Desarrollo del juego Reversi

Para el procesador LPC2105

Javier Herrer Torres (NIP: 776609)
Samuel Torres Fau (NIP: 780505)

Proyecto Hardware
Grado en Ingeniería Informática

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza
Curso 2020/2021

Resumen

Se ha completado el trabajo planteado en la asignatura. Se ha obtenido un juego completo (Reversi8) que funciona de forma autónoma y correcta en el procesador LPC2105. El proyecto realizado engloba la utilización de diferentes componentes y periféricos del procesador. Además, se han gestionado múltiples interrupciones, y los problemas de concurrencia que estas originan.

Se ha trabajado estrechamente con la documentación del manual del procesador para la configuración de los periféricos del mismo (temporizadores, la línea de serie, los botones externos, el RTC y Watchdog, y el GPIO).

Todo el proyecto se ha desarrollado de manera modular, de forma que el sistema sea lo más desacoplado posible. Se ha conseguido por tanto, un código escalable, reutilizable y fácilmente mantenible. Para ello se ha hecho uso principalmente de una cola de eventos (tolerante a condiciones de carrera) que encapsula la interacción entre el juego y los periféricos.

La depuración del juego es correcta y cumple con los requisitos planteados eliminando esperas y condiciones de carrera. Se obtiene por tanto, un juego que interactúa con el usuario por línea de serie y que gestiona en segundo plano los periféricos y sus interrupciones.

Como principales ventajas destacar de nuevo la modularidad del proyecto, además de la gestión de las condiciones de carrera generadas por las múltiples interrupciones de los periféricos.

Índice

1. Introducción

Esta práctica se enmarca dentro de la asignatura de Proyecto Hardware como continuación de las dos realizadas anteriormente. Se pretende completar el trabajo realizado de modo que se obtenga como resultado un juego completo autónomo y funcionando correctamente en el procesador LPC2105.

En esta práctica aparecen problemas más complejos como, el diseño e implementación de una gestión completa y robusta de periféricos, la composición de

una arquitectura de sistema que aproveche los modos del procesador, la implementación y el uso de llamadas al sistema para interactuar con un dispositivo (un temporizador) y para activar y desactivar las interrupciones. En concreto, los periféricos con los que se trabaja son temporizadores genéricos (timer0 y timer1), línea serie, botones externos, RealTimeClock, el WatchDog y el GPIO.

También se concebirá e implementará un protocolo de comunicación robusto para el puerto de serie, y se identificarán condiciones de carrera y su causa, así como sus soluciones. El problema de las condiciones de carrera será tratado como un problema de sección crítica, por lo que se priorizará un acceso a las variables críticas en exclusión mutua.

Para evitar esperas activas en las escrituras en linea serie, se configurará el periférico UART0 para que las escrituras sean por interrupción, empleando un buffer intermedio.

2. Objetivos

Se plantearon varios objetivos a cumplimentar en las prácticas 2 y 3.

2.1. Práctica 2

En esta práctica se introdujo por primera vez la interacción del usuario con el juego mediante periféricos (IO). El objetivo general de la práctica es configurar y emplear estos periféricos para jugar al reversi, y teniendo como objetivo fundamental que el código sea modular y robusto.

- Empleo del timer 1 para la medición de tiempos, con la máxima precisión y el menor número de interrupciones posible
- Programación de la GPIO. Desarrollo de una librería para poder leer y escribir en los pines de GPIO, así como configurar los mismos.
- Diseño e implementación de una cola de eventos. Este objetivo es especialmente importante, ya que el juego va a ser desarrollado como un programa orientado a eventos, por lo que la gestión de escritura y lectura de eventos es un aspecto crítico del desarrollo.
- Como apoyo a la gestión por eventos, configuración y empleo del timer0 para la creación de alarmas y alarmas periódicas.
- Introducción de las interrupciones externas. Son empleadas para despertar al procesador de POWER-DOWN, así como para confirmar o cancelar movimientos, y para empezar las partidas. Además, ha de implementarse una gestión propia para la gestión de pulsaciones, debido a que la configuración mediante los registros `EXTPOLAR` y `EXTMODE` no funcionan en el simulador.
- Empleo de los modos IDLE y POWER-DOWN en el procesador.

- Integración de todo lo anterior:
  1. Se crea un planificador para jugar, mediante gestión por eventos, a reversi.
  2. Se interrumpe mediante la alarma periódica cada 10 microsegundos.
  3. Programación de la GPIO para la lectura de la fila y columna en la que el usuario coloca ficha (En la práctica 3 esto es descartado por la introducción de la linea serie).
  4. Verificación del movimiento del usuario (si es un movimiento no válido no lo realiza).
  5. Un led parpadea cada 0.25s durante 3s cuando el usuario ha introducido un movimiento válido o ha pasado. Pasados los 3 segundos se lleva a cabo. Si durante este tiempo, llega una EINT0, el movimiento se confirma prematuramente, mientras que si llega EINT1 se cancela.
  6. Cuando no hay eventos (no hay nada que gestionar) y el juego no esté a la espera de ningún periférico (se está esperando el movimiento del usuario, y no se está gestionando ninguno de los botones), el procesador pasa al modo POWER-DOWN para el ahorro de energía. En caso contrario, si detecta que no hay nuevos eventos, pasa a modo IDLE.
- El código debe ser modular.

2.2. Práctica 3

A partir del programa desarrollado en las prácticas anteriores, se introducen nuevos periféricos e interacciones, con el fin de desarrollar finalmente juego completo.

- Configuración y empleo del Real-Time Clock y del WatchDog, este último para reiniciar la partida si no se interactúa con el juego en 1 minuto.
- Introducción de system calls (interrupciones software). Serán empleadas para modificar el `CPSR`, permitiendo desactivar y activar las IRQs y/o FIQs.
- Modificación del timer0. Pasará a ser una FIQ (no vectorizada), ya que se trata del periférico más crítico.
- Debido al numero de periféricos que interrumpen y guardan eventos y, sobretodo, el tratamiento de timer0 como FIQ, aparecen una serie de condiciones de carrera en la funciones de la cola de eventos. Debe plantearse una solución apropiada para este problema, que será tratado como un problema de **sección crítica**.

- Introducción de comandos por linea serie. Se emplea la UART0 para introducir jugadas y comandos, de modo que el usuario interactúa con linea serie en vez de con los pines del GPIO. Se crean módulos independientes para la lectura de caracteres, para la detección de comandos y para el reconocimiento de dichos comandos (estructura modular).
- Uso de la UART0 para la interacción con el usuario. Además de leer, se escribirá en ella para mostrar al usuario el tablero, los resultados de partidas y las distintas cabeceras del juego. Adicionalmente, las escrituras se harán mediante un buffer e interrupciones, ya que en caso contrario toman mucho tiempo (esperas activas).
- Introducción de todo lo anterior en el juego, partiendo de la base construida en la práctica 2.

3. Metodología

3.1. Elección de versión de patrón volteo

La primera decisión tomada tras la primera práctica fue el seleccionar una de las versiones de la función patron_volteo. Debido a que la versión más optima, tanto en número de instrucciones como en tiempo de ejecución fue la función en **C** con el compilador ajustado para **-O3 -OTime**, se decidió emplear esta misma.

3.2. timer1

Siguiendo con los requerimientos planteados, se configura el timer1 para que interrumpa lo menos posible. Para ello, se da a su Match Register el máximo valor y se guarda en el módulo una variable con los periodos por los que ha pasado (ver A.2).
Para leer el tiempo con precisión, se empleará dicha variable y el registro `T1TC`:

$$\frac{Periodo \cdot TicksPeriodo + T1TC}{TickMS}$$

3.3. GPIO

Se configuran los pines (salida/entrada) mediante `IODIR`, y para leer y escribir en ellos se hace uso de los registros `IOSET` e `IOCLR` (ver A.3).

