PRÁCTICA 2B: CONTINUACIÓN SISTEMA DE E/S Y DISPOSITIVOS BÁSICOS, MÁQUINA DE ESTADOS FINITA

En esta práctica vamos a seguir trabajando con periféricos sencillos que se encuentran en todos los sistemas actuales: `timers` y `GPIO` (General-Purpose I/O). Utilizando el timer 1, vamos a construir un módulo que gestione alarmas periódicas en el sistema. Y vamos a conectar dos de los puertos de la GPIO para que reciban interrupciones externas que simularán botones. Además, vamos a aprender a reducir aún más el consumo del procesador con la opción de power-down.

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.

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 que en la práctica anterior.

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 vuestras funciones generadas en la práctica 2.

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

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). Seguimos con un **modelo de eventos** y el código debe ser **modular** y **portable**.

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 siguiendo el ejemplo de las prácticas anteriores, en lugar de usar constantes se recomienda definir etiquetas, comentar el código, etc. 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: Comunicación entre módulos - Gestor Alarmas

En cualquier sistema es muy probable que necesitaremos planificar diversas tareas. Vamos a crear un módulo gestor **`alarmas`** que nos permitirá programar alarmas.

El módulo gestionará múltiples alarmas con temporizaciones distintas, partiendo del timer 1, que configuraremos para que interrumpa una vez cada 10ms. El número de alarmas distintas que en un momento dado pueden estar activas viene determinado por la constante en compilación `ALARMAS_MAX` del módulo `alarmas` (para las pruebas jugaremos con un máximo de 4 alarmas vivas).

Los servicios de este gestor deben ser utilizados por otros módulos gestores y por el planificador. A su vez el gestor de alarmas generará eventos añadiéndolos a la cola para que sean gestionados por el planificador.

Cuando un módulo quiera programar una nueva alarma invocará a la función **`alarma_activar(EVENTO_T ID_evento, uint32_t retardo, uint32_t auxData)`**. El `ID_evento` determina el evento a insertar en la cola al vencer la alarma (del conjunto de eventos posibles en la cola). El `retardo` codifica en el bit de más peso si la alarma es periódica (debe repetirse), los restantes bits indican el retardo en milisegundos. El campo `auxData` es el campo auxiliar que se encolara en la cola de eventos además del `ID_evento`. Si el `retardo` es cero se cancelará la alarma. Si se reprograma un determinado evento antes de que salte, se reprogramará al nuevo valor (identificador único).

Cada `tick` de 1ms, el planificador recibirá el evento generado por la activación periódica (timer 1) e invocará la función **`alarma_tratar_evento`** para qué revise las alarmas. Esta función comprobará si hay que disparar el evento asociado a alguna de las alarmas programadas pendientes. Las alarmas no periódicas se cancelan tras dispararse. Las periódicas continúan activas de forma indefinida hasta que se eliminen.

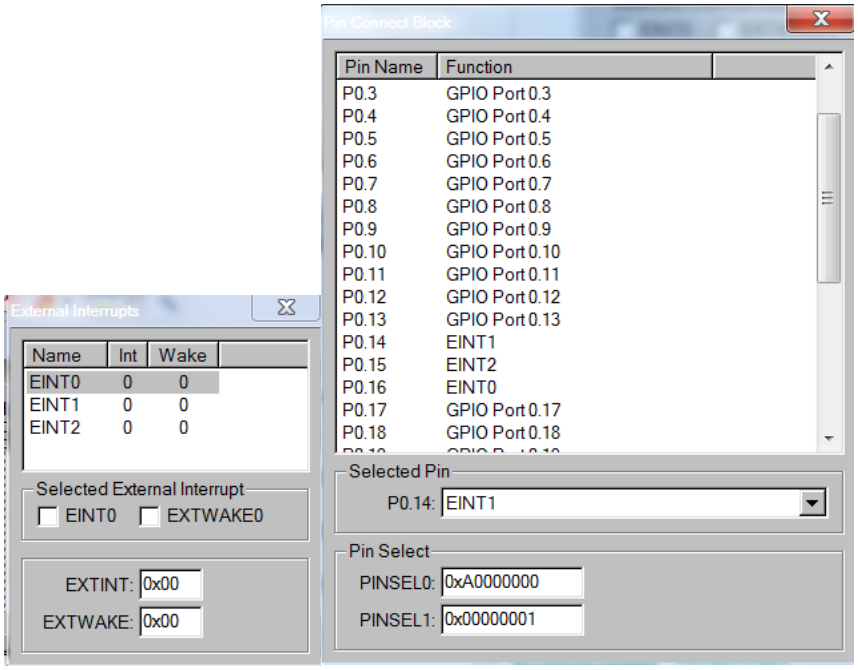
El planificador inicializará el módulo ejecutando la función `alarma_inicializar`.

Al igual que la cola de eventos, las alarmas se crean y tratan en tiempo de ejecución siendo una estructura estática. Cuando se programa la alarma de un evento nuevo, se asignará a la primera alarma libre. Si no hay alarmas disponibles se generará un evento de ALARMA_OVERFLOW que será gestionado por el planificador. Al gestionarlo, el planificador lo mostrará en el GPIO de *overflow* y parará la ejecución.

