) utilizando los registros `EXTPOLAR` y `EXTMODE`. Pero ni en el simulador ni en las librerías aparecen esos registros. Por tanto, vamos a tener que trabajar con los valores por defecto: Las interrupciones se activan por nivel y son activas a baja: es decir que si hay un cero en el pin `P0.14` se activará la solicitud de interrupción de `EINT1`.

Como las interrupciones son muy rápidas y un botón puede estar bastante tiempo presionado, si no hacemos algo una misma pulsación podría generar multitud de interrupciones. Para evitarlo debéis implementar un esquema que garantice que sólo hay una interrupción por pulsación:

- Cuando se detecta la interrupción se procesa la pulsación. Procesar la pulsación consistirá en asignar a una variable (flag) el valor 1. Como usaremos dos botones tendremos dos flags: *nueva_pulsacion_eint2* y *nueva_pulsacion_eint1*. Diseñaremos dos funciones para leer estas variables (*button_nueva_pulsacion_2()* y *button_nueva_pulsacion_1()*) que devolverán `1` si ha habido una nueva pulsación, y otras dos para resetearlas a cero (*button_clear_nueva_pulsacion_2()* y *button_clear_nueva_pulsacion_1()*).
- Durante la gestión de la interrupción se deshabilitará la interrupción externa correspondiente en el VIC, para que no vuelva a interrumpir hasta que no termine la gestión de la pulsación.

Como se ve en la figura inicial, la gestión de la pulsación de un botón estará en el módulo **`Gestor_Pulsacion`** aparte. En él se gestionará una sencilla máquina de estados para cada botón. Basta con dos estados (pulsado, no pulsado)

- Con cada nueva pulsación se creará el evento correspondiente para ser tratado por el planificador.
- Se utilizará una alarma para monitorizar la pulsación cada 100ms. Si la tecla sigue pulsada no se hará nada. En caso contrario se volverá a habilitar la interrupción de ese botón. Importante: hay que limpiar su solicitud antes de volver a habilitarla tanto en el VIC, como en el registro EXTINT, o conforme se habilite se producirá una interrupción no deseada.

Paso 5: Eliminación de bucles de espera innecesarios

Normalmente el programa principal estará pendiente de si tiene algo que hacer en un bucle *while* en lo que se conoce como espera activa. Si la mayor parte del tiempo el planificador no tiene nada útil que hacer, el procesador sencillamente estará desperdiciando energía ejecutando esta espera activa. Este consumo reduce la duración de la batería, o aumenta la factura de la luz, y no aporta nada. Como sabéis reducir el consumo de energía, y desarrollar estrategias de consumo responsable, son dos puntos clave de los Objetivos de -Desarrollo Sostenible de la ONU. Es una buena política para nuestro futuro alinearnos con estos objetivos, y aprender a gestionar mejor el consumo de energía de nuestro procesador. Reducir el consumo de nuestro micro puede parecer poco importante, pero lo cierto es que en 2020 el sector TIC fue responsable del 15% del total de las emisiones de CO2 [informe eficiencia energética y TIC]. Y ese porcentaje está aumentando año tras año. Por lo que es realmente importante para nuestro futuro que nuestros sistemas informáticos sean eficientes y no realicen consumos innecesarios.

La estrategia que vamos a seguir es muy sencilla, y se puede realizar de una forma o de otra en la mayoría de los procesadores actuales. Cuando el procesador no tenga cálculos que hacer, en lugar de ejecutar un bucle innecesario, lo vamos a dormir para que reduzca su consumo. Para ello utilizaremos el registro `PCON` (Power Control) descrito en el capítulo 3.10 del manual.

Haréis dos funciones. Una función que ponga al procesador en modo `idle` y otra que lo duerma (`power-down`). En `idle` el procesador se para, pero los periféricos del chip, como el temporizador, siguen activos y lo pueden despertar al realizar una interrupción. En el estado *power-down* los periféricos también entran en bajo consumo y dejan de funcionar pero se sigue manteniendo el estado. El procesador despertará si recibe una interrupción externa si las configuráis con el registro `EXTWAKE` (ver la configuración de las interrupciones externas en el manual).

Se desarrollará un manejador de estados de energía (**`Gestor_Energia`**) que se encarga de gestionar los estados del procesador y decidir si se pasa a powerdown.

