

Proyecto Hardware

Prácticas 2 y 3

Implementación del juego «Conecta 4» para un microcontrolador NXP LPC 2105 con un sistema entrada/salida para la interacción y planificación basada en eventos

Dorian Boleslaw Wozniak (817570@unizar.es)

Pablo Latre Villacampa (778043@unizar.es)

En Zaragoza, a 10 de enero de 2023

Índice

Índice	2
Resumen ejecutivo	4
1. Introducción	5
2. Desarrollo de la práctica 2	7
2.1: Creación del planificador	7
2.1.1: Cola de eventos	7
2.1.2: Cola de mensajes	8
2.1.3: Bucle de ejecución del planificador	9
2.2: Periféricos para la gestión de tiempo	10
2.2.1: timer0	10
2.2.2: timer1	12
2.3: Gestor de alarmas	14
2.4: Periféricos y gestión de entrada y salida	15
2.4.1: GPIO	15
2.4.2: Botones	16
2.5: Gestión de energía	18
2.6: Integración del proyecto	19
3. Desarrollo de la práctica 3	22
3.1: Correcciones a la práctica 2	22
3.2: Implementación de los modos FIQ y SWI	22
3.3: Inclusión del Real-time Clock y del Watchdog	24
3.4: Comunicación por línea serie	25
3.4.1: UART0	25
3.4.2: Cola búfer	26
3.4.3: Implementación de la comunicación	26
3.5: Integración del proyecto definitivo	28
4. Conclusiones	32
4.1: Comentarios: Dorian Wozniak	32
4.2: Comentarios: Pablo Latre	33
Anexo A: Código fuente del proyecto	34
Anexo A.1: Main	34
Anexo A.1.1: main.c	34
Anexo A.2: Conecta 4	38
Anexo A.2.1: conecta4_2022.c	38
Anexo A.3: Colas	41
Anexo A.3.1: cola_eventos.c	41

Anexo A.3.2: eventos.h	43
Anexo A.3.3: cola_msg.c	43
Anexo A.3.4: msg.h	44
Anexo A.3.5: cola_buffer.c	45
Anexo A.4: Periféricos	46
Anexo A.4.1: tiempo.c	46
Anexo A.4.2: gpio.c	50
Anexo A.4.3: power.c	52
Anexo A.4.4: boton.c	52
Anexo A.4.5: uart.c	55
Anexo A.4.6: irq_control.h	57
Anexo A.4.7: pll.s	57
Anexo A.5: Gestores	59
Anexo A.5.1: G_Alarm.c	59
Anexo A.5.2: G_Energia.c	62
Anexo A.5.3: G_IO.c	62
Anexo A.5.4: G_Boton.c	64
Anexo A.5.5: G_Serie.c	65
Anexo A.5.6: alarmas.h	68
Anexo B: Ficheros adjuntos	69

Resumen ejecutivo

Se ha realizado una implementación de un juego Conecta 4 para un microcontrolador NXP LPC2105 con interacción con el usuario mediante periféricos, siguiendo un patrón de diseño modular para la gestión de periféricos y acciones de alto nivel, con un planificador basado en eventos para tratar las acciones síncronas y asíncronas (generadas por periféricos) para implementar el programa.

En la práctica 2, se ha implementado el planificador, con dos colas para eventos síncronos y asíncronos, y un sistema de alarmas programables con un control de tiempo mediante eventos periódicos producidos por un temporizador programable por el usuario. Se ha utilizado el otro temporizador programable disponible para permitir la medición del tiempo con precisión de microsegundos.

Para la interacción con el usuario, se han implementado como botones dos de las tres interrupciones externas disponibles con un tratamiento de las pulsaciones para evitar que se generen múltiples eventos con una sola pulsación. Para comunicar el estado de la partida y la columna donde introducir una ficha, se ha configurado el periférico GPIO con las funciones de alto nivel apropiadas para interactuar con el usuario.

Basándose en la implementación de la práctica 1, se ha modificado las funciones disponibles en el módulo del juego para permitir comunicar las acciones a realizar mediante el planificador. El sistema diseñado no tiene esperas activas y se han las interrupciones generadas a las estrictamente necesarias. Además, se realiza una gestión energética del microprocesador, permitiendo cambiar su estado para suspender la ejecución si no hay eventos a tratar, y dormir el procesador tras un tiempo sin interacción.

Para la práctica 3, se han corregido errores detectados en la práctica anterior. Además, se han realizado una serie de tareas adicionales: se ha modificado el temporizador periódico para que genere interrupciones rápidas; se han implementado otros dos temporizadores, uno en tiempo real para medir el tiempo en segundos, y un *watchdog* para reiniciar el procesador en caso de fallo; y se han implementado llamadas al sistema para permitir desactivar interrupciones para garantizar la exclusión mútua, y para obtener el tiempo transcurrido.

La entrada/salida se ha modificado con el uso de una línea serie. Se ha implementado el tratamiento de lecturas de la línea, que acepta una serie de comandos predefinidos, y de la escritura, que se realiza de forma asíncrona y se utiliza para comunicar al usuario las instrucciones del juego, el estado del tablero durante el juego, y la causa del fin de juego estadísticas de rendimiento al acabar.

Para finalizar, se ha modificado la implementación anterior para implementar las modificaciones anteriores. El juego permite iniciar la partida, introducir jugadas nuevas, cancelarlas en un tiempo de gracia, y reiniciar la partida, en todo momento manteniendo la funcionalidad anterior y el correcto funcionamiento del sistema. El resultado final es perfectamente funcional, aunque se han detectado una serie de errores no críticos y subsanables.

1. Introducción

En la primera práctica de esta asignatura, se ha realizado un trabajo de análisis de rendimiento y optimización sobre una implementación de un juego Conecta 4 desarrollado en C y ensamblador ARM para un microcontrolador NXP LPC 2105.

En esta práctica, se parte de dicho proyecto para implementar la comunicación entre el juego y el usuario mediante periféricos, abandonando el método de manipulación de memoria usado anteriormente para comunicar las intenciones del jugador al programa, que además requiere del uso de las herramientas de depuración para realizarlo. Esta comunicación se ha realiza mediante periféricos de entrada/salida de propósito general (*General-purpose input/output*, GPIO), una línea de comunicación en serie (mediante el *Universal asynchronous receiver-transmitter*, UART), y fuentes de interrupción externas para transmitir el estado del juego, así como recibir nuevas órdenes.

Además, se requerirá de herramientas para la medición del tiempo transcurrido, así como realización de tareas periódicas, y una gestión de los estados de energía del procesador y del propio programa. Para poder integrar toda esta funcionalidad, incluido el propio juego, se ha realizado un diseño del sistema modular, donde cada parte significativa del sistema forma un componente independiente con una interfaz bien definida, reutilizable y fácil de modificar sin realizar cambios extensos a otros módulos del programa que lo utilizan.

Finalmente, para atar todas las componentes, se implementará el sistema sobre un planificador basado en eventos, que permitirá una comunicación entre estos componentes del programa tanto para eventos asíncronos, generados principalmente por los periféricos, así como eventos síncronos de los módulos de más alto nivel.

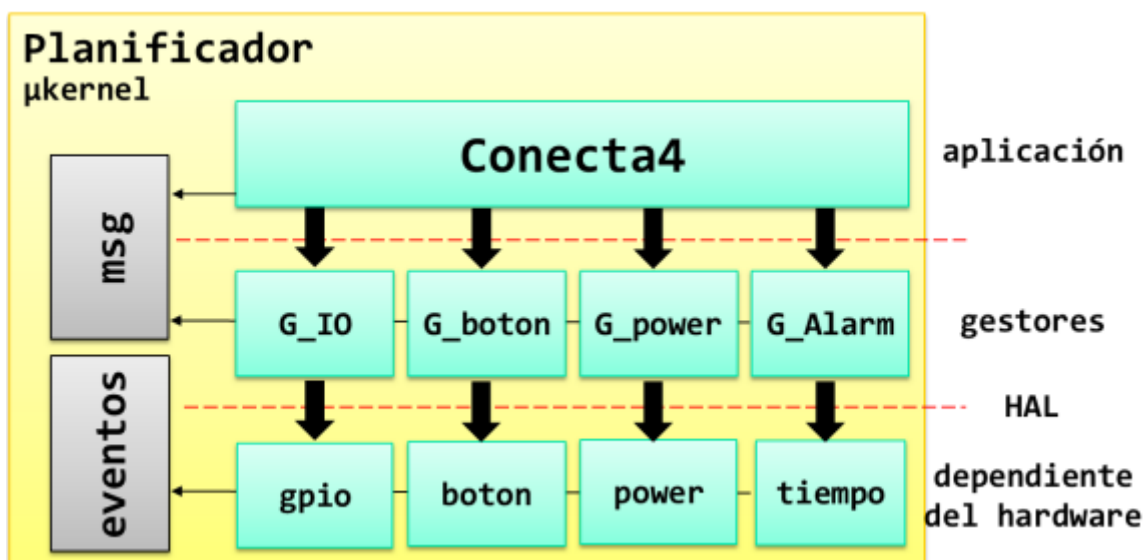


Figura 1: Estructura del planificador

Para el diseño de dicho planificador, se puede definir una estructura de tres capas:

- **Capa de periféricos:** Implementan la funcionalidad y comportamiento a bajo nivel de los periféricos necesarios. Los periféricos, al ocurrir un evento asíncrono relevante a este (en este caso, siempre será como consecuencia de una interrupción) enviarán la información al planificador (de ahora en adelante, todo evento asíncrono será llamado «evento»).
- **Capa de gestores:** Implementan funcionalidades a alto nivel a partir de la interfaz de los periféricos descrita anteriormente. Así, se abstrae el funcionamiento de los periféricos de las tareas que pueden realizar. Si un gestor necesita comunicar una acción síncrona que deberá realizar otro módulo o la propia aplicación, lo realizará enviando la información al planificador (de ahora en adelante, todo evento síncrono será llamado «mensaje»).
- **Capa de aplicación:** Esta capa está conformada por la aplicación a desarrollar, es decir, el juego Conecta 4, y el planificador. El segundo gestiona el comportamiento descrito anteriormente: al recibir un evento o un mensaje, invoca la función o funciones adecuadas de los gestores o la aplicación disponibles. Por otro lado, la aplicación también puede acceder a la funcionalidad de los gestores, pero es el planificador quien invoca primero a la aplicación.

2. Desarrollo de la práctica 2

2.1: Creación del planificador

Para poder desarrollar el resto del programa, primero se necesita definir la estructura del planificador: cómo obtendrá y almacenará los eventos y mensajes que recibirá a medida que se ejecuta el programa, así como el tratamiento posterior de estos eventos y mensajes.

2.1.1: Cola de eventos

En primer lugar, se ha definido el formato que sigue un evento así como el método de almacenamiento de dichos eventos.

Un evento está formado por un identificador único al tipo de evento (1 byte) y un campo para datos adicionales que el evento pudiera incluir (4 bytes). Para ello, se ha decidido crear un tipo de datos abstracto para representar a dicho evento:

```
// Definición de un evento
typedef struct EventoHW {
    uint8_t    ID_evento;    // Identificador de evento
    uint32_t    auxData;    // Datos auxiliares del evento
} evento ;
```

El ID de cada tipo de evento se encuentra definido en el fichero *eventos.h*

Dichos eventos serán almacenados en otro tipo de datos abstracto: una cola circular de tamaño estático. Dicha cola puede almacenar hasta 32 elementos de forma simultánea antes de desbordar.