3.4. Cola de eventos

Este apartado es el más crítico, ya que vamos a desarrollar el programa orientado a eventos, por lo que una gestión adecuada de los mismos es fundamental (ver A.4). El tipo colaEventos es el siguiente:

```
// Struct cola circular de eventos
struct colaEventos
{
    // Último evento introducido
    int8_t ultimo;
    // Evento mas antiguo sin procesar
    int8_t primero_no_procesado;
    // Eventos
    uint32_t eventos[MAX_SIZE_QUEUE];
    // Momento en que se guarda el evento
    uint32_t tiempos[MAX_SIZE_QUEUE];
};
```

Como se ve en la estructura, se guardan índices al último evento introducido y al evento más antiguo sin procesar, con el objetivo de detectar posibles desbordamientos (intentar sobreescribir un evento no tratado).
Los índices permiten escribir y leer de la cola de eventos. Sin embargo, con la aparición de las FIQs (timer0) y otros eventos creados en modo usuario, aparecen una serie de condiciones de carrera que han de ser eliminadas.

3.4.1. Eliminación de las condiciones de carrera en la cola de eventos

Para solucionar este problema, va a tratarse como un problema de sección crítica, y por ello, se hará uso de variables auxiliares y se implementarán las funciones `lock()` y `unlock()` en un nuevo módulo (ver A.5). Estas funciones a su vez hacen uso de llamadas al sistema para desactivar y activar las diferentes interrupciones (modificación de los bits I y F):

- `lock()` guarda el estado de los bits I y F, y pone a ambos a 1 (desactiva cualquier interrupción).
- `unlock()` Devuelve el estado anterior (el guardado por `lock()`) a los bits I y F.

Por tanto, siempre que se lea y/o modifique un índice de la cola, el protocolo será el siguiente:

```
// SC
lock();
// Variable auxiliar para control de concurrencia
aux = (eventos.ultimo + 1) & mod_max;
eventos.ultimo = aux;
unlock();
// FIN SC
```

Gracias a la desactivación de las interrupciones, se asegura que nadie más va a interrumpir la escritura, y gracias al uso de auxiliares, se facilita un buen empleo de los índices desde los distintos componentes que generan eventos.

3.5. Interrupciones externas

Lo más relevante de la gestión de interrupciones externas es que, como se ha dicho en el apartado de objetivos, `EXTPOLAR` y `EXTMODE` no funcionan en el simulador, por lo que tiene que hacerse una gestión paralela al juego para que los botones no interrumpan continuamente si se dejan pulsados. La máquina de estados de dicha gestión se encuentra representada en la figura 1.

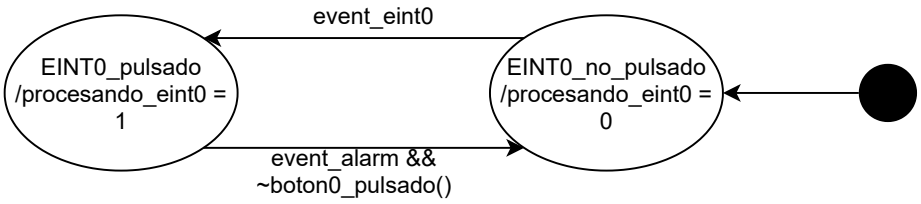

Figura 1: Máquina de estados de gestión de pulsación

Cuando el botón es pulsado, se pasa el evento al módulo para iniciar la gestión del mismo. El módulo desactiva las interrupciones del botón para evitar interrupciones consecutivas y pone una alarma (ver A.7). Cuando llega dicha alarma comprueba si sigue pulsado: Si es el caso vuelve a poner la alarma, y en caso contrario habilita de nuevo las interrupciones del botón.

3.6. Introducción de modos IDLE y POWER_DOWN

Siguiendo con lo mostrado en el ejemplo, y haciendo uso del código de restauración de la frecuencia del PLL, se crea un módulo (ver A.8) para poder pasar a los modos IDLE y POWER_DOWN.

3.6.1. Condición de carrera adicional

Se ha descubierto una posible condición de carrera independiente de la condición de carrera de al cola de eventos que será gestionada más adelante.

El bucle del planificador del juego extrae un evento de la cola de eventos por iteración. Es posible que el planificador lea de la cola en un momento en que no hay nada que leer, y justo tras esto llega una interrupción externa (se pulsa un botón o los dos). Esta sucesión de eventos provocará que las interrupciones externas (tanto EINT0 como EINT1), se hayan desactivado y se pase al modo POWER_DOWN (si se pasa a IDLE no hay problema, el temporizador periódico despertará al procesador), por lo que ya no se podrán emplear dichas interrupciones para despertar al procesador, es el único modo que se ha dispuesto para ello.

Para solucionar este problema se fuerza antes de entrar a POWER_DOWN que las interrupciones externas estén activadas (ver A.8).

3.7. Llamadas al sistema

Se introducen las siguientes llamadas al sistema:

- `enable_isr`: Activa IRQs.
- `disable_isr`: Desactiva IRQs.
- `enable_isr_fiq`: Activa IRQs y FIQs.
- `disable_isr_fiq`: Desactiva IRQs y FIQs.
- `lock_isr_fiq`: Desactiva IRQs y FIQs, y devuelve un entero para poder restaurar el estado anterior.
- `enable_fiq`: Activa FIQs.

Todas ellas se definen en el fichero header sys_calls.h del siguiente modo:

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

Y se implementan en el fichero ensamblador SWI.s (ver A.11). Para activar y desactivar las interrupciones se trabaja con los bits I y F del registro de palabra SPSR, es decir, el registro de palabra del modo desde el que se ha llamado al system call.

3.8. timer0 como FIQ

Para la práctica 3 se exigió la conversión de timer0, ya configurado y funcionando en la práctica (ver A.2), a FIQ. Por ello, y con el objetivo de evitar más complicaciones así como de intentar que la ejecución de la RSI sea lo más reducida en el tiempo posible, se decidió configurar el timer0 como RSI no vectorizada. Para la ejecución de la RSI se llama a esta desde el FIQ_Handler de Startup.s:

```
FIQ_Handler     B       temporizador0_rsi
```

La conversión del tratamiento del timer0 a FIQ será el principal causante de la aparición de condiciones de carrera, por lo que es esencial tener esto en cuenta e introducir las modificaciones apropiadas (ver 3.4.1)

3.9. Lectura de comandos de línea serie

Con el objetivo de crear un código lo más modular posible, se divide la lectura e interpretación de comandos en 3 fases: lectura de carácter, reconocimiento de comando y procesamiento de comando.

3.9.1. Lectura de carácter

En el propio módulo de la UART0, además de la configuración inicial de la línea serie, se implementa la RSI de lectura de la UART0. Cada vez que se introduzca un carácter por linea serie, la RSI lo leerá y guardará un evento de carácter leído (en el campo auxiliar del evento guarda el propio carácter) (ver A.12.2).

3.9.2. Reconocimiento de comando

Se crea un nuevo módulo para la lectura de comandos completos (ver A.13). Cada vez que el planificador lea de la cola de eventos un evento de carácter leído, se lo pasará al módulo creado, para que este reconozca comandos con formato válido ($\#C_1C_2C_3$! en el caso del reversi8). Una vez reconocido un comando introducirá en la cola un evento de comando reconocido, guardando como campo auxiliar los 3 caracteres del comando leído.

3.9.3. Procesamiento de comando

Finalmente, dentro del propio módulo del reversi8, se crea el procesador de comandos (ver A.1.3), función que se encarga de reconocer comandos propios del reversi8. Si recibe un comando válido, guarda el evento correspondiente. Si es un evento inválido, es decir, no existe o bien es un comando de movimiento con un checksum incorrecto, no hace nada. La máquina de estados del procesador puede verse en la figura 2.

