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`). 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
- 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 entrada). 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

En esta práctica vamos a ir construyendo funcionalidades de soporte, pero independientes del juego o aplicación concreta a ejecutar (un juego, un airbag o un horno microondas).

Nuestro sistema funcionara con un **modelo de eventos**. Tendremos un **planificador** que procesará los **eventos** que se produzcan y planificará las **tareas** a realizar para cada evento. Los eventos pueden ser generados por la interacción del usuario (E/S) o por tareas planificadas por el planificador.

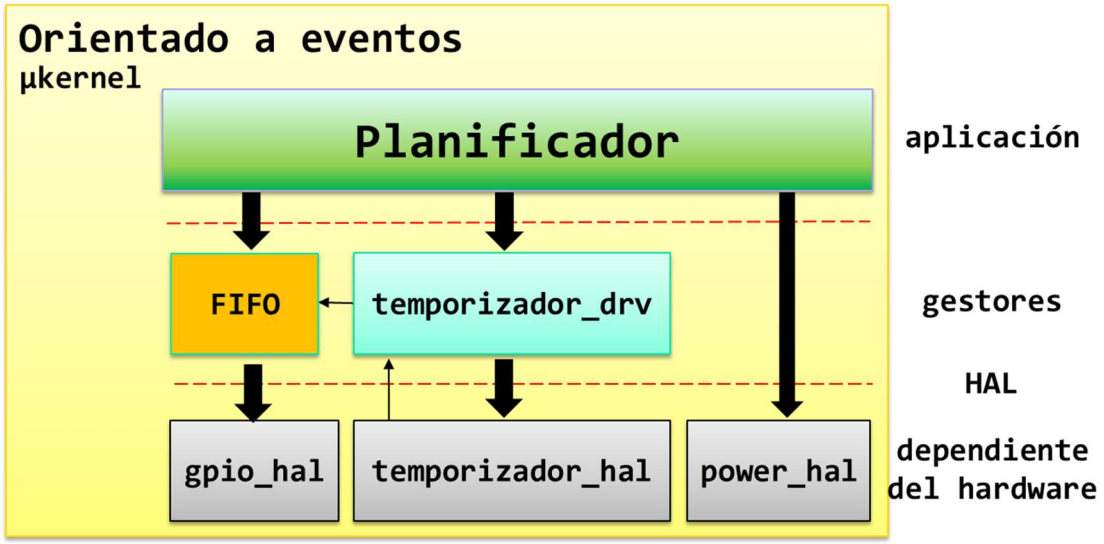
El código será **modular** y **portable**. Las funciones se encapsularán y se independizarán del hardware. Esto permitirá reutilizar el código en diferentes plataformas con solo recompilarlo.

Para la entrada / salida vamos a **abstraernos del nivel hardware.** Tendremos un módulo dependiente del hardware que nos aislara del mismo (HAL). La parte independiente del mismo lo encapsularemos en manejadores o gestores de dispositivos (independientes del hardware concreto) que se encargarán de la lógica de los periféricos (modelo de dispositivo) e independizando su gestión de la aplicación concreta a ejecutar.

El objetivo es tener un código modular, en el que la lógica del juego y la lógica de los periféricos estén claramente separadas. Esto permitirá **reutilizar** el código en diferentes juegos o aplicaciones y en diferentes plataformas hardware.

Los módulos superiores podrán utilizar **servicios** de los módulos inferiores. La comunicación de eventos asíncronos entre los módulos inferiores y los superiores se realizará a través de un mecanismo de **respuesta asíncrona** y gestionados por el planificador.

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 encapsulará y se incluirán funciones visibles en el código para leerla o modificarla desde fuera si procede (TAD). 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.
- Cada módulo debe incluir pruebas automáticas para garantizar su correcto funcionamiento.
- El código debe ser legible: Usar nombres para las variables y las funciones claros y auto explicativos, en lugar de usar constantes directamente se recomienda definir etiquetas, comentar el código, etc...

ESTRUCTURA DE LA PRÁCTICA

Paso 0: Estudiar la documentación y el ejemplo suministrado

Paso 1: Crear un temporizador

El primer módulo que vamos a crear es un temporizador. El temporizador nos permitirá medir la distancia temporal entre dos acciones. Nos interesa medir el tiempo en microsegundos (para tener precisión). Para la realización de la librería nos ayudaremos de un `timer` hardware.