Para la cola de eventos, además del propio evento se almacena la cuenta de veces que ha ocurrido el evento en particular (4 bytes).

```
// Definición de un elemento de la cola
typedef struct ElementoCola {
    evento      unEvento;    // Evento almacenado
    uint32_t    veces;    // Número de evento
} elemento ;
```

La cola, al ser circular, requiere de dos variables para conocer el índice del primer y último elemento insertados. La cola ofrece la siguiente interfaz (definida en *cola_eventos.h*) para meter elementos en al cola y luego sacarlos:

```

// Encola un evento en la cola.
void cola_encolar_eventos(uint8_t ID_evento, uint32_t veces, uint32_t auxData);

// Devuelve si la cola está vacía.
uint8_t cola_hay_eventos();

// Devuelve el siguiente evento de la cola.
void cola_obtener_sig_evento(evento *e);

```

Cabe destacar tres cosas sobre la cola:

- La cola se puede desbordar al introducir más de 32 elementos. En dicho caso, el programa detendrá su ejecución (en este caso, se quedará en un `while(1)` al no haber sistema operativo que pueda finalizar la tarea).
- Para obtener el siguiente evento, se debe comprobar primero si hay eventos almacenados o si la cola está vacía utilizando la función adecuada. Intentar desencolar un evento sin comprobar primero si se puede puede resultar en comportamiento indefinido (pues podría leer un evento antiguo).
- Los eventos obtenidos se almacenan en un *struct* pasado por referencia. Se ha decidido que un evento forme parte de la interfaz de *eventos.h* para evitar que los datos que forman un elemento no se encuentren separados.

2.1.2: Cola de mensajes

La implementación de los mensajes y la cola es similar a la implementación de la cola de eventos con varias particularidades descritas a continuación.

Las colas de mensajes forman un tipo de dato similar a los eventos, donde almacenan un identificador único al tipo de mensaje (1 byte), así como el contenido de dicho mensaje (4 bytes).

```

// Definición de un evento
typedef struct MensajeG {
    uint8_t ID_msg;      // Identificador de mensaje
    uint32_t mensaje;    // Contenido
} msg ;

```

Los identificadores del tipo de mensaje quedan definidos en *msg.h*, así como el TAD anterior.

Los elementos de la cola de mensajes, sin embargo, se diferencian en que en este caso, en vez de almacenar el número de ocurrencias, almacena el instante de tiempo en el que se ha generado el mensaje en microsegundos (4 bytes).


```
// Definición de un elemento de la cola
typedef struct ElementoCola {
    msg         unMsg;           // Mensaje almacenado
    uint32_t     tiempo;         // Instante al encolar (us)
} elemento ;
```

La interfaz de la cola de mensajes, definida en *cola_msg.h*, es similar y funciona de forma análoga a la de la cola de mensajes:

```
// Encola un mensaje en la cola.
void cola_msg(uint8_t ID_msg, uint32_t mensaje);

// Devuelve si la cola está vacía.
uint8_t cola_hay_msg(void);

// Devuelve el siguiente mensaje de la cola.
void cola_obtener_sig_msg(msg *m);
```

2.1.3: Bucle de ejecución del planificador

Finalmente, dadas las dos colas descritas anteriormente junto a la definición de un evento y un mensaje, se ha creado la estructura de la ejecución del planificador. El planificador, en cada iteración, intentará obtener primero un evento de la cola de eventos. Si lo logra, realizará la acción asociada a este evento. En caso contrario, comprueba si puede obtener un mensaje de la cola de mensajes. Si tampoco logra obtener un mensaje nuevo a procesar, el procesador entrará en estado de reposo, cuyo funcionamiento se describe más adelante.

```
// Structs para almacenar lo descolado
evento e = {0, 0};
msg m = {0, 0};

// Bucle del planificador
while (1) {
    // Mira si hay eventos pendientes
    if (cola_hay_eventos()) {
        cola_obtener_sig_evento(&e);
        tratar_evento(&e);
    }

    // Mira si hay mensajes pendientes
    else if (cola_hay_msg()) {
        cola_obtener_sig_msg(&m);
        tratar_mensaje(&m);
    }

    // Si no hay nada que hacer, entra en idle
    else { energia_reposo(); }
}
```

Al ser los eventos y mensajes abundantes, se ha decidido separar el procesamiento en dos funciones separadas que seleccionan la acción adecuada según el ID del mensaje o evento:

```
/* Trata el evento recibido */
void tratar_evento(evento* e) {
    switch (e -> ID_evento) {
        case ETIQ_EVENTO_1:
            /* Realiza una acción */
            break;
        ...
    }
}

/* Trata el mensaje recibido */
void tratar_mensaje (msg* m) {
    switch (m -> ID_msg) {
        case ETIQ_MSG_1:
            /* Realiza una acción */
            break;
        ...
    }
}
```

2.2: Periféricos para la gestión de tiempo

Los primeros periféricos cuyo código se ha desarrollado del proyecto han sido los relacionados al control del tiempo. El microcontrolador LPC 2105 dispone de dos temporizadores de propósito general: *timer0* y *timer1*. El comportamiento a definir para estos periféricos es el siguiente:

- ***timer0*** será utilizado para realizar **interrupciones con una frecuencia periódica**, indicada en milisegundos.
- ***timer1*** será utilizado para realizar **mediciones de tiempo**. Deberá ser capaz de devolver instantes de tiempo con precisión de microsegundos.

Además, ambos temporizadores deberán interrumpir la ejecución del procesador el menor número de veces posible, es decir, evitando interrupciones superfluas y sólo cuando sea relevante.

La interfaz de ambos temporizadores se encuentra disponible en *tiempo.h*

2.2.1: *timer0*

Para configurar el temporizador periódico, interesa que pueda medir el tiempo que transcurre en milisegundos tal que, pasado el tiempo indicado, produzca una interrupción y, consecuentemente, un evento para el planificador.

En cuanto a la interfaz, *timer0* solo ofrecerá una función para inicializarlo:

```
// Inicia el periférico Timer0 con el periodo dado en milisegundos.  
void temporizador_reloj(uint32_t periodo);
```

Al invocar la función, se realizarán una serie de pasos:

Primero se debe configurar el **registro de preescalado** (*prescale register*, TOPR). El reloj, cada ciclo del reloj de periféricos (*peripheral devices clock*, PCLK), incrementará en uno el valor de su contador de preescalado (*prescale counter*, TOPC). Cada vez que el PC alcance el valor del PR, incrementará en uno el valor del registro de control de tiempo (*timer control register*, TOTCR), lo que se denominará un *tick*. Se desea que cada *tick* ocurra cada 1 milisegundo.

Para ello, se requiere calcular cuántos ciclos del PCLK ocurren en un milisegundo. Tal como se encuentra configurado el proyecto de *Keil*, el reloj de los periféricos es un cuarto de la frecuencia del reloj del procesador, que a su vez es cinco veces la velocidad del reloj generado por un oscilador de cristal. Si el oscilador tiene una frecuencia de 12 megahercios, el procesador tiene una frecuencia de 60 MHz, y los periféricos una frecuencia de 15 MHz.

Por tanto, la fórmula para obtener el valor del PR es

$$PR = \frac{T(\text{segundos})}{T_{PCLK}} = f_{PCLK} * T(\text{segundos})$$

Para $1\text{ ms} = 10^{-3}\text{ s}$ y una frecuencia de $15\text{ MHz} = 15 * 10^6\text{ Hz}$, el resultado es de 15000 ciclos por cada milisegundo, al que se le resta 1 por razones del simulador de *Keil*.

Una vez definido el periodo de cada *tick*, se debe configurar el resto del periférico. Es necesario configurar alguno de los **registros de comparación** del temporizador (*match register*, TOMR) para que interrumpa con el periodo dado y reinicie la cuenta de *ticks*. En este caso, se carga en el MR0 el periodo dado por parámetro (menos 1), y en el registro de control asociado (*match control register*, TOMCR) se activan los bits correspondientes para el MR0 (bit 0 para interrumpir al alcanzar valor y bit 1 para reiniciar cuenta).

También es necesario configurar el **registro de control del temporizador** (*timer control register*, TOTCR) para reiniciar el temporizador (bit 1) y activar las interrupciones (bit 0). Es importante poner valor del bit 1 a 0 para que no se reinicie constantemente el temporizador.

Finalmente, se debe asignar la rutina de servicio asociada al *timer0* al **controlador vectorizado de interrupciones** (*Vectored Interrupt Controller*, VIC). Esta interrupción quedará asociada al puesto 0 del VIC (el más prioritario). Para ello, se carga en el registro VICVectAddr0 la dirección en memoria de la rutina *timer0_RSI*, y en su registro de control asociado, VICVectCntl0, se asocian las interrupciones generadas por *timer0* a la posición 0 (*timer0* es la interrupción con valor 4), y se activa su tratamiento (bit 5).

```

// Inicia el periférico Timer0 con el periodo dado.
void temporizador_reloj(uint32_t periodo) {
    // Ajusta la cuenta para contar ms e interrumpir cuando alcance periodo
    T0PR = 15000 - 1;
    T0MR0 = periodo - 1;
    T0MCR |= 0x1 | 0x2;

    // Reinicia TC
    T0TCR |= 0x1 | 0x2;
    T0TCR &= ~0x2;

    // Configura VIC[1] para tratar timer1
    VICVectAddr0 = (uint32_t) timer0_RSI;
    VICVectCntl0 |= 0x20 | 4;
}

```

En cuanto a la rutina de servicio, su comportamiento es muy sencillo: actualiza su cuenta de interrupciones, encola un evento nuevo de tipo *T0_PERIODO* sin datos y baja la interrupción.

```

// Rutina de servicio para Timer0.
void timer1_RSI (void) __irq {
    static volatile uint32_t timer0_count = 0;

    // Trata la interrupción
    timer0_count++;
    cola_encolar_eventos(T0_PERIODO, timer0_count, 0);

    // Baja la interrupción
    T0IR = 1;
    VICVectAddr = 0;
}

```

Cabe destacar el símbolo *__irq* incluido en la declaración para indicar al compilador que este código se debe ejecutar en modo IRQ del procesador.

2.2.2: *timer1*

Por otro lado, se desea configurar el *timer1* como un contador con precisión de microsegundos. No es necesario que interrumpa la ejecución del procesador porque se puede obtener el valor de su cuenta leyendo del registro TCR descrito anteriormente. Sin embargo, hay un caso en el que si es necesario una interrupción: si se desborda el TCR, lo cual ocurrirá cada 4294 segundos, es decir, cada 71 minutos .

Para tratarlo, es necesario que se interrumpa cada vez que este registro de 4 bytes llega a su valor máximo (0xffffffff - 1). Así, con la cuenta de interrupciones, se puede determinar cuanto tiempo hay que añadir adicionalmente al valor devuelto al leer el TCR. En caso de devolver el tiempo como un entero de 32 bits no es relevante, pero si se quisiera actualizar a devolver enteros de 64 bits, este comportamiento solventará el problema.

```

// Lee el instante de tiempo del Timer1
uint32_t temporizador_leer() {
    // Lee tiempo actual
    uint32_t time = T1TC;

    // Añade las veces que se ha desbordado (cada ~71 min)
    for (int i = timer1_count; i > 0; i--) time += 0xffffffff;

    return time;
}

```

En cuanto a la RSI, sólo se encola un evento por razones de depuración.

