PRÁCTICA 2c: DISEÑO DE UNA PLATAFORMA INDEPENDIENTE

INTRODUCCIÓN

En esta práctica completaréis el trabajo realizado en las dos anteriores, terminando de diseñar los módulos que vamos a necesitar para nuestro diseño final: un juego completo autónomo y funcionando correctamente en el procesador LPC2105.

OBJETIVOS:

El objetivo principal de esta práctica es finalizar el desarrollo de los módulos que utilizaremos en la parte final del proyecto. Particularmente, nos fijaremos los siguientes objetivos de aprendizaje:

- Diseñar e implementar una gestión completa y robusta de periféricos.
- Componer una arquitectura de sistema que aproveche los modos del procesador.
- Implementar y usar llamadas al sistema para interactuar con un dispositivo, un temporizador, y para activar y desactivar las interrupciones.
- Concebir e implementar un protocolo de comunicación robusto para el puerto serie.
- Identificar condiciones de carrera, su causa y solucionarlas.

REQUISITOS FUNCIONALES Y NO FUNCIONALES

RF1: El demostrador deberá iniciar y terminar de manera ordenada.

RF2: No habrá en el juego ninguna espera activa, salvo las condiciones de overflow.

RF3: Se gestionarán como llamadas al sistema la activación y desactivación de las rutinas de servicio de periféricos y la lectura del tiempo transcurrido.

RF4: El sistema será capaz de recibir comandos, y enviar arrays de caracteres a través de la línea serie.

RF5: En caso de mal funcionamiento, el sistema se reseteará de forma autónoma.

RNF1: Se puede depurar con nivel de optimización O0, pero la versión final debe funcionar correctamente en modo *release* con **O3 y -Otime**.

PASOS A SEGUIR

La práctica consta de los siguientes apartados obligatorios:

A. EJECUCCIÓN EN MODO USUARIO

Tal y como habéis aprendido en la asignatura de sistemas operativos, el núcleo de un sistema operativo (SO) se ejecuta en un modo de ejecución diferente al de las aplicaciones. Al primer modo de ejecución se le suele denominar supervisor y al segundo usuario. Una de las ventajas de estos modos es que permiten restringir ciertas

operaciones para que sean realizadas únicamente por el núcleo del SO y hacer los sistemas más seguros.

En nuestro diseño previo (prácticas 1y 2) el procesador trabaja siempre en modo supervisor. En la versión final queremos que antes de ejecutar el programa principal, llamada a `main`, la ejecución pase a **modo usuario**, pasando a modo supervisor para realizar algunas tareas del sistema.

En el fichero de arranque, `Startup.s`, **debemos comprobar que cambia a modo usuario** antes de invocar a la función `__main`.

IMPORTANTE: revisad en `Startup.s` el tamaño de las pilas de los distintos modos de trabajo. Si vais a usar un modo que antes no se usaba, asignar un espacio adecuado a su pila.

B. IMPLEMENTACIÓN DE LLAMADAS AL SISTEMA

En la arquitectura ARM, cuando las aplicaciones requieren de servicios del SO, emplean la instrucción `SWI` #número_servicio para requerirlos. Por ejemplo, leer un fichero puede corresponderse con el número 4 tal y como sucede en algunos sistemas Linux[1]. Al ejecutarse, SWI causa una interrupción software. Es decir, el modo de ejecución cambia a supervisor, el registro CPSR se almacena en el registro SPSR del modo supervisor y se ejecuta el código de la entrada correspondiente en el vector de SWI.

En la práctica anterior el temporizador se ponía en marcha al principio de la ejecución y cualquier módulo podía consultarlo con `temporizador_drv_leer`. Vamos a convertirlo en un reloj del sistema. Los módulos que quieran leer el temporizador realizarán una llamada al sistema.

En concreto, se implementarán las siguientes llamadas al sistema:

- **clock_get_us**: para leer el tiempo transcurrido en microsegundos utilizando el temporizador. Envuelve a la llamada `temporizador_drv_leer` que ya no podrá ser invocada directamente desde espacio de usuario.
- **read_IRQ_bit**: lee el bit IRQ del registro de estado para saber si las interrupciones están habilitadas o deshabilitadas (ver instrucción MRS. En A4.74 del manual de ARM).
- **enable_irq:** para activar las interrupciones irq en el registro de estado.
- **disable_irq**: para desactivar interrupciones irq en el registro de estado.
- **disable_fiq**: para desactivar interrupciones fiq en el registro de estado.