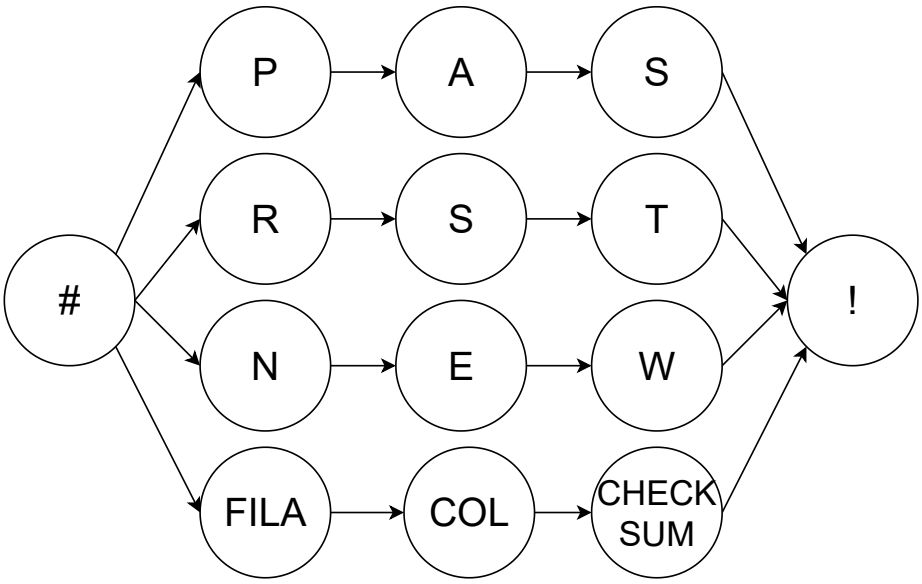

Figura 2: Máquina de estados para el reconocimiento de comandos válidos de reversi8

3.10. Escritura en línea de serie (adicional)

Además de recibir los comandos por la línea de serie (`UART0`), el tablero y los resultados de tiempo también se enviarán por ella. El tablero será enviado cada vez que se actualice para poder ser visualizado en pantalla. Para ello, se codifi-

carán las celdas con caracteres ASCII fácilmente entendibles mostrándolos por filas.

El envío se puede realizar carácter a carácter, pero es muy ineficiente. Por ello, se ha realizado una función que permite enviar un vector de caracteres mediante interrupciones, evitando esperas activas.

Al inicializar la UART0, se habilitan las interrupciones `THRE` (*Transmitter Holding Register Empty*) en el registro `U0IER`. Esto evitará la espera activa que se realizaba en el bucle `while (!(U0LSR & 0x20))` del método `sendchar`.

Por otro lado, se contará con un *buffer* circular que almacenará los caracteres pendientes de envío. La función `sendstring` hará uso de este *buffer* escribiendo una cadena completa en él.

Ahora, cada vez que se produzca una interrupción se deberá identificar para clasificarla según sea `RDA` o `THRE` mediante el registro `U0IIR`. Si se trata de una interrupción de tipo `THRE`, se escribirá un carácter mediante `sendchar` pero ahora sin la espera activa (ver A.12.3).

Por último, el envío de los resultados y del tablero se realizará desde el módulo reversi8 mediante las funciones `sendTablero` y `mostrar_estadisticas` que escriben los caracteres deseados en el *buffer*.

3.11. Planificador del juego

Siguiendo las directrices marcadas a partir de la práctica 2, y siguiendo con el objetivo de que el código desarrollado sea lo más modular posible, se reorienta el juego a un programa dirigido por eventos, luego la cola es el elemento más crítico del mismo.

Una vez se realizan las configuraciones necesarias de periféricos, el *main* llama al planificador del juego. El planificador consiste en un bucle infinito que trata de leer un evento de la cola en cada iteración. Dependiendo del evento leído, realizará unas acciones u otras.

3.11.1. No hay evento

Si lee un evento *none*, significa que no hay eventos en la cola. Por ello, el planificador pondrá al procesador en un modo de bajo consumo: POWER_DOWN, si ninguna máquina de estados está a la espera de alarmas o alarmas periódicas, o IDLE, en caso contrario.

3.11.2. Interrupción externa

Si lee un evento de este tipo, quiere decir que ha habido una pulsación del botón, por lo que llama (pasando el propio evento leído), a las funciones de gestión del botón correspondiente y a la función de gestión del juego `gestion_reversi8` (ver A.1.2), es decir, las máquinas de estados a las que puede interesar ser conscientes de la llegada de una interrupción externa.

3.11.3. Alarma periódica

Si llega un evento de alarma periódica se llamarán a las funciones cuyas máquinas de estados necesiten conocer de la llega de dicho evento, es decir, la máquina del juego (tiempo de espera de 3s antes de confirmar movimientos) y la del LED que parpadea cada 0.25s.

3.11.4. Alarma

Las máquinas de gestión de pulsación de los botones son las que necesitan saber de la llegada del evento alarma, por lo que se llama a las mismas pasando el evento extraído de la cola.

3.11.5. Carácter leído

Como se ha dicho en el apartado correspondiente 3.9.2, se pasa el carácter a la máquina de estados del reconocedor.

3.11.6. Comando reconocido

Se llama al procesador de comandos pasándole el auxiliar del evento, es decir, los 3 caracteres reconocidos.

3.11.7. Resto de casos

El resto de casos se trata de los diferentes comandos reconocidos por el procesador de comandos. Por esto mismo, dado que son acciones que introduce el usuario para interactuar con el juego, se llama a la máquina de gestión del juego pasándole el propio evento.

3.12. Máquina de estados del juego

Como se ha dicho anteriormente, el juego consiste en una máquina de estados (ver figura 3) a la que se le facilitan los distintos eventos leídos por el planificador (solo si el evento puede afectar al juego) para que el juego progrese de acuerdo a dichos eventos.

4. Resultados

La depuración del juego es correcta y cumple con los requisitos planteados en las dos prácticas. El juego inicia y termina de manera ordenada. Se minimiza el consumo de la placa entrando a power-down al terminar una partida y a modo idle siempre que no hay nada pendiente de procesar. No existen esperas activas.

También se gestiona la activación y desactivación de las rutinas de servicio de periféricos como llamadas al sistema. La entrada de los valores fila y columna se realiza a través de la línea de serie, usada también para enviar al usuario el tablero para visualizar los movimientos por pantalla. Al final de la partida, se

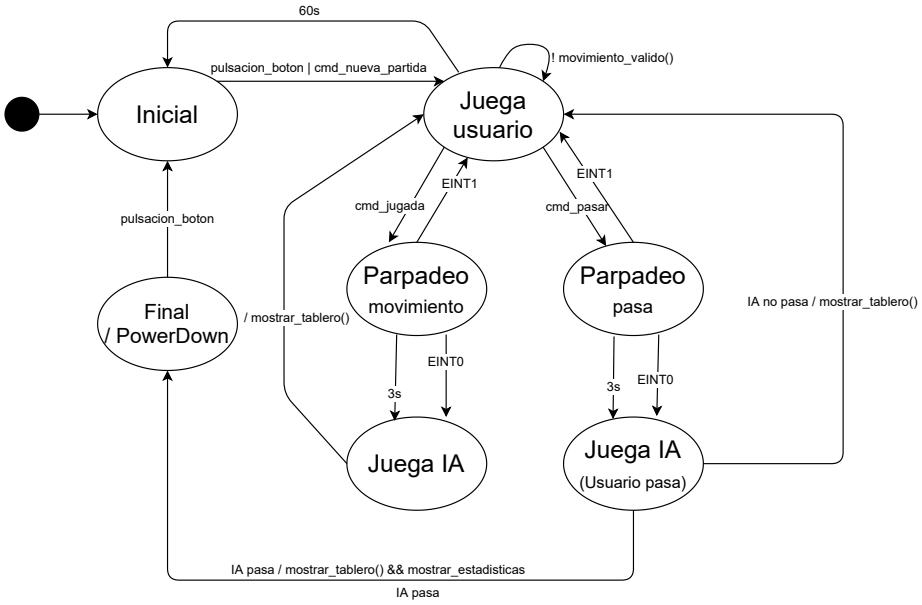

Figura 3: Máquina de estados del juego reversi8

envían el tiempo de juego y el tiempo total de cálculo a través de ella. Por otro lado, también se da la opción de confirmar, pasar turno y cancelar mediante los botones.

4.1. Código modular

Como se ha comentado anteriormente, uno de nuestro objetivos principales en el desarrollo del proyecto ha sido el generar un **código modular**.

Este objetivo se ha cumplimentando gracias al desarrollo y empleo de diversos módulos, que interactúan entre si. Los diferentes componentes y como interactúan entre si pueden verse en la figura 4.