El temporizador ofrece una interfaz más amplia:

```

// Inicia el periférico Timer1.
void temporizador_iniciar(void);

// Inicia a contar el tiempo mediante timer1.
void temporizador_empezar(void);

// Lee el instante de tiempo del Timer1
uint32_t temporizador_leer(void);

//Detiene el contador Timer1 junto al instante tiempo al detenerse
uint32_t temporizador_parar(void);

```

La configuración sigue una estructura similar al del *timer0*. Las diferencias están en el valor del PR (siguiendo la fórmula, $15 - 1$ ciclos para $1\ \mu\text{s}$), del MRO ($0\text{xffffffff} - 1$, el entero de 32 bits más grande), sustituyendo los registros de T0 por T1, y asociando las interrupciones al VIC 1, siendo el valor de la interrupción del *timer1* el número 5. Además, las interrupciones no se habilitan hasta cuando se decide empezar a contar.

2.3: Gestor de alarmas

Contando con el comportamiento del *timer0*, se ha desarrollado un gestor de alarmas que permitirá programar el envío de mensajes tras un tiempo determinado en milisegundos.

El módulo de alarmas podrá almacenar hasta 8 alarmas en un vector. Una alarma se define como:

```
// Tipo alarma
typedef struct AlarmaGestor {
    uint8_t    mensaje;    // Mensaje programado
    uint8_t    periodico;  // Alarma periódica
    uint32_t    retardo;    // Retardo programado (ms)
    uint32_t    restante;  // Tiempo restante (ms)
} alarma ;
```

Este gestor ofrecerá el la siguiente interfaz:

```
// Programa, reprograma o cancela una alarma dado los contenidos del mensaje dado
void alarma_programar(uint32_t msg);

// Actualiza todas las alarmas activas reduciendo su contador con el periodo dado en
// milisegundos. Si alguna alarma vence, encola el mensaje apropiado
void alarma_refrescar(uint32_t periodo);
```

Si un módulo o la aplicación quisieran programar una alarma, deberán encolar un mensaje SET_ALARMAS con un mensaje adjunto con el siguiente formato:

- **Bits 31-24:** Tipo de mensaje (tal como está definido en *msg.h*)
- **Bit 23:** Determina si es una alarma periódica (si lo es, se reinicia tras vencer, si no se elimina)
- **Bits 22-0:** Retardo en milisegundos

Al llegar el mensaje SET_ALARMAS al planificador, invocará la primera función, la cual tiene el siguiente comportamiento:

1. Obtiene la información del mensaje y trata de encontrar una alarma cuyo campo «mensaje» sea igual al obtenido.
2. Si se encuentra, y el retardo es distinto de cero, se actualiza la alarma con los datos obtenidos y se reinicia la cuenta atrás; en caso contrario, se elimina.
3. Si no se encuentra, se añade en el siguiente hueco libre y se inicia la cuenta atrás.

Cada vez que llega un evento TO_PERIODO, queda invocada la segunda función. Esta reduce el campo «restante» de cada alarma programada con el periodo dado. Si el resultado fuera igual o menor a cero, encola el mensaje programado y elimina o reinicia la alarma según si es periódica o no.

2.4: Periféricos y gestión de entrada y salida

La interacción entre el usuario y el programa se desarrolla principalmente mediante el GPIO, que servirá para mostrar y obtener información sobre el estado de la partida, y periféricos externos, que actúan como botones. Las RSI se vectorizan de forma similar a los temporizadores.

2.4.1: GPIO

Los pines del GPIO se utilizan principalmente como «*leds*» para mostrar información de la partida, así como permitir la entrada de información por el usuario. Estos pines se pueden marcar como de entrada (el usuario puede modificar el estado del pin, pero no el programa) o de salida (viceversa), utilizando el registro IODIR. El valor de los pines se puede obtener leyendo del registro IOPIN, y se puede escribir en ellos limpiando el valor actual y a continuación escribiendo el valor deseado mediante los registros IOCLR y IOSET, respectivamente.

La interfaz que ofrece el módulo *gpio.h* es:

```
// Inicializa todos los pines de GPIO
void GPIO_iniciar(void);

// Lee una serie de pines de GPIO.
uint32_t GPIO_leer(uint8_t bit_inicial, uint8_t num_bits);

// Escribe en una serie de pines de GPIO.
void GPIO_escribir(uint8_t bit_inicial, uint8_t num_bits, uint32_t valor);

// Marca una serie de pines de GPIO como entrada.
void GPIO_marcar_entrada(uint8_t bit_inicial, uint8_t num_bits);

// Marca una serie de pines de GPIO como salida.
void GPIO_marcar_salida(uint8_t bit_inicial, uint8_t num_bits);
```

bit_inicial y *num_bits* determinan el rango de pines sobre los que realizar la operación. Para facilitar las operaciones de manipulación de bits, se ha desarrollado una función para generar una máscara de bits a partir de los parámetros dados:

```
// Devuelve una máscara para leer/escribir en GPIO.
uint32_t GPIO_mascara(uint8_t bit_inicial, uint8_t num_bits) {
    uint32_t mask = 0;

    for (uint8_t i = num_bits; i > 0; i--) { mask <= 1; mask |= 0x1; }

    return mask << bit_inicial;
}
```

La implementación de las funciones del GPIO de alto nivel se desarrollará más adelante.

2.4.2: Botones

La implementación de los botones se realiza mediante las interrupciones externas 1 y 2 (EINT). Las interrupciones externas sustituyen los pines 14 y 15 del bloque de conectores del procesador, sustituyendo los respectivos pines del GPIO (modificando los registros PINSEL0 y PINSEL1).

Los botones presentan un problema al implementarlos: si se mantiene pulsado un botón, este seguirá realizando interrupciones mientras siga pulsado. Esto provoca que una simple pulsación genere cientos de interrupciones que, en este caso, son redundantes y problemáticas.

Para ello, el módulo gestor de los botones se encargará de decidir cuándo tratar y cuando ignorar las interrupciones generadas por EINT. Para ello, utiliza un autómata con el siguiente comportamiento:

- Si el botón se pulsa, generando un evento *BOTON1*, las interrupciones quedarán desactivadas para este botón y se iniciará una alarma periódica, *CHK_BOTON*.
- Cada vez que se trate un *CHK_BOTON*, si el botón sigue pulsado, se ignorará. Si por el contrario el botón no está pulsado, se cancela la alarma y se reactivan las interrupciones.

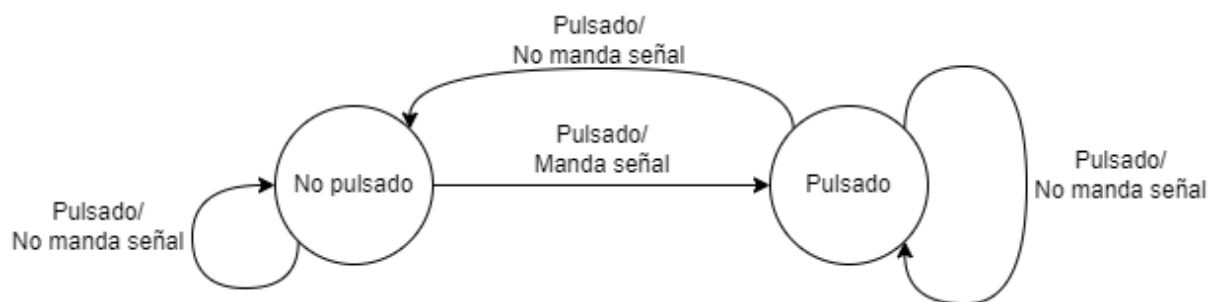


Figura 2: Máquina de mealy que representativa el funcionamiento de los botones

Por un lado, el módulo *boton.h* ofrece la siguiente interfaz:

```
// Inicializa la interrupción externa EINT1
void eint1_iniciar(void);

// Habilita EINT1 para poder interrumpir
void eint1_habilitar(void);

// Obtiene el estado de la interrupción EINT1
uint32_t eint1_leer(void);

// Las funciones son análogas para EINT2
```

En la rutina de servicio para los botones, hay una particularidad importante: no se reactivan las interrupciones directamente, sino que es necesario llamar a la función *eint_habilitar* desde la aplicación.

Por otro lado, el módulo G_Boton.h ofrece la siguiente interfaz:

```
// Empieza a comprobar si se ha pulsado el botón 1
void boton1_pulsado(void);

// Si el botón ya no está pulsado, deja de comprobar si el botón está pulsado
void boton1_comprobar(void);

// Análogo para boton2
```

Otra particularidad es cómo determina si el botón continúa pulsado: se baja primero la interrupción si el botón estuviera pidiendo una interrupción, y si al bajarla se lee que se sigue pidiendo una interrupción, entonces el botón sigue pulsado:

```
// Obtiene el estado de la interrupción EINT1
uint32_t eint1_leer() {
    EXTINT |= EXTINT_1;
    VICVectAddr = 0;
    return ((EXTINT & EXTINT_1) >> 1);
}
```

2.5: Gestión de energía

El procesador ofrece tres modos de energía:

- **Despierto:** El procesador está procesando nuevas instrucciones a la frecuencia de reloj establecida.
- **En reposo:** El procesador deja de procesar nuevas instrucciones, pero el reloj de periféricos sigue activo y el procesador se despierta al solicitarse una interrupción.
- **Dormido:** La ejecución se suspende completamente. Únicamente responde a interrupciones externas, que se deben establecer con anterioridad escribiendo en el registro EXTWAKE.

power.h dispone de las funciones *power_idle* y *power_down* para gestionar los estados de energía, escribiendo el valor adecuado en el registro PCON. Así mismo, se utiliza el gestor *G_Energia.h* como interfaz para acceder a dichas funciones respectivamente mediante *energia_reposo* y *energia_dormir*.

Un último problema a resolver es reactivar el circuito multiplicador de frecuencia (*Phase Locked Loop*, PLL), pues este queda deshabilitado al dormir el procesador. El código ensamblador para activarlo se encuentra en *pll.s* como la rutina *reiniciar_pll*, y está basado en la misma rutina que se emplea en *Startup.s*.

En cuanto al funcionamiento de los estados de energía del procesador:

- El procesador se dormirá cuando **no haya ni eventos ni mensajes a procesar**. En este caso esperará a la siguiente interrupción, normalmente del *timer0*.
- El procesador se dormirá tras un **tiempo de inactividad**. Se deberá pulsar alguno de los dos botones para reiniciar la ejecución. Pulsar los botones no afectará al estado del programa.

2.6: Integración del proyecto

Antes de desarrollar la lógica final del juego, se implementa en el G_IO el comportamiento de los pines del GPIO necesario. Los pines son:

- Pines 1 y 2 para indicar el **jugador** actual.
- Pines 3 a 9 para marcar la **columna a realizar la jugada**.
- Pines 16 a 18 respectivamente para marcar **jugada realizada**, **jugada no válida** y **final de partida**.
- Pines 30 y 31 para marcar **overflow** y **latido** (procesador despierto o en reposo).