Para implementar estas llamadas al sistema realizaremos los siguientes pasos:

- Para acceder al temporizador vía `swi`, definiremos su número de swi, por ejemplo, el 0, y añadiremos el siguiente código en el módulo del timer:

[1] https://chromium.googlesource.com/native_client/nacl-newlib/+/master/libgloss/arm/linux-syscall.h

```
uint32_t __swi(0) clock_getus(void);
uint32_t __SWI_0 (void) {
   …
}
```

Debéis analizar en el binario generado el código de esta función para entender cómo se produce el cambio de contexto.

- Para `read_IRQ_bit` se hará de forma similar (asignando otros números de swi). Es importante revisar el código en ensamblador generado para esta función para entender cómo se produce el cambio de contexto.
- A partir de este momento, las marcas temporales en microsegundos se leerán llamando a la función `clock_get_us()`. La función `temporizador_drv_leer` no podrá ser invocada desde espacio de usuario.
- Estudiaremos el fichero `SWI.s` suministrado para ver cómo se accede al vector de interrupciones. (Ver vídeo y ejemplo en moodle).
- Buscaremos en `Startup.s` el vector de excepciones y cambiaremos el código asociado a las `SWI` para que invoque a nuestro manejador de interrupciones software.

Para activar/desactivar excepciones IRQ y FIQ, asumiremos que no nos caben en el vector y saltaremos a ellas desde la propia rutina de servicio de SWI como en se hace en el ejemplo de moodle con la función *decrease_var()*. Además, su bloque de código retornará a modo usuario. Activar/Desactivar las IRQs o IRQs y FIQs corresponderán a las llamadas al sistema 0xFF, 0xFE, 0xFD y 0xFC, respectivamente.

```
void __swi(0xFF) enable_irq (void);

void __swi(0xFE) disable_irq(void);

void __swi(0xFD) disable_fiq(void);
```

Nota: no incluimos enable_fiq, porque en este proyecto no vamos a usar las interrupciones FIQ. Por tanto, bastará con deshabilitarlas al comenzar la ejecución.

- Dentro del código para activar/desactivar las interrupciones IRQ y FIQ, que deberá estar dentro del fichero `SWI.s`, hay que cargar el registro **SPSR**, modificar los bits de IRQ o FIQ, actualizar SPSR, y finalmente volver a modo usuario. Para cargar el registro SPSR, deberéis de revisar la documentación de la instrucción `SWI` y la copia en memoria del registro CPSR al cambiar de modo.

C. UTILIZACIÓN DEL WATCHDOG

Nuestro procesador dispone de dos contadores con funcionalidades específicas: el `Real-time clock` (RTC)y el `watchdog` (WD). El RTC se utiliza para tener datos temporales en segundos, minutos, horas, días, meses y años. Pero en este proyecto no lo vamos a usar. Sí que utilizaremos el `watchdog` para salir de los bucles infinitos que pudieran generarse por algún tipo de malfuncionamiento.

- El `watchdog` es un contador que permite resetear el procesador. Es muy útil para resetear un programa que se haya quedado colgado. Se puede utilizar también

como un contador convencional, pero lo que nos interesa es aprender a utilizarlo para que genere un `reset`. Funciona de la siguiente forma. En primer lugar, se le asigna un tiempo de cuenta con el registro `WDTC` y se activa el bit de reset y el bit de enable en el registro `WDMOD`. A continuación, hay que "alimentarlo" realizando dos escrituras **consecutivas** en `WDFEED`: `WDFEED = 0xAA`; `WDFEED = 0x55` (cuidado, si las escrituras no son consecutivas, por ejemplo, si hay una interrupción entre ambas, se genera una excepción). Cuando el WD se alimenta por primera vez comienza su cuenta. Si la termina, y el bit de enable y de reset del WDMOD están activos, el WD resetea la ejecución. Si no queremos que el sistema se resetee se debe volver a alimentar antes de que termine. En tal caso, el WD volverá a empezar la cuenta desde el principio. Diseñad las siguientes funciones:

- `WD_hal_inicializar(int sec)`: inicializa el watchdog timer para que resetee el procesador dentro de *sec* segundos si no se le "alimenta".
- `WD_hal_feed()`: alimenta al watchdog timer
- `WD_hal_test()`: test con un bucle infinito y comprobar que el `watchdog` realmente funciona.

Tenéis que implementar la funcionalidad y comprobar el correcto funcionamiento del módulo .

D. ELIMINACIÓN DE CONDICIONES DE CARRERA

Una **condición de carrera** es un problema que ocurre cuando se accede de manera concurrente a un recurso compartido y el resultado final de las operaciones depende de la temporización. Es decir, el resultado de un programa puede depender de cómo ocurran las operaciones desde, por ejemplo, una rutina de servicio y el hilo del programa principal.