El resultado de haber creado un código modular es que el sistema desarrollado es desacoplado, es decir, se trata de un código **escalable** (pueden añadirse nuevos módulos y la funcionalidad de los anteriores no se verá afectada), **reutilizable** (no se repiten fragmentos de código, y por ende el tamaño del programa es menor) y **fácilmente mantenible** (Si se detecta un fallo o se quiere actualizar alguna componente, basta con modificar el módulo apropiado). Además de lo anterior, el código modular permite tener un proyecto más **organizado** y limpio, evitando entremezclar distintas funcionalidades en los mismos fragmentos del programa.

Sin embargo también es cierto que al hacer el código modular, existe una **mayor sobrecarga**: Se produce un mayor número de llamadas a funciones, ya que cada funcionalidad del sistema está presente en su módulo propio, aunque

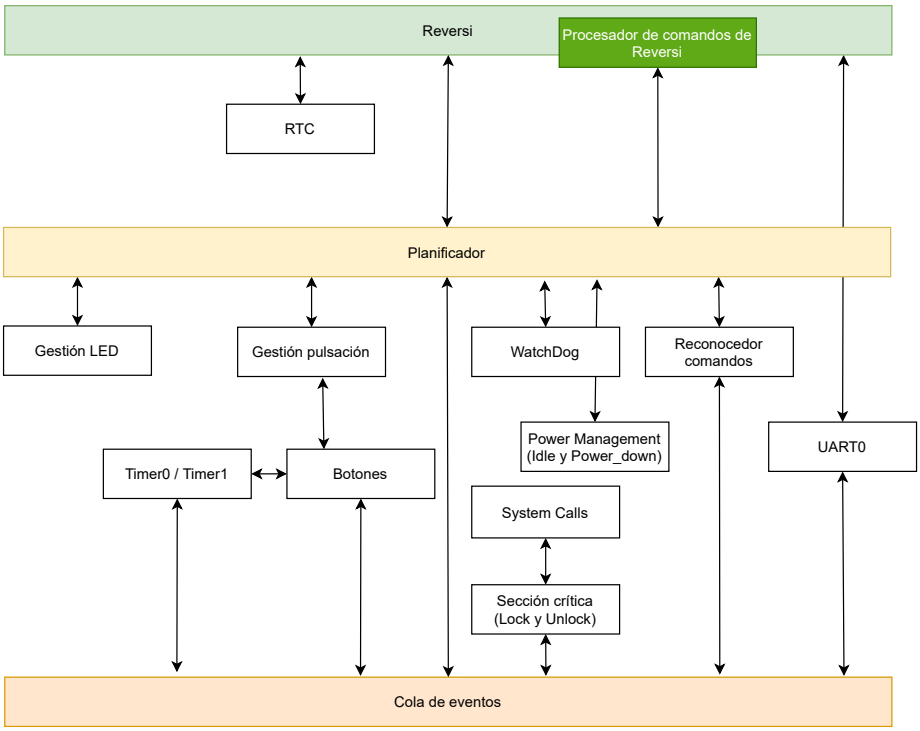

Figura 4: Paquetes del proyecto y dependencias entre ellos (Solo aparecen los paquetes más relevantes)

con las opciones apropiadas, el compilador es capaz de paliar este problema.

4.2. Eliminación de esperas activas

Gracias al empleo de los modos del procesador IDLE y POWER_DOWN, se evita una espera activa en el planificador: Si detecta que no hay ningún evento en la cola, se pone al procesador en uno de estos dos modos de bajo consumo, consiguiendo además un menor gasto en electricidad.

Además, inicialmente las escrituras en linea serie se hacía mediante esperas activas (comprobaba continuamente si podía escribir y, una vez podía, escribía). Estas esperas traían problemas serios al programa, ya que la cola de eventos se llenaba antes de que hubiese terminado la escritura, provocando el colapso del mismo, **Overflow**.

Gracias al empleo de un buffer intermedio, y la configuración de las escrituras por interrupción A.12, y de los modos de bajo consumo del procesador, se consigue solucionar el problema comentado, además de reducir el consumo energético.

4.3. Eliminación de condiciones de carrera

Debido a la existencia en el sistema de numerosas fuentes de interrupción (periféricos), así como de los módulos que generan eventos del programa, aparecen condiciones de carrera en la cola de eventos del sistema. Como se ha comentado en metodología, se ha tratado esta cuestión como un problema de sección crítica.

Gracias al acceso en exclusión mutua preparado, se consigue que las variables no puedan ser accedidas y modificadas por diferentes componentes a la vez, por lo que se consigue eliminar las condiciones de carrera en la cola.

También gracias a la activación de las interrupciones externas antes de poner el procesador en POWER_DOWN se consigue eliminar la condición de carrera detectada en dicha parte.

4.4. Juego empotrado final

El resultado final y más obvio del proyecto es el propio juego del reversi desarrollado. El juego desarrollado cumple con los diferentes requisitos planteados en las prácticas 2 y 3, y es completamente autónomo: El usuario puede jugar con el mismo empleando únicamente la linea serie, aunque si desea confirmar o cancelar movimientos también tendrá que emplear los botones.

5. Conclusiones

Una de las principales ventajas del proyecto realizado es su gran modularidad. Se ha conseguido un código muy desacoplado que permite su reutilización y un fácil mantenimiento.

Se ha conseguido integrar en el sistema una cantidad considerable de periféricos, que funcionan paralelamente al propio juego.

Además, se ha realizado una buena gestión de las condiciones de carrera que generan las interrupciones, encontrando incluso algunas adicionales de las planteadas en la práctica.

También se ha realizado como apartado opcional la escritura en la línea de serie mediante interrupción evitando esperas activas.

Por otro lado, como posibilidad de mejora, se podrían realizar *Toolboxes* conectadas al GPIO que emulen botones. También se podría haber desarrollado la llamada a la RSI del timer0 como FIQ creando un módulo separado, o incluso implementando la función en ensamblador.

A. Anexo: Código fuente comentado

A.1. Módulo de Reversi

A.1.1. Planificador del juego

```
/*
 * Lee un evento de la cola y realiza la gestión oportuna
 *
 * Cambia el procesador a modo IDLE o POWER-DOWN
 */
void planificador() {
    uint32_t auxData;
    uint32_t times;
    event_t evento;

    int idle_reversi8 = 1;
    int idle_gestion_boton0 = 0;
    int idle_gestion_boton1 = 0;
    int idle_escribiendo = 0;

    mostrarInicioJuego();
    // Interrupcion cada 10 ms
    temporizador_interrupcion_periodica(10);
    WD_init(60);

    // Descomentar para empezar el juego sin pulsacion
    //idle_reversi8 = gestion_reversi8(event_eint0);

    while(1)
    {
        evento = leer_evento(&auxData, &times);

        if( evento == event_none )
        {
            idle_escribiendo = uart0_leyendo();
            if( (idle_reversi8 | idle_gestion_boton0 |
                 idle_gestion_boton1 | idle_escribiendo) != 0)
            {
                PM_idle();
            }
            else
            {
                PM_power_down();
            }
        }
```

```
        else if( evento == event_eint0 )
        {
            // Gestionamos el boton 0
            idle_gestion_boton0 = gestion_eint0_pulsado(evento);
            // Gestionamos el juego
            idle_reversi8 = gestion_reversi8(evento,0);
        }
        else if( evento == event_eint1 )
        {
            // Gestionamos el boton 1
            idle_gestion_boton1 = gestion_eint1_pulsado(evento);
            // Gestionamos el juego
            idle_reversi8 = gestion_reversi8(evento,0);
        }
        else if( evento == event_alarma_periodica )
        {
            // Gestionamos el juego
            idle_reversi8 = gestion_reversi8(evento,0);
            gestion_led(evento);
        }
        else if( evento == event_alarma)
        {
            // Gestionamos el boton 0
            idle_gestion_boton0 = gestion_eint0_pulsado(evento);
            // Gestionamos el boton 1
            idle_gestion_boton1 = gestion_eint1_pulsado(evento);
        }
        else if( evento == event_char_leido)
        {
            // Gestionamos el boton 0
            procesar_caracter(auxData);
        }
        else if( evento == event_comando_reconocido )
        {
            procesar_comando(auxData);
        }
        else
        {
            // Gestionamos el juego
            idle_reversi8 = gestion_reversi8(evento, auxData);
        }
    }
}
```