Nuestro SoC (System on Chip) LPC2105 incluye dos temporizadores de propósito general (`timers` 0 y 1, capítulo 14 del manual del microprocesador), 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. Para ello vamos a separar las funciones dependientes del hardware (HAL – *Hardware Abstraction Layer*) y las del gestor de dispositivo (manejador o *DRiVer*).

Programa el ***`timer0`*** para que mida tiempos con la máxima precisión (*ticks*), pero interrumpiendo (y por tanto sobrecargando al programa principal) lo menos posible. Determina y justifica el rango y la precisión del contador.

El módulo dependiente del hardware tiene las siguientes funciones o servicios que nos independizan del hardware concreto a usar:

- `la constante temporizador_hal_ticks2us:` permite convertir de ***ticks*** a microsegundos.
- `temporizador_hal_iniciar()`: función que programa un contador para que pueda ser utilizado.
- `temporizador_hal_empezar()`: función que inicia la cuenta de un contador de forma indefinida.
- `uint64_t temporizador_hal_leer()`: función que lee el tiempo que lleva contando el contador desde la última vez que se ejecutó `temporizador_hal_empezar` y lo devuelve en ***ticks***.
- `uint64_t temporizador_hal_parar()`: detiene el contador y devuelve el tiempo en *ticks* transcurrido desde el último `temporizador_hal_empezar`.

El manejador o driver es un módulo independiente del hardware y nos presenta un dispositivo temporizador donde podemos contar el tiempo entre dos intervalos en microsegundos. Hace uso de los servicios del módulo de abstracción del hardware. Dispone de las siguientes funciones:

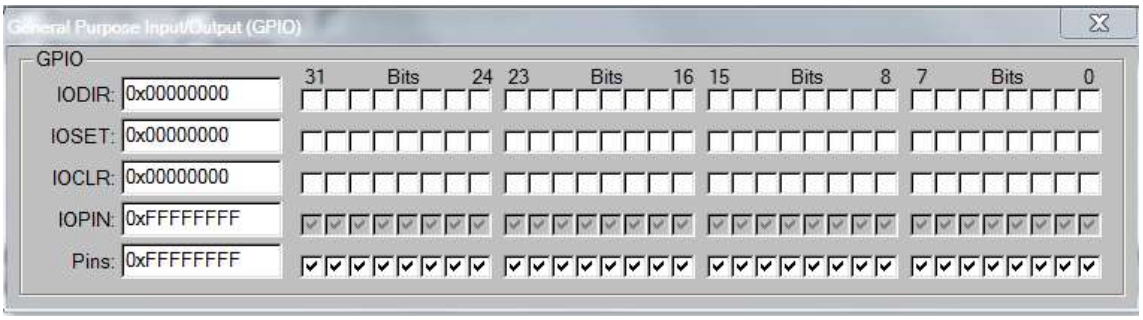
- `temporizador_drv_iniciar()`: función que programa un contador para que pueda ser utilizado.
- `temporizador_drv_empezar()`: función que inicia la cuenta de un contador de forma indefinida.
- `uint64_t temporizador_drv_leer()`: función que lee el tiempo que lleva contando el contador desde la última vez que se ejecutó `temporizador_drv_empezar` y lo devuelve en ***microsegundos***.
- `uint64_t temporizador_drv_parar()`: detiene el contador y devuelve el tiempo transcurrido desde el último `temporizador_drv_empezar`.

Notar que si un día cambiamos de plataforma o de timer solo deberíamos tocar el módulo ***hal***. El manejador y el resto del código quedarían invariantes.

Paso 2: Entrada / Salida básica

Nuestro juego deberá comunicarse con el mundo exterior. Por ello iremos añadimiento mecanismos de entrada/salida. Para los siguientes puntos necesitaremos implementar un mínimo de entrada/salida.

El `GPIO` es un periférico que se puede utilizar para conectar al chip elementos de entrada salida como leds o entradas digitales. En el LPC2105 hay disponible un puerto de 32 bits que se pueden utilizar como entrada o salida (capítulo 8).