En nuestro caso concreto la cola de eventos pude sufrir condiciones de carrera si mientras la estamos leyendo desde el programa principal, nos interrumpen desde una rutina de servicio a interrupción para añadir un nuevo elemento, y finalmente volvemos a actualizar los índices al completar la operación de lectura.

Analizad el problema e identificad la ventana de posible error intentando que sea lo más pequeña posible. Una vez minimizada la ventana de error, debéis emplear las nuevas llamadas al sistema de desactivar/activar interrupciones para asegurar la actualización atómica de los índices de la cola al leerla y así solucionar la condición de carrera.

Además, tal y como describe el manual del procesador en la sección 4.3, el watchdog requiere no ser interrumpido al alimentarlo. Por tanto, **antes de alimentar el watchdog hay que deshabilitar todas las interrupciones** y habilitarlas, si procede, inmediatamente después de la alimentación.

IMPORTANTE: al gestionar una condición de carrera **debéis garantizar que al salir los bits I y F de la palabra de estado tengan el mismo valor que en el momento que se llama a la gestión de condiciones de carrera**. Un error típico, y muy grave, es activarlos siempre al salir, aunque al entrar estuviesen desactivados. Como se ha comentado antes, en este proyecto vamos a tener el bit F siempre desactivado, así que no hace falta que lo

gestionemos. Pero sí que debéis comprobar que en las condiciones de carrera el bit I se restaura a su valor inicial.

E. INTRODUCCIÓN DE JUGADA POR LÍNEA SERIE

Vamos a permitir jugar con un interfaz gráfico agradable. Los comandos recibidos nos llegaran por la línea serie (UART0). Así mismo el tablero y los resultados de la partida se enviarán por la línea serie.

Seguiremos la misma estructura de dos niveles en la gestión del periférico desarrollando un módulo que gestione el `UART0` a bajo nivel, es decir todo lo que implique tocar los registros del periférico: configuración, interrupciones, etc (`linea_serie_hal`)y otro módulo que gestione las comunicaciones a alto nivel: `linea_serie_drv` (gestor de la línea serie).

Entrada

La línea serie envía carácter tras carácter en un protocolo asíncrono, nos interesa poder encapsular los distintos mensajes en comandos. Para ello se formarán tramas con delimitador de inicio ($), carga y delimitador de final (!) para poder determinar dónde empieza y dónde acaba cada mensaje.

Los comandos aceptables codificados en ASCII son:

- Acabar la partida: `$END!`
- Nueva partida: `$NEW!`
- Jugada: `$#-#!` donde #-# será el número de fila-columna codificado con dos caracteres (1..7, 1..7). Ejemplo: el comando $2-3! Indica la fila 2 y la columna 3.

Al igual que el resto de los periféricos vamos a encapsular lo dependiente del hardware.

Cuando la RSI detecte la llegada de un carácter llamara a la función de callback del driver (siguiendo el patrón que ya hemos usado). El gestor serie procesará la llegada de cada carácter con una máquina de estados. Sí recibe el carácter $ empezará a almacenar los caracteres en un buffer de tamaño 3. Si recibe el carácter ! comprobará si ha recibido un comando correcto y en tal caso generará el evento `ev_RX_SERIE` poniendo en el campo auxiliar los caracteres recibidos como carga en el mensaje. Si recibe más de 3 caracteres encenderá un led (siguiendo el mimo patrón que hasta ahora, el GPIO30 reservado como GPIO_SERIE_ERROR que se le ha indicado al inicializar el módulo). Ignorará todo lo que reciba hasta que llegue un nuevo $. En ese instante apagará el led y empezará a tratar el nuevo comando.

En resumen: `linea_serie_hal` se encargará de la recepción carácter a carácter (nivel físico), `linea_serie_drv` comprobará que la trama llegue correctamente (nivel de enlace). Posteriormente el planificador le mandará el mensaje al juego para que procese el comando y haga lo que le han dicho que debe hacer (nivel aplicación).

Salida

A lo largo de la partida del juego queremos visualizar instrucciones de juego, el tablero y otros datos que consideremos pertinentes. Como no tenemos una pantalla vamos a utilizar la línea serie.

El gestor recibirá el encargo de enviar un array de char (tipo C) en la función `linea_serie_drv_enviar_array`. El gestor dispondrá de un **buffer interno** e ira enviando por la línea serie carácter a carácter utilizando las funciones de `linea_serie_hal` hasta enviar todo el buffer. Cuando acabe de enviar todo el buffer se generará el evento ev_TX_SERIE. Se debe realizar sin esperas activas, es decir ni el gestor ni el juego deben quedar a la espera obstaculizando la ejecución orientada a eventos ya que podríamos tener otras cosas en marcha.