A.1.2. Gestión del juego

```
/*
 * Realiza las acciones oportunas dado un evento
 * determinado
 *
 * Devuelve 0 si no necesita IDLE
 * Devuelve 1 si necesita IDLE
 */
int gestion_reversi8(event_t evento, uint32_t aux)
{
    static estado state = INICIAL;
    static int parpadeos_restantes = 0;
    static int interrupciones_restantes = 0;
    //numero de movimientos
    static int movUsuario = 0;
    static int movIA = 0;
    //tiempo total de IA
    static int time_init = 0;
    static int time_final = 0;
    static int timeIATotal = 0;
    //tiempo inicial
    static uint8_t segIni = 0;
    static uint8_t minIni = 0;

    // 25 * 10ms
    int interr_parpadeo = 25;
    // Parpadeos totales
    int veces_parpadeo = 12;
    // Por defecto, siempre IDLE
    int power_mode = 1;

    // la máquina ha conseguido mover o no
    int done;
    // número de fichas de cada color
    int blancas;
    int negras;
    //tiempo de partida
    uint8_t segFin;
    uint8_t minFin;
    // fila y columna elegidas por la máquina para
    //  su movimiento
    int8_t f, c;


    if(evento == event_comando_acabar)
```

```
{
    state = FINAL;

    segFin = rtc_leer_segundos() - segIni;
    minFin = rtc_leer_minutos() - minIni;

    contar(tablero, &blancas, &negras);
    mostrar_estadisticas(movUsuario, movIA, blancas,
                         negras, 1, timeIATotal, segFin,
                         minFin);

    power_mode = 0;
}
else {
    switch(state)
    {
        case INICIAL:
            if(evento == event_eint0 || evento == event_eint1
               || evento == event_comando_nueva)
            {
                init_table(tablero, candidatas);
                // Reset movimientos
                movIA = 0;
                movUsuario = 0;
                // Reset timers
                timeIATotal = 0;
                segIni = rtc_leer_segundos();
                minIni = rtc_leer_minutos();
                // Pasa a usuario
                state = USUARIO;
                // Imprime el tablero inicial
                sendTablero();
                // Alimenta el WD
                WD_feed();
            }
            break;


        case USUARIO:
            // Caso usuario introduce un movimiento (boton)
            if(evento == event_comando_jugada)
            {
                // Obtiene de aux la fila y columna
                fila = aux >> 8;
                columna = aux & 0xf;
```

```
                if(movimiento_valido(fila, columna) == 1)
                {
                    tablero[fila][columna] = FICHA_NEGRA;
                    usuario_ha_pasado = 0;
                    state = PARPADEO;
                    parpadeos_restantes = veces_parpadeo;
                    interrupciones_restantes = interr_parpadeo;
                }
                WD_feed();
            }
            // Caso usuario pasa
            else if(evento == event_comando_pasar)
            {
                usuario_ha_pasado = 1;
                state = PARPADEO;
                parpadeos_restantes = veces_parpadeo;
                interrupciones_restantes = interr_parpadeo;
                WD_feed();
            }
            break;


        case PARPADEO:
            if(evento == event_alarma_periodica)
            {
                interrupciones_restantes--;
                if(interrupciones_restantes == 0)
                {
                    interrupciones_restantes = interr_parpadeo;
                    parpadeos_restantes--;
                    // Solo parpadea si el usuario ha
                    //  introducido movimiento
                    if(usuario_ha_pasado == 0)
                    {
                        if(tablero[fila][columna] == CASILLA_VACIA)
                        {
                            tablero[fila][columna] = FICHA_NEGRA;
                        }
                        else
                        {
                            tablero[fila][columna] = CASILLA_VACIA;
                        }
                    }
                }
            }
```

```
        if(parpadeos_restantes == 0 ||
             evento == event_eint0)
        {
            if(usuario_ha_pasado == 0)
            {
                movUsuario++;
                tablero[fila][columna] = FICHA_NEGRA;
                actualizar_tablero(tablero, fila, columna,
                                   FICHA_NEGRA);
                actualizar_candidatas(candidatas, fila,
                                      columna);
            }
            // Pasamos a movimiento de la IA
            state = IA;
        }
        else if(evento == event_eint1)
        {
            state = USUARIO;
            sendTablero();
            WD_feed();
        }
        break;

    // Seguramente vendra llamado por una interr
    //  periodica, pero no se necesita
    case IA:
        time_init = temporizador_leer();
        done = elegir_mov(candidatas, tablero, &f, &c);

        // Usuario pasa y IA pasa
        if (usuario_ha_pasado == 1 && done == -1)
        {
            state = FINAL;
            sendTablero();
            segFin = rtc_leer_segundos() - segIni;
            minFin = rtc_leer_minutos() - minIni;
            contar(tablero, &blancas, &negras);
            mostrar_estadisticas(movUsuario, movIA, blancas,
                                 negras, 0, timeIATotal,
                                 segFin, minFin);
            power_mode = 0;
        }
        // IA no ha pasado
        else if(done != -1)
        {
            movIA++;
```

```
                    tablero[f][c] = FICHA_BLANCA;
                    actualizar_tablero(tablero, f, c, FICHA_BLANCA);
                    actualizar_candidatas(candidatas, f, c);
                    state = USUARIO;
                    sendTablero();
                    WD_feed();
                }
                // IA ha pasado pero usuario no
                else
                {
                    state = USUARIO;
                    sendTablero();
                    WD_feed();
                }
                // Tiempo de la IA
                time_final = temporizador_leer() - time_init;
                timeIATotal = time_final + timeIATotal;

                break;

            case FINAL:
                if(evento == event_eint0 ||evento == event_eint1) {
                    state = INICIAL;
                    mostrarInicioJuego();
                }
                else {
                    power_mode = 0;
                }
                break;

            default:
                // PANIC
                while(1);
                // Aqui no debería llegar
                //printf("CASO RARO");
        }
    }
    return power_mode;
}
```

A.1.3. Procesador de comandos

```
/*
 * Procesa un comando siguiendo la sintaxis dada
 * Introduce el evento correspondiente en la cola
 */
```

```
void procesar_comando(uint32_t aux) {
    uint8_t comando[3];
    //Variables auxiliares para comando jugada
    int fila;
    int col;
    int checksum;

    comando[0] = aux >> 16;
    comando[1] = (aux & 0xff00) >> 8;
    comando[2] = aux & 0xff;


  if(comando[0] == 'P' && comando[1] == 'A'
      && comando[2] == 'S')
  {
    cola_guardar_eventos(event_comando_pasar,0);
  }
  else if(comando[0] == 'R' && comando[1] == 'S'
           && comando[2] == 'T')
    {
    cola_guardar_eventos(event_comando_acabar,0);
  }
  else if(comando[0] == 'N' && comando[1] == 'E'
           && comando[2] == 'W')
  {
    cola_guardar_eventos(event_comando_nueva,0);
  }
    //comando jugada
  else {
        fila = (comando[0] - '0');
        col = (comando[1] - '0');
        checksum = (comando[2] - '0');
    if( ( ( fila + col ) & 0x7 ) == checksum )
    {
      cola_guardar_eventos(event_comando_jugada,
                           (fila << 8) + col);
    }
  }
}
```

A.2. Módulo de temporizadores

A.2.1. Inicialización de los temporizadores

```
/*
 * Inicializa timer0 y timer1
```

```
 */
void temporizador_iniciar(void)
{
    // Interrumpe cada ms
    T0MR0 = TICKS_MS;
    // Interrumpe y reset a T0TC when MR0 is reached
    T0MCR = 0x3;
    // Enables Timer0 Interrupt
    VICIntEnable = VICIntEnable | 0x00000010;
    // Selects FIQ interrupts
    VICIntSelect = VICIntSelect | 0x00000010;

    // Interrumpe cada periodo * ms =
    //         t1_periodo * 150.000-1 counts / 0.05 ms
    T1MR0 = TICKS_MAX;
    // Generates an interrupt and resets the count when
    //         the value of MR0 is reached and stops the counter
    //         (should act like an alarm)
    T1MCR = 0x3;
    // configuration of the IRQ slot number 0 of the VIC
    //         for Timer 0 Interrupt
    // set interrupt vector in 1
    VICVectAddr1 = (unsigned long)temporizador1_rsi;
    // 0x20 bit 5 enables vectored IRQs.
    // 4 is the number of the interrupt assigned.
    // Number 4 is the Timer 0
    //        (see table 40 of the LPC2105 user manual
    // Timer1 -> Slot 1
    VICVectCntl1 = 0x20 | 5;
    // Enable Timer1 Interrupt
    VICIntEnable = VICIntEnable | 0x00000020;
}
```