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

Debéis realizar una pequeña biblioteca que nos abstraiga del hardware e interactúe con los GPIO implementando:

- el conjunto de direcciones E/S del GPIO `gpio_hal_pin_dir_t`: `GPIO_HAL_PIN_DIR_INPUT`, `GPIO_HAL_PIN_DIR_OUTPUT`.
- el tipo de datos de la representación del pin del GPIO: `GPIO_HAL_PIN_T`
- `gpio_hal_iniciar()`: Permite emplear el GPIO y debe ser invocada antes de poder llamar al resto de funciones de la biblioteca.
- `gpio _hal_sentido(GPIO_HAL_PIN_T gpio_inicial, uint8_t num_bits, gpio_hal_pin_dir_t direccion)`: los bits indicados se utilizarán como entrada o salida según la dirección.
- `uint32_t gpio _hal_leer(GPIO_HAL_PIN_T gpio_inicial, uint8_t num_bits)`: gpio_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_hal_escribir(GPIO_HAL_PIN_T bit_inicial, uint8_t num_bits, uint32_t 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)

Todas estas funciones deberían ser declaradas aconsejando al compilador que incruste el código (*`inlining`*) al igual que se hacía con `celda.h` en la P1.

Vamos a crear un fichero de cabecera que centralice la reserva de GPIOs. En este fichero de cabecera (`io_reserva.h`) se asignarán los GPIO a un nombre humano y una longitud que posteriormente utilizarán los otros módulos.

Paso 3: Diseño de una cola de eventos

Nuestro programa funcionará bajo el paradigma de orientado a eventos. Los eventos pueden ser generados asíncronamente por la interacción del usuario (E/S) o por tareas planificadas previamente. En un momento determinado podemos tener más de un evento a la espera de ser tratado. Para gestionarlo vamos a crear una cola de eventos FIFO. El **planificador** básicamente estará a la espera pendiente de la cola de eventos. Cuando aparezca un evento nuevo no tratado, lo procesará.

Diseñar el módulo software de la cola FIFO de eventos. El módulo tendrá las siguientes funciones:

- Definición del tipo de datos ***EVENTO_T***: Conjunto de eventos posibles. En el fichero de cabecera se incluirá el tipo de datos y el conjunto de posibles eventos identificados con nombre humano. Reservemos el ID VOID con valor cero para inicializar la cola.
- `FIFO_inicializar(GPIO_HAL_PIN_T pin_overflow)`: Inicialización de la cola. Se le pasa como parámetro el pin del GPIO utilizado para marcar errores.
- `FIFO_encolar(EVENTO_T ID_evento, uint32_t auxData)`: Esta función guardará en la cola el evento. El campo ID_evento, que permita identificar el evento (p.e. qué interrupción ha saltado) y el campo *auxData* en caso de que el evento necesite pasar información extra.
- `uint8_t FIFO_extraer(EVENTO_T *ID_evento, uint32_t* auxData)`: Si hay eventos sin procesar, devuelve un valor distinto de cero y el evento más antiguo sin procesar por referencia. Cero indicará que la cola está vacía y no se ha devuelto ningún evento.
- `uint32_t FIFO_estadisticas(EVENTO_T ID_evento)`: Dado un identificador de evento nos devuelve el número total de veces que ese evento se ha encolado. El evento VOID nos devolverá el total de eventos encolados desde el inicio.

Al ser colas circulares de tamaño estático, podría pasar que lleguen nuevos eventos antes de haber procesado los anteriores y se borrasen eventos que no han sido aún procesados al desbordar la cola (*overflow*). Es un error en diseño por tener que limitar tamaño. Como no podemos insertar un nuevo evento la ejecución a partir de ese momento seria errónea si no lo insertásemos. Por ello la ejecución no puede continuar y debemos parar la ejecución en FIFO_encolar y mostrar de alguna forma el error. Si eso ocurriese se debe visualizar en el led de **`overflow`** y parar la ejecución (bucle infinito).

El planificador inicializará la cola indicando que el GPIO para marcar el `overflow` es el `GPIO31` definido convenientemente en la cabecera `io_reserva.h` como `GPIO_OVERFLOW = 31` y de longitud `GPIO_OVERFLOW_BITS = 1`.

En un sistema concurrente 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. Podemos hacer uso de la propia cola de eventos para depurar la ejecución.