- La función `linea_serie_drv_enviar_array` inicializará las estructuras necesarias y enviará el primer carácter.
- La función interna `linea_serie_drv_continuar_envio` enviará el siguiente carácter hasta llegar al final del buffer.

Importante: tened en cuenta qué debe ocurrir si nos mandan un array vacío. Es una entrada válida y no debe generar ningún malfuncionamiento.

Iremos implementado funciones específicas para la salida del juego, por ejemplo, para el demostrador vamos a diseñar la función `conecta_K_visualizar_tablero,` y la función `conecta_K_visualizar_tiempo.` Más adelante diseñaremos otras como `conecta_K_visualizar_instrucciones`, o `conecta_K_visualizar_final`. Dimensionar el tamaño del buffer interno del gestor adecuadamente para poder enviar todo el tablero con una única invocación de `linea_serie_drv_enviar_array`.

Ejemplo de interacción entre módulos. Cuando el juego necesite presentar un tablero por pantalla invocará a la función `conecta_K_visualizar_tablero`, con un puntero al tablero a visualizar. Esta función, generará un array de caracteres con el formato que hayáis elegido para visualizar el tablero y lo enviará a `linea_serie_drv_enviar_array` que lo copiará en su buffer interno. A partir de ahí los caracteres se irán enviando uno a uno sin que el juego tenga que hacer nada. Cuando se envíe el último carácter se generará el evento `ev_TX_SERIE` y el juego sabrá que su orden ha sido atendida.

¿Qué ocurre si se solicita el envío de un array sin haber terminado el anterior? Lo más sencillo en la versión inicial es permitir un único envío en cada instante y esperar a que llegue el ev_TX_SERIE antes de mandar la siguiente petición. Como **apartado opcional**, podéis plantear esquemas que permitan almacenar varias peticiones simplificando la máquina de estados. En ese caso `linea_serie_drv` debe incluir una función que nos diga la cantidad de espacio libre. Si se envía un array nuevo que no cabe `linea_serie_drv` ignorará la solicitud.

F. DEMOSTRADOR

Para demostrar el funcionamiento de todos los módulos e ir empezando a enlazar con el juego vamos a crear un demostrador.

Al módulo juego de la práctica anterior le vamos a ir incorporando las funciones de conecta_k realizadas en la práctica 1 (conecta_k_2023). En concreto debemos incorporar el código y reutilizar las funciones ya realizadas para ser capaces de cargar

el tablero inicial y visualizarlo en el nuevo entorno (en memoria como en P1 y por línea serie).

Modificar **`juego_inicializar`** para que inicialice el juego conecta_k y cargue el tablero inicial de test.

Vamos a añadir un comando serie para visualizar el tablero (`$TAB!),` cuando juego trate el evento `ev_RX_SERIE` en **`juego_tratar_evento`** invocará a la función `conecta_K_visualizar_tablero` para visualizar el tablero.

Debéis ser capaces de enviar el tablero codificando las celdas con caracteres ASCII fácilmente entendibles. Podéis elegir el formato que queráis. Por ejemplo:

```
6| | | | | | | |
5| | | | | | | |
4| | | | | | | |
3| | | |N| | | |
2| | |B|B| | | |
1| | |N|B| | | |
-|1|2|3|4|5|6|7|
```

Siendo en este ejemplo "B" el jugador 1 y "N" el jugador 2.

Además, se leerá el tiempo del sistema antes y después de visualizar el tablero, usando la función **`clock_getus.`** Sabremos cuándo ha terminado el envío del tablero porque el juego recibirá el evento ev_TX_SERIE. El tiempo transcurrido se presentará por pantalla con la función `conecta_K_visualizar_tiempo` que recibe un entero y debe ser capaz de convertirlo en un array de caracteres para poder enviarlo a `linea_serie_drv_enviar_array`.

Finalmente, el juego entrará en un bucle infinito: while (1). Aquí queremos ver cómo el watchdog es capaz de resetear el sistema. Configuradlo para que salte al cabo de unos segundos. Ajustad el valor para que dé tiempo a enviar el comando, y presentar el tablero antes de que salte el WD.

MATERIAL DISPONIBLE:

En la página de la asignatura en Moodle podéis encontrar códigos de ejemplo. Os recordamos, podrán aparecer indicaciones a lo largo de la práctica, se recomienda consultar Moodle con frecuencia y en particular la información que vaya apareciendo en la wiki.

EVALUACIÓN DE LA PRÁCTICA

Está práctica se presentará durante la semana 12 de clase (EINA), se publicarán los detalles en Moodle.