A.2.2. Interrupciones timer0

```
/*
 * Guarda evento alarma o alarma_periodica si corresponde
 * para esa interrupción
 */
void temporizador0_rsi (void) __irq
{
        t0_periodo_actual++;
        if(t0_periodo_actual == alarma_periodo_esperado)
        {
            // Interrupcion alarma
            cola_guardar_eventos(event_alarma, t0_periodo_actual);
```

```
            if(interr_set == 0) {
                // Si no hay nada mas que hacer, para el reloj
                T0TCR = 0;
            }
        }
        if(interr_set == 1 &&
            t0_periodo_actual == interr_periodo_esperado)
        {
            // Interrupción periódica
            cola_guardar_eventos(event_alarma_periodica, period);
            interr_periodo_esperado =
                        interr_periodo_esperado + period;
        }
        // Clear interrupt flag
        T0IR = 1;
        // Acknowledge Interrupt
        VICVectAddr = 0;
}
```

A.2.3. Interrupciones timer1

```
/*
 * Aumenta el contador en cada interrupción
 */
void temporizador1_rsi (void) __irq
{
    t1_periodo_count++;
    T1IR = 1;           // Clear interrupt flag
    VICVectAddr = 0;    // Acknowledge Interrupt
}
```

A.2.4. Inicio del timer1

```
/*
 * Inicia la ejecución de timer1 de forma indefinida
 */
void temporizador_empezar(void)
{
    t1_inicializado = 1;
    // Timer0 reset
    T1TCR = 0x3;
    // Reset periodos
    t1_periodo_count = 0;
    // Prescale Register reset
    T1PC = 0;
    // Timer0 enable (T0TC)
```

```
    T1TCR = 0x1;
}
```

A.2.5. Leer tiempo del timer1

```
/*
 * Acceso al timer1 vía una interrupción
 */
uint32_t __swi(0) clock_gettime(void);
uint32_t __SWI_0 (void) {
    return temporizador_leer();
}

/*
 * Devuelve el tiempo (en us) que lleva contando el timer1 desde
 * que se ejecutó temporizador_empezar
 */
uint32_t temporizador_leer(void)
{
    if(T1TCR == 0x1)
    {
        // Si están contando timer1,
        //   devuelve el tiempo transcurrido
        return ((long long)t1_periodo_count *
            ((long long)TICKS_MAX + 1) + T1TC) / TICKS_MS;
    }
    else
    {
        // Sino devuelve 0
        return 0;
    }
}
```

A.2.6. Parada del timer1

```
/*
 * Detiene timer1
 *
 * Devuelve el tiempo transcurrido desde
 *  temporizador_empezar
 */
uint32_t temporizador_parar(void)
{
    uint32_t aux;
    if(T1TCR == 1)
    {
```

```
        // Si está activado timer0
        t1_inicializado = 0;
        aux = temporizador_leer(); // Toma el tiempo
        T1TCR = 0;                 // Disables timer0
        return aux;                // Devuelve t transcurrido
    }
    else
    {
        // Sino devuelve 0
        return 0;
    }
}
```

A.2.7. Programación de alarma

```
/*
 * Genera una notificación (evento en cola de eventos)
 *  dentro de un retardo indicado en milisegundos
 */
void temporizador_alarma(uint32_t retardo)
{
    if(retardo > 0)
    {
        // Hay una alarma establecida
        alarma_set = 1;
        if(T0TCR == 0)
        {
            t0_periodo_actual = 0;
            T0TCR = 3;
            T0PC = 0;
            T0TCR = 1;
            // Timer0 reset and enable
        }
        alarma_periodo_esperado = t0_periodo_actual + retardo;
    }
}
```

A.2.8. Programación de alarma periódica

```
/*
 * Programa el timer0 para que realice una notificación
 *  periódica. El periodo se indica en ms
 */
void temporizador_interrupcion_periodica(uint32_t periodo)
{
    if(periodo > 0)
```

```
    {
        // Hay una alarma establecida
        interr_set = 1;
        period = periodo;
        if(T0TCR == 0)
        {
            t0_periodo_actual = 0;
            T0TCR = 3;
            T0PC = 0;
            T0TCR = 1;
        }
        interr_periodo_esperado = t0_periodo_actual + periodo;
    }
}
```

A.3. Módulo de GPIO

A.3.1. Lectura del GPIO

```
/*
 * bit_inicial indica el primer bit a leer
 * num_bits indica cuántos bits queremos leer
 *
 * Devuelve un entero con el valor de los bits indicado
 */
uint32_t GPIO_leer(uint32_t bit_inicial, uint32_t num_bits)
{
    aux = IOPIN;
    aux = lsl(aux, dim - bit_inicial - num_bits);
    aux = lsr(aux, dim - num_bits);
    return aux;
}
```

A.3.2. Escritura en GPIO

```
/*
 * Similar a GPIO_leer pero escribiendo en los bits
 * indicados el valor
 *
 * Si el valor no puede representarse en los bits indicados
 *  se escribirán los num_bits menos significativos a
 *  partir del inicial
 */
void GPIO_escribir(uint32_t bit_inicial, uint32_t num_bits,
                   uint32_t valor)
{
```

```
    // Bits inicial y final a tratar
    uint32_t ini = lsl(0x1, bit_inicial);

    // contador de bits escritos
    int bits_written = 0;
    uint32_t j;
    for(j = ini; bits_written < num_bits; j = lsl(j,1) )
    {
        if( (j & valor) > 0)
        {
            // Si el bit j de valor es 1, set bit j en GPIO
            IOSET = j;
        }
        else
        {
            // Si el bit j de valor es 0, clear bit j en GPIO
            IOCLR = j;
        }
        bits_written++;

        if(j == 0x8000000)
        {
            break;
        }
    }
}
```

A.3.3. Marcar entrada GPIO

```
/*
 * Los bits indicados se utilizarán como pines de entrada
 */
void GPIO_marcar_entrada(int32_t bit_inicial,
                         int32_t num_bits)
{
    uint32_t i;
    // Bit inicial
    uint32_t ini = lsl(0x1, bit_inicial);
    int bits_set = 0;
    for(i = ini; bits_set < num_bits; i = lsl(i,1) )
    {
        // Bit i = 0
        IODIR = IODIR & (~i);
        bits_set++;

        if(i == 0x8000000) break;
```

```
    }
}
```

A.3.4. Marcar salida GPIO

```
/*
 * Los bits indicados se utilizarán como pines de salida
 */
void GPIO_marcar_salida(int32_t bit_inicial,
                        int32_t num_bits)
{
    // Bit inicial
    uint32_t ini = lsl(0x1, bit_inicial);
    int bits_set = 0;
    uint32_t i;
    for(i = ini; bits_set < num_bits; i = lsl(i,1) )
    {
        // Bit i = 1
        IODIR = IODIR | i;
        bits_set++;

        if(i == 0x80000000) break;
    }
}
```

A.4. Módulo de cola de eventos

A.4.1. Inicialición de la cola

```
/*
 * Inicializa la cola de eventos con un primer evento
 * none que realiza la función de centinela
 */
void inicializarCola()    {
    int i;

    if(temporizador_iniciado() == 0)
    {
        // PANIC, temporizador no iniciado
        while(1);
    }
    // Centinela (Hay un primer evento none)
    eventos.ultimo = 0;
    eventos.primero_no_procesado = 0;

    for(i = 0; i < MAX_SIZE_QUEUE; i++)
```

```
    {
        // Todos los eventos son nulos al inicio
        eventos.eventos[i] = event_none;
        // No hay tiempo inicial
        eventos.tiempos[i] = 0;
    }
}
```