La cola circular almacenará la información de los últimos 32 eventos generados en el sistema. Al depurar podemos mirar la cola y ver el histórico de eventos que nos llevó a ese punto. La función `FIFO_extraer` no borrará la información del evento, solo marcará que ha sido tratado.

Nuestro programa empieza a tomar cuerpo y es hora de organizar el código. Al empezar la ejecución `Startup.s` llegaremos a la función `main`. Vuestra función `main` debería inicializar el sistema (p.e. controlador de interrupciones) y los módulos de los manejadores necesarios. Tras ello debería llamar a la función `planificador()` del módulo `planificador.c` que se encargará de iniciar la cola y quedar en un bucle infinito pendiente de tratar eventos. Conforme añadamos funcionalidad el planificador ira añadiendo para cada tipo de evento quien o quienes están pendientes de él para cuando se procese ese evento llamar a las funciones de gestión de cada uno.

Paso 4: Activación periódica

A la hora de planificar tareas a realizar es interesante poder programar un temporizador periódico para por ejemplo poder tener alarmas. Utilizando el *timer1* añadir la siguiente función a los módulos de la biblioteca del temporizador.

- `temporizador_hal_reloj (uint32_t periodo, void (*funcion_callback)())`: función dependiente del hardware (timer1) que programa el reloj para que llame a la función de callback cada periodo. El periodo se indica en ms. Si el periodo es cero se para el temporizador.
- `temporizador_drv_reloj (uint32_t periodo, void (*funcion_encolar_evento)(), EVENTO_T ID_evento)`: función que programa el reloj para que encole un evento periódicamente en la cola del planificador. El periodo se indica en ms.

Cuando programemos el reloj, al manejador le indicaremos la función a la que llamar para encolar el evento generado (`FIFO_encolar`) y el identificador del evento a generar (`ID_evento`). El manejador inicializará la parte dependiente del hardware pasándole la función del manejador a avisar cada vez que venza el periodo.

Podemos aprovechar la activación periódica para testear el funcionamiento y desbordamiento de la cola de eventos.

Paso 5: Contador - Hello world

Los programadores suelen crear pequeños programas de prueba. Es habitual que el primer programa tenga un mensaje del estilo de "Hello World". En sistemas empotrados este test suele ser un diodo led encendiéndose y apagándose a una determinada frecuencia (a.k.a. Blink).

En vez de un Blink o latido vamos a desarrollar un breve contador de décimas de segundo para saber que el programa está corriendo. El contador se mostrará en los primeros ocho GPIO como si fueran leds. En `io_reserva.h definiremos` como `GPIO_HELLO_WORD = 0` y de longitud `GPIO_HELLO_WORD_BITS = 8`.

`hello_word` será el modulo software encargado de mantener este contador. Al principio de la ejecución el planificador iniciara el módulo `hello_word_inicializar` pasándole la definición de los pins de GPIO. El planificador le avisara con un `hello_word_tick_tack()` cada `tick` de 10ms para qué actualice el contador y el estado de los leds cuando toque.

Programar adecuadamente el planificador y la activación periódica decidiendo el periodo apropiado.

Paso 6: Logic Analyzer

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.

Paso 7: Reducción de consumo

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 (modo *idle*).

Para ello utilizaremos el registro `PCON` (Power Control) descrito en el capítulo 3.10 del manual. Crear un módulo de abstracción denominado power con la función `power_hal_wait()` que duerma el procesador (en prácticas posteriores añadiremos la función power_hal_deep_sleep() que desconectará partes del procesador). La función retornará cuando se produzca algún evento (interrupción), continuando la ejecución. Notar que si los temporizadores estaban en marcha antes de ir a idle los `timers` seguirán contando y produciendo interrupciones.

El planificador deberá llamar a esta función cuando la cola de eventos este vacía y no tenga eventos que tratar.

EVALUACIÓN DE LA PRÁCTICA

La primera parte de la práctica (paso 3) se deberá mostrar al profesor de vuestro grupo al inicio de vuestra segunda sesión de esta práctica (es decir, en vuestra semana 7 del calendario de la EINA.

La práctica completa deberá entregarse en la semana 8. Las fechas definitivas y turnos de corrección se publicarán en Moodle de la asignatura.