Figura 3: GPIO correspondiente al proyecto resultante de la práctica 2

La interfaz final desarrollada es la siguiente:

```
// Inicializa el gestor de entrada-salida, incluyendo el GPIO
void IO_iniciar(void);

// Altera el estado de GPIO para una nueva partida
void IO_nueva_partida(void);

// Lee la columna introducida
uint8_t IO_leer_entrada(void);

// Alterna el estado del latido
void IO_cambiar_latido(void);

// Apaga el indicador de latido
void IO_bajar_latido(void);

// Apaga el indicador de jugada inválida
void IO_indicar_jugada_valida(void);

// Enciende el indicador de jugada inválida
void IO_indicar_jugada_invalida(void);

// Alterna jugador y marca jugada realizada
void IO_jugada_realizada(void);

// Apaga el indicador de jugada realizada
```

```

void IO_bajar_jugada_realizada(void);

// Indica estado de victoria
void IO_victoria(void);

// Indica estado de empate
void IO_empate(void);

// Enciende el indicador de desborde
void IO_marcar_overflow(void);

```

Finalmente, se procede a implementar el proyecto de la práctica 2 con la siguiente funcionalidad:

- Al iniciar el programa, empieza una nueva partida.
- Entrada
 - Al pulsar el **botón 2**, se inicia una partida nueva.
 - Se puede introducir la columna a jugar en los **pinos 3 a 9** del GPIO
 - Al pulsar el **botón 1** se introduce la jugada
- Se comprueba la entrada de forma periódica programando una alarma para realizar un **CHK_ENTRADA** 10 veces por segundo. Si el usuario marca una y solo una de las columnas en los pines del GPIO, y la columna no está llena, se apaga el pin 17. En caso contrario, se enciende.
- Al introducir la jugada, si la jugada es válida, se llama a la función del conecta 4 para realizar la jugada. Si no lo es, ocurre el comportamiento descrito anteriormente.
- Si la jugada no termina la partida, se alterna de jugador y se indica alternando los pines 1 y 2, además de marcar el pin 16 de jugada realizada Si es victoria, se enciende el pin 18 para marcar final de partida con el pin correspondiente al ganador encendido. Si es empate, se encienden los pines de ambos jugadores.
- Al realizar una jugada, tras 2 segundos se apaga el pin 18 mediante el mensaje **APAGAR_REAL**.
- Al pulsar el botón 2 restablece los pines a su valor por defecto.
- Energía:
 - Se programa una alarma periódica para parpadear el pin 31 con una frecuencia de 2 hercios mientras esté activo o en reposo enviando mensajes **PARPADEAR**.
 - Se programa una alarma **GO_SLEEP** para dormir el procesador tras 10 segundos.
 - Pulsar uno de los dos botones o cambiar el estado de la entrada del GPIO reinicia la alarma **GO_SLEEP**
 - Tras dormir el procesador, pulsar cualquier botón reinicia la ejecución. La pulsación no debe realizar ninguna alteración del estado del juego, solo despertarlo.

Se modificó para este propósito la lógica del Conecta 4, que implementa la siguiente interfaz:

```

// Inicializa el tablero de juego y prepara el sistema para el juego
void conecta4_iniciar(void);

// Devuelve error o el valor de la fila donde se colocará la ficha

```

```
uint8_t conecta4_comprobar_columna(uint8_t columna);

// Introduce la jugada almacenada y trata el siguiente estado del juego
uint8_t conecta4_jugar(void);

// Indica si se ha detenido el juego
uint8_t conecta4_detenido(void);
```

El módulo *conecta4_2022.c* almacena internamente el estado del tablero así como de la jugada leída al comprobar la entrada introducida. También almacena si se está jugando el juego o no.

3. Desarrollo de la práctica 3

3.1: Correcciones a la práctica 2

Aunque funcional, se cometieron una serie de errores durante el diseño inicial del programa.

El problema más importante es cómo se ha mezclado la lógica del juego en tres módulos: el planificador, el juego y el gestor de E/S:

- La función *IO_leer_entrada* contenía, en vez de sólo devolver el valor de la columna o columnas seleccionadas, trataba si había más de una columna seleccionada. Este comportamiento se debería encontrar en la lógica del juego al comprobar la entrada. Al suprimir más adelante la forma en la que se introduce la columna, no se ha corregido directamente este error.
- Por otro lado, la lógica para tratar si la columna es correcta o no, y de la jugada, se encontraba parcialmente en funciones auxiliares del planificador. Este comportamiento también debería encontrarse en el Conecta 4, que a su vez llama a las funciones de los gestores que requiere.

Otros problemas detectados son:

- La gestión de energía requiere conocer si se ha salido o no del modo dormir para poder inhibir el comportamiento por defecto de los botones. La lógica se encontraba en el gestor de energía cuando es más apropiado que sea un estado de la máquina del planificador.
- El gestor de botones contenía lógica adicional para reiniciar la alarma para dormir el procesador que era más apropiado que se encontrara en el planificador.
- Otros problemas menores no relacionados con la estructura del sistema son: un error al calcular el periodo para parpadear el led de latido, un *switch* redundante para lanzar un mensaje a partir de una alarma, etc...
- Aunque no es un error, para facilitar la eliminación de problemas de concurrencia futuros, se han modificado las colas del planificador para que sólo utilicen los índices, y no un contador de elementos insertados, para determinar si la cola está llena o vacía. También se han modificado el cálculo de los índices, utilizando el operador módulo en vez de condicionales.

3.2: Implementación de los modos FIQ y SWI

Para asegurar un mejor funcionamiento del planificador, el temporizador periódico se ha modificado para que sea una interrupción rápida, es decir, una interrupción prioritaria y con su propio modo de procesador, FIQ en vez de IRQ, con registros r8 a r14 propios para evitar sobrescribir valores de otros modos sin apilarlos.

Los tres cambios realizados para activar las FIQ para el *timer0* son:

- Aumentar en *Startup.s* el tamaño de la pila del modo FIQ.
- En el mismo fichero, modificar la etiqueta de salto a la función de captura del modo FIQ (*FIQ_handler*) al *timer0_RSI*.
- Eliminar el uso del VIC para vectorizar la interrupción. Ahora, únicamente se debe indicar en el registro VICIntSelect que el *timer0* es una interrupción rápida.

Por otro lado, se desea implementar una serie de llamadas al sistema para obtener el tiempo y gestionar el estado de las IRQ y FIQ.

Para permitir pasar al modo SWI, se han realizado dos cambios en *Startup.s*:

- Se ha modificado, al final de la rutina de inicio, el orden en el que se cambia de modos, que reservan el espacio sus pilas, tal que entre en el *main* del programa principal en modo usuario.
- La función de captura del SWI (*SWI_Handler*) se ha comentado, y sustituido por una nueva función de captura del mismo nombre en el fichero *SWI.s*, que contiene el código de inicialización y de salida del modo SWI.

La primera llamada al sistema se utiliza para sustituir a *temporizador_leer*, tal que la lectura se realice al modo sistema:

```
// Devuelve el tiempo transcurrido en microsegundos
uint32_t __swi(0) clock_getus(void);

uint32_t __SWI_0 (void) {
    return temporizador_leer();
}
```

Más adelante se definirá la otra llamada al sistema relacionada con el tiempo.

Por otro lado, se requiere una serie de llamadas al sistema para poder inhibir las interrupciones normales y rápidas para garantizar exclusión mutua en la cola de eventos, vulnerable a condiciones de carrera. Sus definiciones están disponibles en *irq_control.h*:

```
// Habilita interrupciones rápidas
void __swi(0xff) enable_fiq(void);

// Habilita interrupciones rápidas
void __swi(0xfe) disable_fiq(void);

// Habilita interrupciones estandar y rápidas
void __swi(0xfd) enable_irq_fiq(void);

// Habilita interrupciones estandar y rápidas
void __swi(0xfc) disable_irq_fiq(void);
```

```
// Devuelve el valor del bit IRQ
uint8_t __swi(0xfb) read_IRQ_bit(void);

// Devuelve el valor del bit FIQ
uint8_t __swi(0xfa) read_FIQ_bit(void);
```

Las funciones están implementadas en el fichero *SWI.s*, donde se opera sobre el registro de estado almacenado del modo del procesador que ha invocado a la función.

Estas funciones se pueden utilizar para crear secciones críticas donde no se pueden producir eventos asíncronos:

```
// Inicio sección crítica
uint8_t irq = read_IRQ_bit(), fiq = read_FIQ_bit();
if (!irq) disable_irq_fiq(); else if (!fiq) disable_fiq();
...
// Final sección crítica
if (!irq) enable_irq_fiq(); else if (!fiq) enable_fiq();
```

3.3: Inclusión del Real-time Clock y del Watchdog

Junto a los temporizadores 0 y 1 utilizados hasta ahora, se ha añadido otros dos temporizadores: el temporizador en tiempo real (RTC), que cuenta el tiempo real transcurrido (en segundos, minutos, horas, días, meses, años, ...), y el *Watchdog*, que se puede utilizar para controlar el tiempo, pero su uso principal es poder reiniciar el procesador tras un tiempo sin reiniciar la cuenta atrás.

Para configurar el RTC, es necesario ajustar los valores para que la velocidad de la cuenta coincida con el tiempo real. Para ello hay que ajustar los registros PREINT y PREFRAC, uno para la parte entera del preincrementador, y el otro para la parte fraccionaria (aproximada) del mismo.

El cálculo del valor de ambos registros se realiza con las siguientes fórmulas (dadas por el manual del procesador):

$$PREINT = \text{ceil}\left(\frac{f_{PCLK}}{32768} - 1\right) \quad (\text{la parte entera})$$

$$PREFRAC = f_{PCLK} - 32768 * (PREINT + 1) \quad (\text{la parte fraccionaria})$$

Activar la cuenta de tiempo es similar a la de los temporizadores mediante el registro CCR.

Para leer el tiempo calculado, se utiliza una llamada al sistema, *clock_gettime*, que a su vez llama a otra función que obtiene los minutos y segundos transcurridos leyendo los registros MIN y SEC, y luego devolviendo el tiempo total en segundos:


```
// Devuelve el tiempo transcurrido en segundos
uint32_t __swi(1) clock_gettime(void);

uint32_t __SWI_1 (void) {
    uint8_t min, seg;
    RTC_leer(&min, &seg);

    return 60 * min + seg;
}
```

En cuanto al *watchdog*, la fórmula para calcular el tiempo es:

$$WDTC = \frac{T(segundos)}{1 / (f_{PCLK} * 4)}$$

El resultado se multiplica por los segundos máximos antes de reiniciar el procesador y se escribe al registro WDTC. Luego se habilita mediante el WDMOD el contador y se indica que reinicia al vencer.

Para iniciar la cuenta, y luego para evitar el reinicio del procesador, se debe escribir, de forma no interrumpida, los bits aa y 55 en el registro WDFEED. La acción se implementa en exclusión mútua mediante *WD_feed*.

3.4: Comunicación por línea serie

La implementación de la comunicación por línea serie se realiza mediante la UART.