La comunicación de la actividad (eventos asíncronos) se realizará mediante la cola de eventos previamente creada y serán tratados por el planificador. El planificador es el elemento clave en la ejecución del sistema. Su labor es ejecutar un bucle infinito que monitoriza la cola de eventos e invoca a la función o funciones que deben tratar cada uno de ellos. Si no hay ningún evento en la cola, el planificador solicitará al Gestor_Energia el paso del procesador a modo idle para ahorrar energía.

Paso 6: integración de la I/O en el juego

Vamos a realizar toda la entrada utilizando la GPIO y las EINT2 y 1. La salida seguirá siendo mostrando el tablero en memoria y en GPIO.

Se ha pensado utilizar los GPIO tanto para la interface con el usuario, tanto para introducir fila, columna y valor como para visualizar candidatos y errores. El reparto elegido es:

- GPIO0-3: valor de la celda seleccionada
- GPIO4-12: candidatos de la celda seleccionada
- GPIO13: led validar entrada o mostrar error en la celda seleccionada
- GPIO14: EINT1 (escribir dato en celda seleccionada)
- GPIO15: EINT2 (borrar dato celda seleccionada)
- GPIO16-19: Fila seleccionada (de 1 a 9)
- GPIO20-23: Columna seleccionada (de 1 a 9)
- GPIO24-27: nuevo valor a introducir (de 1 a 9)
- GPIO30: overflow
- GPIO31: latido modo idle

Queremos incluir las siguientes funcionalidades:

- Habrá que cambiar la lógica del juego para adaptarla al esquema propuesto haciendo el tratamiento de eventos (iniciar, cambio en entrada y jugada).
- Crear un **`Gestor_IO`** que se encargue de traducir y aislar de los GPIO concretos utilizados tanto para la entrada como para la visualización. Recibirá eventos para visualizar información (p.e. candidatos, valor, overflow, error, etc) y generará eventos si hay cambios en la entrada. Se programará una alarma periódica para que 5 veces por segundo se actualice la visualización de la celda seleccionada.
- Se medirá el tiempo de procesamiento de actualizar las candidatas en microsegundos. Es decir, desde que el jugador ha introducido su movimiento, hasta que candidatos y errores han sido recalculados.
- El botón `EINT1` escribirá el nuevo valor en la celda. Acto seguido se recalcularán candidatos y posibles errores. Si la jugada es válida se mostrará encendiendo el led de validar durante un segundo para mostrar la validación. Si la celda fuese una pista no se podrá modificar el valor y se encenderá mientras este seleccionada la celda el led de validar (error).
- El botón `EINT2` indicará que el usuario quiere borrar el valor de esa celda. El led de validar actuará como en el caso anterior.

- Si se introduce la jugada fila 0, columna 0, valor 0, la partida terminará. El juego se quedará sin hacer nada hasta que se pulse un botón (cualquiera de los dos). Al detectar la pulsación se comenzará una partida nueva.
- Sobre la gestión de `idle` y `power-down`:
  - Pondremos al procesador en modo idle siempre que el sistema no tenga ningún evento que planificar. En idle los temporizadores están activos, así que periódicamente despertarán al procesador para las actualizaciones. EXTINT2 y EXTINT1 también están activas así que se procesará cualquier pulsación de confirmación o cancelación.
  - Apagaremos el procesador, modo power-down, siempre que pasen 15 segundos sin que haya ningún cambio en las entradas. La implementación consistirá en activar una alarma al comenzar la ejecución con periodo 15 segundos. Cada vez que se detecte un cambio, ya sea porque se pulse un botón, o porque cambien las entradas de la GPIO, se reiniciará la alarma en cuestión. Si alguna vez llega a activarse la alarma entraremos en el modo de ahorro de energía. Para salir de este modo el usuario debe pulsar uno de los dos botones. Esa pulsación no debe escribir ni borrar nada en el juego. Para que un botón pueda despertar al procesador debéis configurar `EXTINT2` y `EXTINT1` para que despierten al procesador al pulsar usando el registro `EXTWAKE2` y `EXTWAKE1`. De esta forma cuando pulséis se despertará al procesador.

EVALUACIÓN DE LA PRÁCTICA

La primera parte de la práctica (paso 5) se deberá mostrar al profesor de vuestro grupo al inicio de vuestra tercera sesión de esta práctica (es decir, en vuestra sesión 7 del calendario de la EINA).

La práctica completa deberá entregarse en la sesión 9. Las fechas definitivas y turnos de corrección se publicarán en Moodle de la asignatura.