Paso 2: 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 por ejemplo la acción de dos botones.

Siguiendo el ejemplo de los otros módulos ya creados, debéis crear un módulo que gestione la parte dependiente del hardware (`int_externas_hal`), y otro independiente (`botones`) definiendo vosotros la interface siguiendo el patrón aprendido.



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.
- 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.
- Se llamará a una función independiente del hardware para avisar que se ha producido una pulsación enviándole como parámetro el ID del botón pulsado.

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

- Con cada nueva pulsación se encolará el evento de pulsación de botón indicando en el campo auxiliar el identificador del botón pulsado.
- Se utilizará una alarma para monitorizar la pulsación cada 100ms. Cuando salte la alarma el planificador avisará al módulo. Si la tecla sigue pulsada no se hará nada. En caso contrario se volverá a habilitar la interrupción de ese botón. Importante: en este hardware 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.

Programas las interrupciones `EINT1` y `EINT2` y comprueba el correcto funcionamiento utilizando la ventana de GPIO del Keil.

Paso 5: Reducción de consumo, power-down

Como ya vimos, parar al procesador nos permite reducir el consumo mientras esperamos a tener algo que hacer. Si las esperas son muy altas puede ser interesante desconectar más partes del sistema.

En la práctica anterior creamos el módulo *power* con la función `power_hal_wait()` que dormía al procesador a la espera de que llegase algún evento (`idle`). En este paso vamos a añadir al módulo la función power_hal_deep_sleep() que reduce aún más el consumo al pasar al estado `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).

Quien mejor conoce la actividad del sistema es el planificador. Crear una SFM en el planificador que gestione los modos de reducción de consumo. Si no hay ningún evento en la cola, el planificador solicitará el paso del procesador a modo `idle` para ahorrar energía. Si el planificador no observa actividad de usuario durante un determinado tiempo pedirá pasar a modo `power-down`.

Para esta última parte usar una alarma con un tiempo definido por la constante del planificador `USUARIO_AUSENTE`. Cada vez que observemos actividad de usuario reiniciaremos la alarma. Si vence la alarma pasaremos a `power-down`.

Comprobar el correcto funcionamiento, por ejemplo, con una temporización de 12 segundos. Observar los modos del procesador.

NOTA: Cuidado con el PLL al volver de `power-down` y con tener las interrupciones externas deshabilitadas al dormir. La FSM es propiedad del planificador ya que al volver de `power-down` la primera pulsación nos sirve para despertar.

Paso 7: Demostrador

Cada módulo debería implementar funciones de test y validación automáticas.

Además, para comprobar la integración de todos los módulos entre ellos vamos a realizar un pequeño demostrador. Esto nos permitirá ilustrar la iteración entre el planificador y las aplicaciones.

Crear un nuevo módulo denominado **`juego`**. Este módulo gestionará dos variables estáticas. La variable `cuenta` se inicializa con cero y se incrementará con cada pulsación del `EINT1` y se decrementa con y `EINT2`. Se debe incrementar o decrementar solo una vez con cada pulsación. La segunda variable, **`intervalo`**, guarda el tiempo transcurrido entre las dos últimas pulsaciones.

El planificador invocará al inicio de la ejecución a **`juego_inicializar`** para que juego realice sus tareas necesarias para su inicialización. En este caso asignar valor a sus dos variables. Más adelante, cuando el planificador reciba un evento de pulsación de botón invocara a la función **`juego_tratar_evento(EVENTO_T ID_evento, uint32_t auxData)`**. El planificador habrá puesto en marcha los temporizadores y los botones. A partir de ahora el temporizador pasa a ser de sistema y todos los módulos pueden utilizar `temporizador_drv_leer` para leer el tiempo. El módulo juego podrá determinar el intervalo entre pulsaciones llamando a `temporizador_drv_leer` y calculando la diferencia.

Para visualizar los valores de `cuenta`, vamos a crear un nuevo módulo denominado visualizar. Con cada nueva pulsación, `juego` creará el evento `ev_VISUALIZAR_CUENTA` colocando la cuenta en el campo auxiliar. El módulo visualizar, que habrá sido inicializado por el planificador, mostrará en los gpio del 23 al 16 los ocho bits menos significativos utilizando las funciones de `gpio_hal`. La definición del módulo seguirá el mismo esquema y patrón utilizado en los otros módulos.

El módulo `hello_world` lo vamos a adaptar a este nuevo entorno. Al inicializarse programará una alarma para que cada 10ms se genere el evento `ev_LATIDO`. El planificador al gestionar este evento llamara a la nueva función `hello_world_tratar_evento`. La visualización se realizará mediente un nuevo evento `ev_VISUALIZAR_HELLO`.

El planificador gestionará los modos de bajo consumo. Con esto el demostrador utiliza todos los módulos desarrollados hasta ahora. El demostrador deberá funcionar con las opciones de compilación –O3 –Otime.

EVALUACIÓN DE LA PRÁCTICA

La práctica deberá entregarse en semana 10 de la EINA. Las fechas definitivas y turnos de corrección se publicarán en Moodle de la asignatura.