3.4.1: UART0

Para configurar la UART0, utilizada en este proyecto, se requiere:

- Seleccionar mediante PINSELO las líneas de transmisión y recepción de mensajes de la UART.
- Escribir la tasa de baudios escribiendo en UODLL, que se puede escribir sólo cuando el bit para activar el DLAB (*latch*) está puesto a 1 (y que luego se debe bajar) mediante el UOLCR.
- Indicar que el tamaño de carácter es de 8 bits en el UOLCR.
- Vectorizar la RSI y la interrupción en el VIC de forma parecida al resto de periféricos.
- Activar interrupciones en el UOIER para cuando se ha completado una escritura (THRE), como cuando se recibe un nuevo carácter en la línea (RDA).

El valor a escribir la tasa de baudios depende de la frecuencia del procesador y de la tasa de baudios deseada.

En la rutina de servicio para tratar las interrupciones, es necesario leer el estado del registro UOIRR para determinar el tipo de interrupción que ha ocurrido:

- Si la interrupción es una lectura (RDA), lee el carácter introducido y envía un evento **UART0_LEER** con el carácter como dato adicional.
- Si la interrupción es una escritura completada (THRE), encola un evento **UART0_ESCRIBIR**, que continuará la escritura.

Para iniciar o continuar las escrituras, se ofrece la siguiente interfaz:

```
// Añade la cadena al búfer y comienza la transmisión
void uart0_enviar_array(char *ch);

// Si el bufer no está vacío, transmite a UART0 el siguiente carácter
void uart0_continuar_envio(void);
```

La primera función añade una cadena de texto a transmitir a una cola de caracteres a enviar, y si no está transmitiendo comienza dicha transmisión. La segunda función continúa la transmisión siempre que hay más caracteres a escribir.

3.4.2: Cola búfer

Para almacenar las cadenas de caracteres a escribir en la UART en orden, al admitir sólo transmisiones carácter a carácter, es necesaria una cola para caracteres. La cola es similar a las otras colas implementadas, con la diferencia de los tipos de datos que almacena, y que permite añadir múltiples elementos tal que estén terminados por el carácter NULL. Su interfaz es:

```
// Añade una cadena de caracteres a la cola
void buffer_anyadir(char* c);

// Devuelve si el búfer está vacío
uint8_t buffer_vacio(void);

// Devuelve el siguiente carácter almacenado en el búfer
int8_t buffer_siguiete(void);
```

3.4.3: Implementación de la comunicación

Para interpretar las cadenas de caracteres que se han añadido con el buffer en la UART, primero se necesita convertir los datos que leerá a caracteres, para lo que usaremos 2 funciones *int to string* (itoa), una para un solo carácter, y otra para cadenas de caracteres. Para otra función, necesitaremos también otra función *string to int* (atoi) como función auxiliar, siguiendo la siguiente interfaz:

```
/*
    Convierte un dígito a carácter
    i: Entero a convertir
*/
```

```

char serie_itoa(uint8_t i) ;
/*
    Convierte a caracter y envía a la línea serie un entero.
    num: Entero a convertir
*/
void serie_itoa_wide(uint32_t num);
/*
    Convierte un caracter a entero
    i: Caracter a convertir. Debe ser un caracter entre '0' y '9'
*/
uint8_t serie_atoi(char i);

```

Leerá el contenido de la UART través de la función *serie_leer*, y empezará a leer los caracteres añadidos a partir del caracter '#', seguido de una cadena de caracteres y finalizando el comando con el caracter '!', indicando "Comando incorrecto" si no sigue uno de los 3 patrones de caracteres que albergan un comando, que son "NEW", "END" y 'C', siendo C un dígito entre 1 y 7, y mandando cada uno de estos un mensaje a encolar con la función esperada.

Además de la función encargada de leer, también hay mensajes determinados, como el de inicio del programa o fin de partida, que están escritos en las funciones que siguen el siguiente interfaz:

```

/*
    Escribe el siguiente caracter del búfer de la línea serie
*/
void serie_escribir(void);

/*
    Genera un mensaje de bienvenida al iniciar
*/
void serie_pantalla_bienvenida(void);

/*
    Genera un mensaje final.

    t_juego: Tiempo tomado en ejecutar el programa
    t_medio: Tiempo medio de procesamiento de mensajes
*/
void serie_mensaje_reinicio(uint32_t t_juego, uint32_t t_medio);

/*
    Envía una cadena cualquiera a la línea serie.

    linea: Cadena a enviar. Debe acabar en NULL

    Se añade un salto de línea a la línea añadida
*/
void serie_print(char* linea);

```

3.5: Integración del proyecto definitivo

La implementación final del GPIO varía ligeramente a la parte de la práctica 2, y ahora consta de los siguientes pines:

- Pines 2 y 3 para indicar el **jugador** actual.
- Pines 14 y 15 cumplen la función de **botón 1** y **botón 2** respectivamente.
- Pines 16 a 18 respectivamente para marcar **jugada realizada**, **jugada no válida** y **final de partida**.
- Pines 30 y 31 para marcar **overflow** y **latido** (procesador despierto o en reposo).



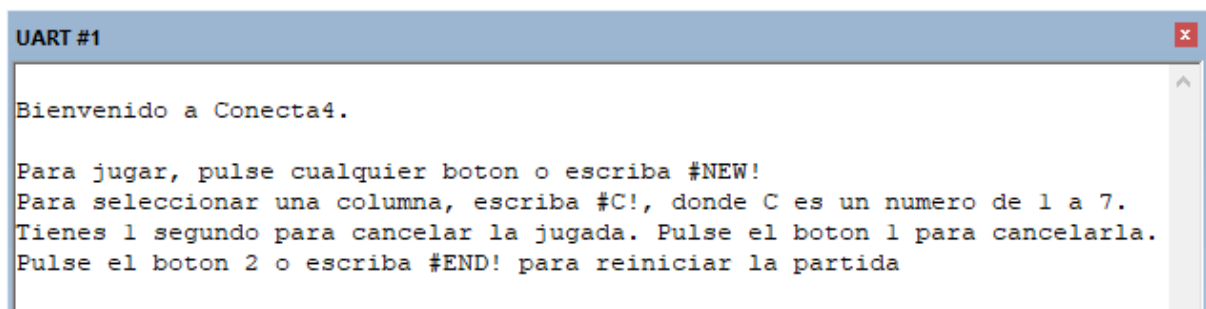
Figura 4: GPIO correspondiente al programa final

Finalmente, se procede a implementar el proyecto de la práctica 3 con la siguiente funcionalidad:

- Al iniciar el programa, se despliegan en la UART varias líneas de texto que explican el funcionamiento correcto del juego. (Figura 5)
- Entrada
 - Al introducir **#NEW!**, o pulsar el **botón 1** o **2**, si no había ninguna partida en curso, se comenzará una nueva partida. (Figura 8)
 - Se puede introducir la columna a jugar escribiendo **#C!**, sustituyendo C por el número de la columna a insertar. (Figura 6)
 - Tras la introducción de una jugada válida, el programa da 1 segundo para pulsar el **botón 1**, lo cual cancelaría el movimiento, y volvería a ofrecer el turno al mismo jugador. (Figura 7)
 - Si hay una partida iniciada, pulsar el **botón 2** o escribir en la UART el comando **#END!**, terminaría la partida. (Figura 8)
- Si el usuario introduce una columna que ya está llena, aparece el mensaje *Columna incorrecta*, y se enciende el pin 17, que se mantendrá hasta que se realice una jugada correcta.
- Si el usuario introduce una columna fuera del rango, o un comando entre # y ! incorrecto, aparecerá en la UART el mensaje *Comando incorrecto*.
- Al introducir la jugada, si la jugada es válida, se previsualiza la jugada. Si no se pulsa el **botón 1** en el segundo posterior a ejecutarse el comando, la jugada será ejecutada correctamente. (Figura 6)

- Si la jugada no termina la partida, se alterna de jugador y se indica alternando los pines 2 y 3, además de marcar el pin 16 de jugada realizada, que se quedará marcado durante 2 segundos. Si es victoria, se enciende el pin 18 para marcar final de partida con el pin correspondiente al ganador encendido. Si es empate, se encienden los pines de ambos jugadores.
- Al realizar una jugada, tras 2 segundos se apaga el pin 18 mediante el mensaje **APAGAR_REAL**.
- Al pulsar el botón 2 restablece los pines a su valor por defecto.
- Energía:
 - Se programa una alarma periódica para parpadear el pin 31 con una frecuencia de 2 hercios mientras esté activo o en reposo enviando mensajes **PARPADEAR**.
 - Se programa una alarma **GO_SLEEP** para dormir el procesador tras 10 segundos.
 - Pulsar uno de los dos botones o escribir algo en la UART reinicia la alarma **GO_SLEEP**

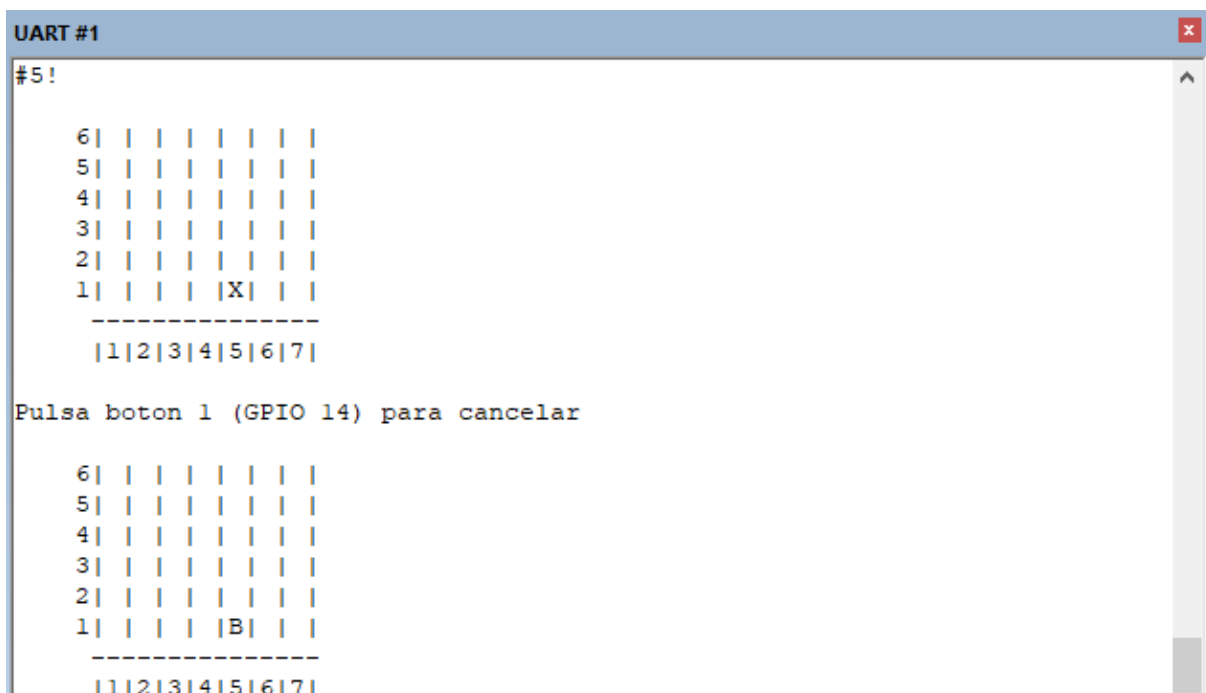
Tras dormir el procesador, pulsar cualquier botón reinicia la ejecución. La pulsación no debe realizar ninguna alteración del estado del juego, solo despertarlo.



```
UART #1
Bienvenido a Conecta4.

Para jugar, pulse cualquier boton o escriba #NEW!
Para seleccionar una columna, escriba #C!, donde C es un numero de 1 a 7.
Tienes 1 segundo para cancelar la jugada. Pulse el boton 1 para cancelarla.
Pulse el boton 2 o escriba #END! para reiniciar la partida
```

Figura 5: Texto inicial desplegado en la UART



```
UART #1
#5!

 6| | | | | | |
 5| | | | | | |
 4| | | | | | |
 3| | | | | | |
 2| | | | | | |
 1| | | |X| | |
-----
|1|2|3|4|5|6|7|

Pulsa boton 1 (GPIO 14) para cancelar

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

Figura 6: Ejemplo de un movimiento correctamente ejecutado

```
UART #1
#5!