A.4.2. Guardado de eventos

```
/*
 * Guarda un evento en la cola evitando condiciones de carrera
 *
 * Guarda en la cola dos palabras:
 *   ID_evento + auxData (24 bits menos significativos)
 *   Momento exacto que se ha invocado a la función
 */
void cola_guardar_eventos(event_t evento, uint32_t auxData)
{
    int aux;
    // SC
    lock();
    // Se usa una variable auxiliar para control de concurrencia
    aux = (eventos.ultimo + 1) & mod_max;
    eventos.ultimo = aux;
    unlock();

    if (aux != eventos.primero_no_procesado )
    {
        // Caso regular, el evento a sustituir ha sido tratado
        // Bits 31-8 auxData[0,23], 7-0 ID_evento
        eventos.eventos[aux] =
            (to_integer(evento) & 0x000000ff) | (auxData<<8);
        eventos.tiempos[aux] = temporizador_leer();
        lock();
        if(eventos.primero_no_procesado == -1)
        {
            eventos.primero_no_procesado = aux;
        }
        unlock();
    }
    else
    {
        // Trataba de sustituir un evento no tratado, OVERFLOW
        // b. infinito
        while(1);
```

```
    }
}
```

A.4.3. Lectura de un evento

```
/*
 * Lee un evento de la cola evitando condiciones de carrera
 */
event_t leer_evento(uint32_t * auxData, uint32_t * tiempo)
{
    event_t aux;
    if(hay_nuevos_eventos() == 0)
    {
        // Si no hay nuevos eventos
        return event_none;
    }
    else
    {
        // Hay nuevos eventos
        *auxData =
            (eventos.eventos[eventos.primero_no_procesado]
                & 0xffffff00)>>8;
        *tiempo = eventos.tiempos[eventos.primero_no_procesado];
        aux =
            to_event(eventos.eventos[eventos.primero_no_procesado]
                & 0x000000ff);
        lock();
        if(eventos.primero_no_procesado != eventos.ultimo)
        {
            // Mueve primero_no_procesado a la siguiente posicion
            eventos.primero_no_procesado =
                (eventos.primero_no_procesado + 1) & mod_max;
        }
        else
        {
            eventos.primero_no_procesado = -1;
        }
        unlock();
        return aux;
    }
}
```

A.5. Módulo de sección crítica

```
/*
 * Bloquea interrupciones y guarda estado
```

```
 */
void lock()
{
    estado = lock_isr_fiq();
}

/*
 * Recupera el estado anterior
 * Reactiva las interrupcines apropiadas
 */
void unlock()
{
    if(estado == 0)
    {
        // MODO FIQ
        // no se hace nada
    }

    else if(estado == 2)
    {
        // MODO IRQ
        enable_fiq();
    }

    else if(estado == 3)
    {
        // MODO USUARIO
        enable_isr_fiq();
    }
}
```

A.6. Módulo de botones

A.6.1. Inicialización de los botones

```
/*
 * Inicialización de los botones
 */
void eints__init (void)
{
    nueva_pulsacion_eint0 = 0;
    eint0_count = 0;
    // clear interrupt flag
    EXTINT =  EXTINT | 1;
    //IRQ slot number 2 of the VIC for EXTINT0
    VICVectAddr2 = (unsigned long)rsi_eint0;
```

```
    // 0x20 bit 5 enables vectored IRQs.
    // 14 is the number of the interrupt assigned.
    // Number 14 is the EINT0
    // (see table 40 of the LPC2105 user manual)
    // Sets bits 0 and 1 to 0
    PINSEL1         = PINSEL1 & 0xfffffffC;
    // Enable the EXTINT0 interrupt
    PINSEL1         = PINSEL1 | 1;
    VICVectCntl2 = 0x20 | 14;
    // Enable EXTINT0 Interrupt
    VICIntEnable = VICIntEnable | 0x00004000;


    nueva_pulsacion_eint1 = 0;
    // clear interrupt flag
    EXTINT =  EXTINT | 2;
    // IRQ slot number 2 of the VIC for EXTINT0
    // set interrupt vector in 2
    VICVectAddr3 = (unsigned long)rsi_eint1;
    // 0x20 bit 5 enables vectored IRQs.
    // 14 is the number of the interrupt assigned.
    // Number 14 is the EINT0
    // (see table 40 of the LPC2105 user manual)
    // Sets bits 0 and 1 to 0
    PINSEL0         = PINSEL0 & 0xcfffffff;
    // Enable the EXTINT1 interrupt 2_1110
    PINSEL0         = PINSEL0 | 0x20000000;
    VICVectCntl3 = 0x20 | 15;
    // Enable EXTINT1 Interrupt
    VICIntEnable = VICIntEnable | 0x00008000;
}
```

A.6.2. Procesamiento de pulsación

```
/*
 * Devuelve 1 si ha habido nueva pulsación
 */
uint8_t nueva_pulsacion_0()    {
    return nueva_pulsacion_eint0;
}

/*
 * Devuelve 1 si ha habido nueva pulsación
 */
uint8_t nueva_pulsacion_1()    {
    return nueva_pulsacion_eint1;
```

```
}

/*
 * Resetea a cero
 */
void clear_nueva_pulsacion_0()    {
    nueva_pulsacion_eint0 = 0;
}

/*
 * Resetea a cero
 */
void clear_nueva_pulsacion_1()    {
    nueva_pulsacion_eint1 = 0;
}
```

A.6.3. Gestión de la interrupción

```
/*
 * Gestiona la interrupción de EINT0
 * Durante la gestión se dehabilitará la interrupción
 *  externa correspondiente en el VIC, para que no vuelva
 *  a interrumpir hasta que no termine la gestión de la
 *  pulsación
 */
void rsi_eint0(void) __irq
{
    VICIntEnClr = 0x00004000;
    nueva_pulsacion_eint0 = 1;
    cola_guardar_eventos(event_eint0, 0);
    // Clear interrupt flag
    EXTINT =  EXTINT | 1;
    // Acknowledge Interrupt
    VICVectAddr = 0;
}

/*
 * Gestiona la interrupción de EINT1
 * Durante la gestión se dehabilitará la interrupción
 *  externa correspondiente en el VIC, para que no vuelva
 *  a interrumpir hasta que no termine la gestión de la
 *  pulsación
 */
void rsi_eint1(void) __irq
{
    VICIntEnClr = 0x00008000;
```

```
    nueva_pulsacion_eint1 = 1;
    cola_guardar_eventos(event_eint1, 0);
    // Clear interrupt flag
    EXTINT =  EXTINT | 2;
    // Acknowledge Interrupt
    VICVectAddr = 0;
}
```

A.7. Módulo de gestión de pulsación

```
/*
 * Monitoriza la pulsación cada 50 ms.
 * Si la tecla sigue pulsada no se hará nada
 * En caso contario se volverá a habilitar la interrupción
 *  de ese botón.
 */
int gestion_eint0_pulsado(event_t evento)
{
    if(procesando_eint0 == PULSADO && evento == event_alarma)
    {
        if(boton0_pulsado() == 1)
        {
            temporizador_alarma(periodo_espera);
            boton0_clear();
        }
        else
        {
            clear_nueva_pulsacion_0();
            boton0_reactivate();
            // Cambio estado
            procesando_eint0 = NO_PULSADO;
        }
    }
    else if(evento == event_eint0)
    {
        // Cambio estado
        procesando_eint0 = PULSADO;
        temporizador_alarma(periodo_espera);
    }
    return gestionando_eint0();
}

/*
 * Monitoriza la pulsación cada 50 ms.
 * Si la tecla sigue pulsada no se hará nada
 * En caso contario se volverá a habilitar la interrupción
```

```
 *  de ese botón.
 */
int gestion_eint1_pulsado(event_t evento)
{
    if(procesando_eint1 == PULSADO && evento == event_alarma)
    {
        if(boton1_pulsado() == 1)
        {
            temporizador_alarma(periodo_espera);
            boton1_clear();
        }
        else
        {
            clear_nueva_pulsacion_1();
            boton1_reactivate();
            // Cambio estado
            procesando_eint1 = NO_PULSADO;
        }
    }
    else if(evento == event_eint1)
    {
        // Cambio estado
        procesando_eint1 = PULSADO;
        temporizador_alarma(periodo_espera);
    }
    return gestionando_eint1();
}
```