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

Pulsa boton 1 (GPIO 14) para cancelar

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

Figura 7: Ejemplo de un movimiento cancelado correctamente

```
UART #1
#END!

Detenido por el usuario
La partida ha durado 44 segundos
Se ha tardado 7 microsegundos de media para procesar un mensaje

Para jugar, pulse cualquier boton o escriba #NEW!
#NEW!

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

Figura 8: Ejemplo de una finalización, y un inicio de partida utilizando los comandos #NEW! y #END!

Para finalizar, esta sería la máquina de estados del juego Conecta4, diseñada en base a los 3 estados del planificador, y los 2 turnos de los distintos jugadores:

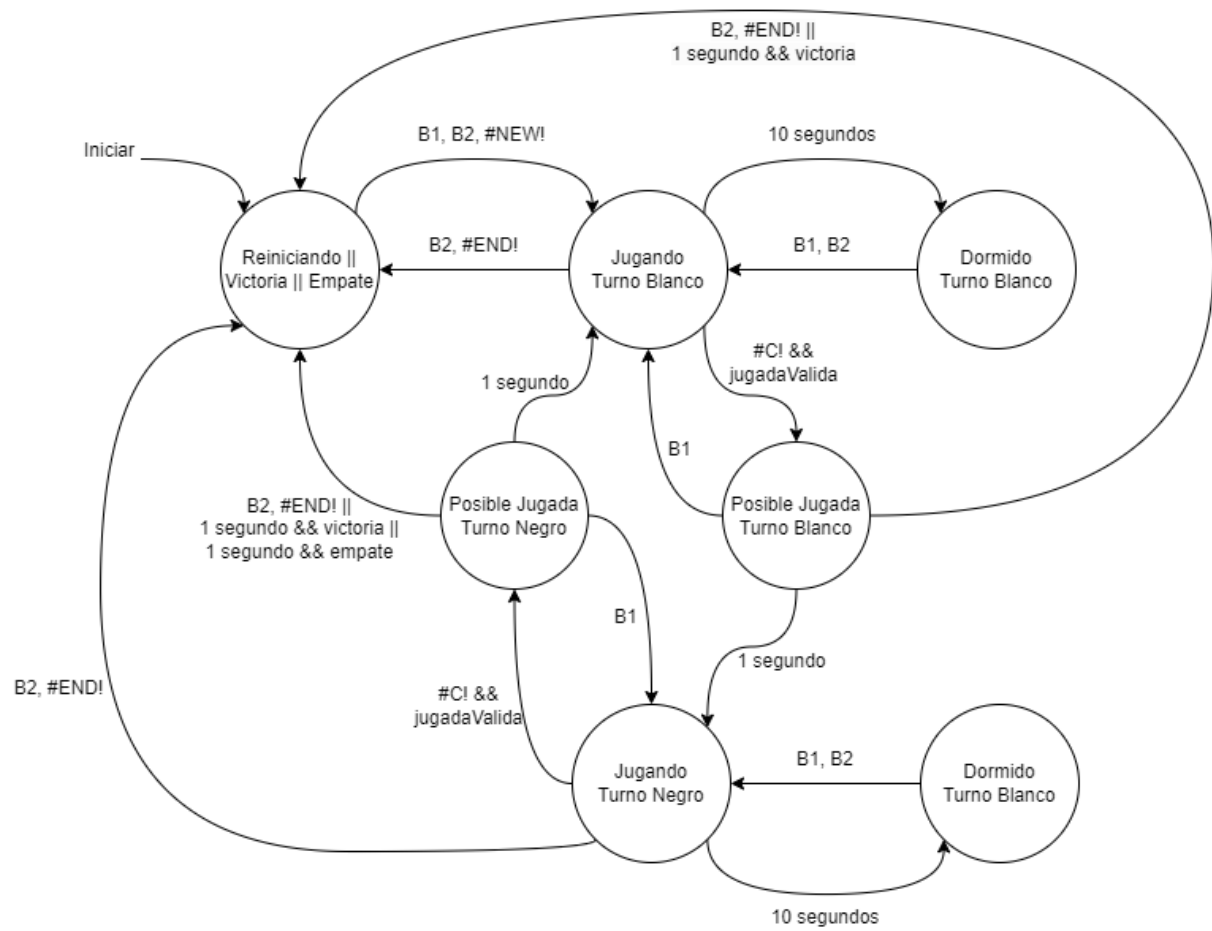


Figura 9: Máquina de estados

4. Conclusiones

Para finalizar, se desea destacar los puntos fuertes y débiles del proyecto realizado, así como una consideración personal de cada integrante del grupo sobre la práctica desarrollada.

Sobre la implementación del propio proyecto, se desea comentar que, a pesar de la dificultad del proyecto, se ha conseguido desarrollar de forma mayoritariamente satisfactoria la práctica. Sin embargo, sí que se han detectado una serie de fallos y carencias que, aunque no críticas, si que se habría deseado subsanar en caso de continuar con el proyecto:

- Un fallo que podría ser potencialmente grave se encuentra en la exclusión mútua: se utiliza una sentencia condicional para determinar si es necesario desactivar todas las interrupciones o sólo las rápidas. Pero al salir se utiliza un *if/else if* para reactivar las interrupciones, con la posibilidad de que no se reactiven por lo menos las interrupciones rápidas (que son necesarias para la ejecución del planificador).
- Para mostrar los caracteres escritos en la UART, en vez de implementar una función para introducir un carácter suelto a la cola búfer (similar a un *puts()*), se escribe directamente en el registro de transmisión, pudiendo sobrecribir un carácter en medio de una escritura.
- En vez de medir el tiempo desde que se encola un mensaje hasta que se trata, se mide el tiempo del tratamiento únicamente, lo cual no es realmente muy útil para medir la calidad de servicio, solo el tiempo de ejecución.
- Continuando el punto anterior, no se ha reanalizado el tiempo de ejecución de las funciones implementadas, ni comparado con los de la primera práctica.

En cuanto a los problemas, el principal problema ha sido las colas y la UART. Se ha tardado un buen tiempo en detectar problemas de desborde y mal cálculo de posiciones en memoria con las implementaciones iniciales, que causaban, por ejemplo, que las colas «pisasen» los datos del gestor de alarmas sin desbordar, o desbordes de colas tras ciertas acciones que han sido difíciles de detectar. En cuanto a la línea serie, el principal problema era sobre los pasos necesarios para implementarlo, pero una vez planteado los errores han sido fácilmente corregidos como con el resto del sistema. Otro error grave fue conceptual en la práctica 2, donde no se ha definido correctamente los límites entre planificador, juego y gestores.

4.1: Comentarios: Dorian Wozniak

En primer lugar, quisiera agradecer a los profesores de la asignatura, quienes han sido amables y nos han ayudado con las dudas que teníamos.

Sobre el proyecto, considero que el mayor problema ha sido organizativo. Buena parte de ambas prácticas se realizó en los últimos días antes de la entrega, y no se ha aprovechado del todo las clases (y como no, el problema no ha sido solo de esta asignatura, con la consecuencia de estar haciendo otras prácticas en la hora de PH). Esto ha estado agravado con un cuatrimestre con menos tiempo que de costumbre.

En cuanto a la nota, considero que se debería situar entre el 8 y 9. Los grandes puntos están cumplidos, pero hay suficientes errores de cierta gravedad y apartados opcionales sin cumplir que empañan lo realizado, aunque no sin desmerecer lo ya hecho.

Finalmente, queda destacar que ha sido útil para conocer cómo funciona la programación en entornos donde se dispone de recursos limitados y las implementaciones se podrían describir como más «manuales» y laboriosas, sin las comodidades de la programación para computadores tradicionales y requiriendo entender y seguir la documentación del fabricante.

4.2: Comentarios: Pablo Latre

Para empezar, agradecer al profesorado ya no solo por la resolución de las dudas, sino también por el interés demostrado, y en buena parte transmitido hacia mi por esta asignatura.

No tengo mucho más que añadir a lo que ha dicho mi compañero, hemos tenido cierto problema organizándonos, pero al final hemos conseguido hacer el juego bastante completo a mi parecer. Si que siento que deberíamos haber aprovechado mejor las horas de clase, ya que las que se aprovechaban bien, no solo se avanzaba bastante, sino que es donde mejor comprendía los conceptos.

Sobre la nota, creo que entre un 8 y un 9 puede estar, ya que, aunque los apartados opcionales no los hemos cumplido, si que creo que en general, ha sido un trabajo bastante bien ejecutado. De paso agradecer a mi compañero, que me ayudó bastante a recuperar el hilo y entender la asignatura, sobre todo en la práctica 2.

Finalmente, respecto a esta asignatura, me gustaría resaltar que trabajar viendo como funciona el código atómicamente en el Keil, y la configuración de los periféricos, ha sido lo que mas me ha llamado la atención.

Anexo A: Código fuente del proyecto

Anexo A.1: Main

Aquí se recoge el código fuente del main del proyecto, que incluye la función main, y el código encargado de tratar los eventos y mensajes.

Anexo A.1.1: main.c

```
// Constantes globales para el planificador
enum {
    ALARM_PERIODO    = 10, // Periodo de actualización de timer0/alarma (ms)
    WD_FEED_PERIODO  = 1    // Periodo de alimentación de watchdog (s)
};

static uint8_t dormido = 0; // Indica si el procesador se ha dormido
static uint8_t jugando = 0; // Indica si hay una partida en curso

/* Trata el evento recibido */
void tratar_evento(evento* e) {
    switch (e -> ID_evento) {
        // Timer 0 -> Comprobar alarmas
        case T0_PERIODO:
            alarma_refrescar(ALARM_PERIODO);
            break;

        // Timer 1 -> Nada
        case T1_OVERFLOW:
            break;

        // Boton 1 -> Cancela jugada (o inicia partida si no ha empezado)
        case BOTON1:
            cola_msg(SET_ALARMA, ALARM_GO_SLEEP_MSG);
            boton1_pulsado();
            if (!dormido) {
                if (jugando) conecta4_cancelar_jugada();
                else cola_msg(INICIAR, 0);
            } else dormido = 0;
            break;

        // Boton 2 -> Detiene la partida (o inicia partida si no ha empezado)
        case BOTON2:
            cola_msg(SET_ALARMA, ALARM_GO_SLEEP_MSG);
            boton2_pulsado();
            if (dormido == 0) {
                if (jugando) {
                    cola_msg(ACABAR, 3);
                } else cola_msg(INICIAR, 0);
            } else dormido = 1;
    }
}
```

```

        break;

// UART0 obtiene un caracter -> Interpreta lo obtenido
case UART0_LEER:
    cola_msg(SET_ALARMA, ALARM_GO_SLEEP_MSG);
    serie_leer(e -> auxData);
    break;

// UART1 termina de escribir un caracter -> Envia siguiente carácter
case UART0_ESCRIBIR:
    serie_escribir();
    break;
    }
}

/* Trata el mensaje recibido */
void tratar_mensaje (msg* m) {
    // Estadísticas de la última partida
    static uint32_t tiempo_total_msg = 0;    // Tiempo total tratando mensajes
    static uint32_t mensajes_procesados = 0; // Mensajes totales procesados
    static uint32_t tiempo_juego = 0;        // Tiempo de inicio del juego

    uint32_t start = clock_getus(); // Empieza a medir siguiente tratamiento

    switch (m -> ID_msg) {
        // Programar/reprogramar/cancelar alarma
        case SET_ALARMA:
            alarma_programar(m -> mensaje);
            break;

        // Comprobar si boton 1 sigue pulsado
        case CHK_BOTON1:
            boton1_comprobar();
            break;

        // Comprobar si boton 2 sigue pulsado
        case CHK_BOTON2:
            boton2_comprobar();
            break;

        // Lee entrada obtenida y prepara acción si fuese necesario
        case CHK_ENTRADA:
            if (jugando) conecta4_comprobar_entrada(m -> mensaje);
            break;

        // Alternar estado del latido
        case PARPADEAR:
            IO_cambiar_latido();
            break;

        // Baja pin de jugada realizada
        case OFF_REALIZAR:
            IO_bajar_jugada_realizada();
            break;

        // Inicia una nueva partida

```

```

    case INICIAR:
        if (!jugando) {
            conecta4_iniciar();
            jugando = 1;
            tiempo_total_msg = 0;
            mensajes_procesados = 0;
            tiempo_juego = clock_gettime();
        }
        break;

    // Compromete la jugada realizada si no es cancela
    case SIGUIENTE:
        conecta4_realizar_jugada();
        break;

    // Actúa si se acaba la partida
    case ACABAR:
        if (jugando) {
            conecta4_detener(m -> mensaje);

            // Calcula tiempo de juego y QoS
            tiempo_total_msg += clock_getus() - start;
            tiempo_juego = clock_gettime() - tiempo_juego;
            serie_mensaje_reinicio(tiempo_juego, tiempo_total_msg /
mensajes_procesados);
            jugando = 0;
        }
        break;

    // Mandar el procesador a dormir
    case GO_SLEEP:
        dormido = 1;
        energia_dormir();
        break;

    // Alimenta Watchdog para no reiniciar
    case FEED_WD:
        WD_feed();
        break;
}

tiempo_total_msg += clock_getus() - start; mensajes_procesados++;
}

int main(void) {
    // Inicializa periféricos
    serie_iniciar();
    IO_iniciar();
    boton_iniciar();
    alarma_iniciar(ALARM_PERIODO, WD_FEED_PERIODO);

    // Structs para almacenar lo desencolado
    evento e = {0, 0};
    msg m = {0, 0};

```

```
// Prepara la partida
cola_msg(SET_ALARMMA, ALARM_GO_SLEEP_MSG);
cola_msg(SET_ALARMMA, ALARM_PARPADear_MSG);
cola_msg(SET_ALARMMA, ALARM_FEED_WD_MSG);

serie_pantalla_bienvenida();

// Bucle del planificador
while (1) {
    // Mira si hay eventos pendientes
    if (cola_hay_eventos()) {
        cola_obtener_sig_evento(&e);
        tratar_evento(&e);
    }

    // Mira si hay mensajes pendientes
    else if (cola_hay_msg()) {
        cola_obtener_sig_msg(&m);
        tratar_mensaje(&m);
    }
    // Si no hay nada que hacer, entra en idle
    else { energia_reposo(); }
}
}
```

Anexo A.2: Conecta 4

Aquí se recoge el código con la lógica del juego conecta4. No mostramos el código íntegro de conecta4_2022.c, ya que de las funciones internas, solo se ha modificado la función mostrar tablero. No incluimos el header, ni los ficheros encargados de proporcionar el tablero (tableros.h) y las celdas (celda.h), ya que los dos últimos en ser mencionados no han sido modificados en estas prácticas.

Anexo A.2.1: conecta4_2022.c

```
/* **** */
/* Funciones internas */

/*
    Imprime mediante la línea serie el tablero actual almacenado
    mostrar_marcada: Si distinta de cero, muestra la posición de la pieza a
                    insertar a continuación
*/
void C4_mostrar_tablero(uint8_t mostrar_marcada) {
    for (uint8_t i = NUM_FILAS; i >= 1; i--) {
        char linea[32] = {' ', ' ', ' ', ' ', ' ', serie_itoa(i)};
        int k = 5;

        for (uint8_t j = 1; j <= NUM_COLUMNAS; j++) {
            linea[k++] = '|';
            if (mostrar_marcada && fila == i && columna == j) linea[k++] = 'X';
            else if (celda_blanca(cuadrícula[i][j])) linea[k++] = 'B';
            else if (celda_negra(cuadrícula[i][j])) linea[k++] = 'N';
            else linea[k++] = ' ';
        }
        linea[k] = '|';
        serie_print(linea);
    }

    serie_print("      -----");
    serie_print("      |1|2|3|4|5|6|7|\n");
}

/* **** */
/* Exportadas */

/*
    Inicializa el tablero para una nueva partida
*/
void conecta4_iniciar() {
    // Limpia el tablero
    for (uint8_t i = 1; i <= NUM_FILAS; i++)
        for (uint8_t j = 1; j <= NUM_COLUMNAS; j++)
            celda_borrar_valor(&cuadrícula[i][j]);

    color = 1; pendiente = 0; columna = 0; fila = 0;

    IO_nueva_partida();
}
```

```

    C4_mostrar_tablero(0);
}

/*
    Dada una columna, determina una jugada y, si se puede insertar, almacena la
    próxima jugada y muestra la posición de la ficha en el tablero.

    c: Valor de la columna donde insertar una ficha
*/
void conecta4_comprobar_entrada(uint32_t c) {
    if (!pendiente && C4_comprobar_columna(c & 0xff)) {
        pendiente = 1;
        IO_indicar_jugada_valida();
        C4_mostrar_tablero(1);
        serie_print("Pulsa boton 1 (GPIO 14) para cancelar\n");
        cola_msg(SET_ALARMA, ALARM_SIG_ON_MSG);
    }

    else {
        IO_indicar_jugada_invalida();
        serie_print("Columna incorrecta");
    }
}

/*
    Si hay una jugada pendiente de realizar, la compromete al tablero y actúa
    según si hay línea, empate o debe cambiar de jugador
*/
void conecta4_realizar_jugada() {
    if (pendiente) {
        C4_actualizar_tablero(fila, columna);

        C4_mostrar_tablero(0);

        pendiente = 0;

        if (C4_hay_linea(fila, columna)) {
            IO_victoria();
            cola_msg(ACABAR, color - 1);
        }
        else if (C4_comprobar_empate()) {
            IO_empate();
            cola_msg(ACABAR, 2);
        }
        else {
            color = C4_alternar_color(color);
            IO_jugada_realizada();
            cola_msg(SET_ALARMA, ALARM_OFF_REALIZAR_MSG);
        }
    }
}

/*
    Si hay una jugada pendiente por realizar, la cancela
*/

```

```
void conecta4_cancelar_jugada() {
    if (pendiente) {
        pendiente = 0;
        C4_mostrar_tablero(0);
    }
}

/*
    Informa del fin del juego, sea voluntario o realizado por el juego
    razon: Motivo por la que se acaba el juego
*/
void conecta4_detener(uint32_t razon) {
    pendiente = 0;
    switch (razon) {
        case 0: serie_print("Victoria de las blancas"); break;
        case 1: serie_print("Victoria de las negras"); break;
        case 2: serie_print("Empate"); break;
        case 3: serie_print("Detenido por el usuario"); break;
    }
}
```


Anexo A.3: Colas

Aquí se incluyen los ficheros encargados de gestionar las distintas de colas que se utilizan en él, además de los que determinan qué tipo de elementos pueden acumular estas (eventos.h y msg.h).

Anexo A.3.1: cola_eventos.c

```
// Constantes
enum {
    TAM_COLA = 32    // Tamaño de la cola
};

// Definición de un elemento de la cola
typedef struct ElementoCola {
    evento        unEvento;    // Evento almacenado
    uint32_t      veces;        // Numero de evento
} elemento ;

static volatile elemento cola[TAM_COLA];    // Cola de eventos
static volatile uint8_t primero = 0;        // Índice al primer elemento
static volatile uint8_t ultimo  = 0;        // Índice al último elemento

/*
    Encola un evento en la cola generado por una IRQ.

    ID_evento: Identificador numérico del evento (ver eventos.h)
    veces:     Número de veces que ha ocurrido el evento
    auxData:   Datos adicionales dados por el evento

    Las escrituras se realizan en exclusión mútua, deshabilitando interrupciones
    rápidas.

    Si se encolan TAM_COLA + 1 elementos, la cola queda desbordada y se para el
    programa
*/
void cola_encolar_eventos(uint8_t ID_evento, uint32_t veces, uint32_t auxData) {
    // Inicio sección crítica
    uint8_t fiq = read_FIQ_bit() ; if (!fiq) disable_fiq();

    // Si desborda se detiene.
    if ((ultimo + 1) % TAM_COLA == primero) {
        IO_marcar_overflow();
        while (1);
    }
    // Añade a la cola
    cola[ultimo].unEvento.ID_evento = ID_evento;
    cola[ultimo].unEvento.auxData   = auxData;
    cola[ultimo].veces               = veces;

    // Avanza índice
    ultimo = (ultimo + 1) % TAM_COLA;
```

```

    // Fin sección crítica
    if (!fiq) enable_fiq();
}

/*
    Encola un evento en la cola generado por una FIQ.

    ID_evento: Identificador numérico del evento (ver eventos.h)
    veces:     Número de veces que ha ocurrido el evento
    auxData:   Datos adicionales dados por el evento

    Las escrituras se realizan en exclusión mútua (FIQ lo hace por defecto).