A.8. Módulo de IDLE/POWER-DOWN

```
/*
 * Set the processor into power down state
 * The watchdog cannot wake up the processor from power down
 */
void PM_power_down (void)
{
  // EXTINT0 and EXTINT1 will awake the processor
  EXTWAKE = 0x3;

//    Se fuerza que los botones puedan interrumpir
//    ¡POSIBLE CONDICION DE CARRERA!
//    Se trata de leer pero no hay evento, a la vez que
//    interrumpe una rsi de boton, luego el procesador se
//    duerme pero el evento eint no se ha tratado

    boton0_reactivate();
```

```
    boton1_reactivate();

    // Power-down mode
    PCON |= 0x02;
    Switch_to_PLL();
}

/*
 * Set the processor into idle state
 * Peripherals can wake up the processor with interrupts
 */
void PM_idle (void)  {
    // Idle mode (PC stopped but peripherals remain active)
    PCON |= 0x01;
    Switch_to_PLL();
}
```

A.9. Real Time Clock

```
/*
 * Inicializa el RTC
 * Resetea la cuenta, ajusta el reloj y activa el enable
 */
void rtc_init(void)
{
    CCR = 0x2;        // Reset

    PREINT = 0x01C8;
    PREFRAC = 0x61C0;

    CCR = 0x1;        // Enable
}

/*
 * Devuelve los minutos del juego (entre 0 y 59)
 */
uint8_t rtc_leer_segundos(void)
{
    // Devuelve directamente los segundos del RTC
    return SEC;
}

/*
 * Devuelve los segundos transcurridos en el minuto actual
 * (entre 0 y 59)
 */
```

```
uint8_t rtc_leer_minutos(void)
{
    // Devuelve directamente los minutos del RTC
    return MIN;
}
```

A.10. Watchdog

```
/*
 * Inicializa el watchdog timer para que se resetee el
 *  procesador dentro de sec segundos si no se le
 *  "alimenta"
 */
void WD_init(int sec)    {
    // Asigna el periodo
    WDTC = sec * TICKS_SEG;
    WDMOD = 0x3;
    // Se debe alimentar una primera vez para ponerlo en
    //  marcha
    WD_feed();
}

/*
 * Alimenta al watchdog timer
 */
void WD_feed(void)    {
    // Desactivar interrupciones
    disable_isr_fiq();

    WDFEED = 0xAA;
    WDFEED = 0x55;

    enable_isr_fiq();
}
```

A.11. Interrupciones software

```
__enable_isr
                ; Load R8, SPSR
                LDMFD   SP!, {R8, R12}
                BIC        R12, R12, #I_Bit
                ; Set SPSR
                MSR     SPSR_cxsf, R12
                ; Restore R12 and Return
                LDMFD   SP!, {R12, PC}^
```

```
__disable_isr
                ; Load R8, SPSR
                LDMFD   SP!, {R8, R12}
                ORR       R12, R12, #I_Bit
                ; Set SPSR
                MSR     SPSR_cxsf, R12
                ; Restore R12 and Return
                LDMFD   SP!, {R12, PC}^


__enable_isr_fiq
                ; Load R8, SPSR
                LDMFD   SP!, {R8, R12}
                BIC       R12, R12, #I_Bit
                BIC       R12, R12, #F_Bit
                ; Set SPSR
                MSR     SPSR_cxsf, R12
                ; Restore R12 and Return
                LDMFD   SP!, {R12, PC}^


__disable_isr_fiq
                ; Load R8, SPSR
                LDMFD   SP!, {R8, R12}
                ORR       R12, R12, #I_Bit
                ORR       R12, R12, #F_Bit
                ; Set SPSR
                MSR     SPSR_cxsf, R12
                ; Restore R12 and Return
                LDMFD   SP!, {R12, PC}^

; Deshabilita las interrupciones, tanto IRQ como FIQ y
;  devuelve en r0 el modo desde el que se llamo
;  (0 - Usuario, 1 - Irq, 3 - Fiq)
__lock_isr_fiq
                ; Load R8, SPSR
                LDMFD   SP!, {R8, R12}

                TST       R12, #I_Bit
                MOVEQ    R0, #0x1
                MOVNE    R0, #0x0
                ORR       R12, R12, #I_Bit

                TST       R12, #F_Bit
                ORREQ    R0, #0x2
                ORR       R12, R12, #F_Bit
```

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

__enable_fiq
                ; Load R8, SPSR
                LDMFD   SP!, {R8, R12}
                BIC        R12, R12, #F_Bit
                ; Set SPSR
                MSR     SPSR_cxsf, R12
                ; Restore R12 and Return
                LDMFD   SP!, {R12, PC}^
```

A.12. Módulo de línea de serie

A.12.1. Inicialización de la UART0

```
/*
 * Initialize Serial Interface
 */
void init_serial (void)
{
  PINSEL0 &= 0x0;
  PINSEL0 |= 0x5; /* Enable RxD0 and TxD0              */
  U0LCR = 0x83;   /* 8 bits, no Parity, 1 Stop bit     */
  U0DLL = 10;     /* 9600 Baud Rate @ 15MHz VPB Clock  */
  U0LCR = 0x03;   /* DLAB = 0                          */

  // RBR and THRE interrupts enabled
  U0IER |= (1<<0)|(1<<1);
  // set interrupt vector in 2
  VICVectAddr4 = (unsigned long)rsi_uart0;

  // 0x20 bit 5 enables vectored IRQs.
  // 6 is the number of the interrupt assigned.
  // Number 6 is the UARTT0
  VICVectCntl4 = 0x20 | 6;
  // Enable UART0 Interrupt
  VICIntEnable = VICIntEnable | 0x00000040;

  leer = 0;
  escribir = 0;
}
```

A.12.2. Interrupciones RDA y THRE

```
/*
 * Si interrupción es RDA:
 *   Se muestra el caracter y se guarda como evento
 * Si interrupción es THRE:
 *   Se muestra uno de los caracteres pendientes del buffer
 */
void rsi_uart0(void) __irq
{
    int leido;
    if (U0IIR & (1<<2))
    {
        // If the RDA interrupt is triggered
        if(U0LSR & 0x01)
        {
            leido = U0RBR;
            sendchar(leido);
            cola_guardar_eventos(event_char_leido, leido);
        }
    }
    else
    {
        // THRE Interrupt
        leido = readFromBuffer();
        if(leido != -1)
        {
            sendchar(leido);
        }
    }

    VICVectAddr = 0;
}
```

A.12.3. Envío de cadenas de caracteres

```
/*
 * Guarda una cadena en el buffer circular
 * Será mostrada mediante interrupciones THRE
 *   o USLR & 0x20
 */
void sendstring(char cadena[])
{
    int i = 0;
    int leido = 0;
```

```
    while(cadena[i] != '\0')
    {
        writeOnBuffer(cadena[i]);
        i++;
    }

    if(U0LSR & 0x20)
    {
        leido = readFromBuffer();
        if(leido != -1)
        {
            sendchar(leido);
        }
    }
}
```

A.13. Módulo de intérprete

```
/*
 * Procesa un caracter recibido siguiendo el patrón #xxx!
 */
void procesar_caracter(uint8_t character)
{
    if(character == '#')
    {
        reading = 1;
        index = 0;
    }
    else if(index == 3)
    {
        if(character == '!' && reading == 1)
        {
            guardar_comando();
            sendchar('\n');
        }
        reading = 0;
        index = 0;
    }
    else if(reading == 1)
    {
        guardar_caracter(character);
    }
}

/*
 * Guarda un caracter en el buffer
```

```
 */
void guardar_caracter(uint8_t caracter)
{
  if(index < MAX_BUFFER_SIZE)
    {
    buffer[index] = caracter;
    index++;
  }
  else
    {
    // ERROR: Buffer overflow
    while(1);
  }
}

/*
 * Guarda un comando completo como evento
 */
void guardar_comando()
{
    uint32_t comando = 0;
    comando += buffer[2];        // 0 1 2
    comando += buffer[1] << 8;
    comando += buffer[0] << 16;
    cola_guardar_eventos(event_comando_reconocido, comando);
}
```