    Si se encolan TAM_COLA + 1 elementos, la cola queda desbordada y se para el
    programa
*/
void cola_encolar_eventos_fiq(uint8_t ID_evento, uint32_t veces, uint32_t auxData) {
    // Si desborda se detiene.
    if ((ultimo + 1) % TAM_COLA == primero) {
        IO_marcar_overflow();
        while (1);
    }
    // Añade a la cola
    cola[ultimo].unEvento.ID_evento = ID_evento;
    cola[ultimo].unEvento.auxData   = auxData;
    cola[ultimo].veces               = veces;

    // Avanza índice
    ultimo = (ultimo + 1) % TAM_COLA;
}

/*
    Devuelve si la cola está vacía
*/
uint8_t cola_hay_eventos() { return ultimo != primero; }

/*
    Devuelve el siguiente evento de la cola.

    e: Dirección donde almacenar el evento obtenido

    La cola de eventos no puede estar vacía, en caso contrario el resultado
    queda indefinido. Las lecturas se realizan en exclusión mútua,
    deshabilitando interrupciones.
*/
void cola_obtener_sig_evento(evento *e) {
    // Inicio sección crítica
    uint8_t irq = read_IRQ_bit(), fiq = read_FIQ_bit();
    if (!irq) disable_irq_fiq(); else if (!fiq) disable_fiq();

    // Obtiene el evento
    *e = cola[primero].unEvento;

    // Avanza índice
    primero = (primero + 1) % TAM_COLA;
}

```

```

// Final sección crítica
if (!irq) enable_irq_fiq(); else if (!fiq) enable_fiq();
}

```

Anexo A.3.2: eventos.h

```

// Tipos de eventos disponibles
enum {
    EV_INDEFINIDO    = 0,    // Evento indefinido

    T0_PERIODO       = 1,    // Timer0 alcanza un periodo
    T1_OVERFLOW      = 2,    // Timer1 desborda

    BOTON1           = 3,    // Se pulsa el botón 1
    BOTON2           = 4,    // Se pulsa el botón 2

    UART0_LEER       = 5,    // Se ha leído un nuevo caracter de la UART
                                // auxData: Caracter leído
    UART0_ESCRIBIR   = 6,    // Se ha escrito el caracter en la UART
};

// Definición de un evento
typedef struct EventoHW {
    uint8_t ID_evento;        // Identificador de evento
    uint32_t auxData;         // Datos auxiliares del evento
} evento ;

```

Anexo A.3.3: cola_msg.c

```

// Constantes
enum {
    TAM_COLA = 32    // Tamaño de la cola
};

// Definición de un elemento de la cola
typedef struct ElementoCola {
    msg        unMsg;        // Mensaje almacenado
    uint32_t    tiempo;       // Instante al encolar (us)
} elemento ;

static volatile elemento cola[TAM_COLA];    // Cola de mensajes
static volatile uint8_t primero = 0;        // Índice al primer elemento
static volatile uint8_t ultimo = 0;        // Índice al último elemento

/*
    Encola un mensaje en la cola.

    ID_evento: Identificador numérico del tipo de mensaje

```

```

    mensaje:    Contenido del mensaje enviado

    Si se encolan TAM_COLA + 1 elementos, la cola queda desbordada y se para el
    programa
*/
void cola_msg(uint8_t ID_msg, uint32_t mensaje) {
    // Si desborda se detiene.
    if ((ultimo + 1) % TAM_COLA == primero) {
        IO_marcar_overflow();
        while (1);
    }

    // Añade a la cola
    cola[ultimo].unMsg.ID_msg    = ID_msg;
    cola[ultimo].unMsg.mensaje   = mensaje;
    cola[ultimo].tiempo          = clock_getus();

    // Avanza índice
    ultimo = (ultimo + 1) % TAM_COLA;
}

/*
    Devuelve si la cola está vacía.
*/
uint8_t cola_hay_msg() { return ultimo != primero; }

/*
    Devuelve el siguiente mensaje de la cola.

    m: Dirección donde almacenar el mensaje obtenido

    La cola de eventos no puede estar vacía, en caso contrario el resultado
    queda indefinido.
*/
void cola_obtener_sig_msg(msg *m) {
    // Obtiene el evento
    *m = cola[primero].unMsg;

    // Avanza índice
    primero = (primero + 1) % TAM_COLA;
}

```

Anexo A.3.4: msg.h

```

// Tipos de mensajes disponibles
enum {
    MSG_INDEFINIDO = 0,    // Mensaje indefinido
    SET_ALARMA     = 1,    // Añadir alarma
    CHK_BOTON1     = 2,    // Comprobar si botón 1 esta pulsado
    CHK_BOTON2     = 3,    // Comprobar si botón 2 esta pulsado
    CHK_ENTRADA    = 4,    // Preparar jugada si es válida
}

```

```

    PARPADEAR      = 5,      // Alternar LED latido
    OFF_REALIZAR   = 6,      // Apagar LED jugada realizada

    INICIAR        = 7,      // Iniciar nueva partida
    SIGUIENTE      = 8,      // Comprometer jugada
    ACABAR         = 9,      // Terminar partida

    GO_SLEEP       = 10,     // Dormir procesador
    FEED_WD        = 11     // Alimentar WD para evitar reinicio
};

// Definición de un evento
typedef struct MensajeG {
    uint8_t ID_msg;      // Identificador de mensaje
    uint32_t mensaje;    // Contenido
} msg

```

Anexo A.3.5: cola_buffer.c

```

// Constantes
enum {
    TAM_COLA = 1024    // Tamaño de la cola
};

static int8_t cola[TAM_COLA];      // Cola
static uint32_t primero = 0;      // Índice al primer elemento
static uint32_t ultimo = 0;      // Índice al último elemento

/*
    Añade una cadena de caracteres a la cola

    c: Cadena de caracteres a introducir

    Si se desborda, para el programa. Las cadenas deben estar terminadas en NULL
*/
void buffer_anyadir(char* c) {
    for (uint32_t i = 0; c[i] != '\0'; i++) {
        // Si desborda se detiene.
        if ((ultimo + 1) % TAM_COLA == primero) {
            IO_marcar_overflow();
            while (1);
        }
        // Añade a la cola
        cola[ultimo] = c[i];

        // Avanza índice
        ultimo = (ultimo + 1) % TAM_COLA;
    }
}

/*
    Devuelve si el búfer está vacío

```

```

*/
uint8_t buffer_vacio() { return ultimo == primero; }

/*
    Devuelve el siguiente caracter almacenado en el búfer

    El búfer no debe estar vacío al extraer
*/
int8_t buffer_siguiete() {
    // Obtiene el evento
    uint8_t c = cola[primero];

    // Avanza índice
    primero = (primero + 1) % TAM_COLA;

    return c;
}

```

Anexo A.4: Periféricos

Se incluye en este apartado del anexo el código que se encarga de la correcta con de los periféricos utilizados en este proyecto. No se incluyen los headers en esta memoria.

Anexo A.4.1: tiempo.c

```

// Constantes para registros de los timers
enum {
    // PCLK = Xtal * 5 / 4

    TIME_SEC    = 3750000 - 1, // 60 MHz -> 1s      = 1 / (1/PCLK * 4)
    TIME_MS     = 15000 - 1,   // 60 Mhz -> 1ms   = (10 ^-3) / (1/PCLK * 4)
    TIME_US     = 15 - 1,      // 60 Mhz -> 1us   = (10 ^-6) / (1/PCLK * 4)

    // Timer0/1
    MR0_INT      = 0x1,
    MR0_RESET    = 0x2,

    TCR_ENABLE   = 0x1,
    TCR_RESET    = 0x2,

    // RTC
    PREINT_60MHZ = 456, // 60 MHz -> (PCLK / 32768) - 1,
    PREFRAC_60MHZ = 27024, // 60 MHz -> (PCLK) - (32768 * (PREINT + 1)),

    CCR_ENABLE   = 0x1,
    CCR_RESET    = 0x2,

    // WD
    WD_ENABLE    = 0x1,
    WD_RESET     = 0x2
}

```

```

};

// Constantes para interrupciones vectorizadas
enum {
    // Timer0
    POS_TIMER0 = 4,
    INT_TIMER0 = (0x1 << POS_TIMER0),

    // Timer1
    POS_TIMER1 = 5,
    INT_TIMER1 = (0x1 << POS_TIMER1),

    // VIC
    VIC_CNTL_ENABLE = 0x20
};

/*****
/* TIMER 0 */

/*
    Rutina de servicio para Timer0. Se interrumpe cada vez que vence el periodo
    dado, y encola un evento T0_PERIODO. La interrupción es FIQ.
*/
void timer0_RSI (void) __irq {
    // Contador de interrupciones Timer0
    static volatile uint32_t timer0_count = 0;

    // Trata la interrupción
    timer0_count++;
    cola_encolar_eventos_fiq(T0_PERIODO, timer0_count, 0);

    // Baja la interrupción
    T0IR = 1;
}

/*
    Inicia el periférico Timer0 con el periodo dado.

    periodo:    Tiempo en milisegundos tras el cual se realiza una interrupción

    Se configura como FIQ, el handler se define en el fichero Startup.s.
*/
void temporizador_reloj(uint32_t periodo) {
    // Ajusta la cuenta para contar ms e interrumpir cuando alcance periodo
    T0PR = TIME_MS;
    T0MR0 = periodo - 1;
    T0MCR |= MR0_INT | MR0_RESET;

    // Reinicia TC
    T0TCR |= TCR_ENABLE | TCR_RESET;
    T0TCR &= ~TCR_RESET;

    // Habilita FIQ
    VICIntSelect |= INT_TIMER0;
    VICIntEnable |= INT_TIMER0;
}

```

```

/*****
/* TIMER 1 */

static volatile uint32_t timer1_count = 0; // Contador de interrupciones Timer1

/*
    Rutina de servicio para Timer1. Se interrumpe cada vez desborda el contador,
    y encola un evento T1_OVERFLOW.
*/
void timer1_RSI (void) __irq {
    // Trata la interrupción
    timer1_count++;

    // Evento encolado para efectos de depuración
    cola_encolar_eventos(T1_OVERFLOW, timer1_count, 0);

    // Baja la interrupción
    T1IR = 1;
    VICVectAddr = 0;
}

/*
    Inicia el periférico Timer1. El timer queda asociado a VIC[2].
*/
void temporizador_iniciar() {
    // Ajusta la cuenta para contar en us e interrumpir cuando desborde
    T1PR = TIME_US;
    T1MR0 = 0xffffffff - 1;
    T1MCR |= MR0_INT | MR0_RESET;

    // Configura VIC[1] para tratar timer1
    VICVectAddr1 = (uint32_t) timer1_RSI;
    VICVectCntl1 |= VIC_CNTL_ENABLE | POS_TIMER1;
}

/*
    Inicia a contar el tiempo mediante timer1.
*/
void temporizador_empezar() {
    // Reinicia la cuenta de interrupciones
    timer1_count = 0;

    // Reinicia y arranca TC
    T1TCR |= TCR_ENABLE | TCR_RESET;
    T1TCR &= ~TCR_RESET;

    // Habilita interrupciones para timer1
    VICIntEnable |= INT_TIMER1;
}

/*
    Lee el instante de tiempo del Timer1
*/
uint32_t temporizador_leer() {
    // Lee tiempo actual

```



```

uint32_t time = T1TC;

// Añade las veces que se ha desbordado (cada ~71 min)
for (int i = timer1_count; i > 0; i--) time += 0xffffffff;

return time;
}

/*
Devuelve el tiempo transcurrido en microsegundos
*/
uint32_t __SWI_0 (void) {
return temporizador_leer();
}

/*
Detiene el contador Timer1 junto al instante tiempo al detenerse
*/
uint32_t temporizador_parar() {
// Detiene timer
T1TCR &= ~TCR_ENABLE;
VICIntEnClr |= INT_TIMER1;

return temporizador_leer();
}

/*****
*/ RTC */

/*
Inicia el contador RTC
*/
void RTC_init() {
// Ajusta velocidad de cuenta para 60Mhz
PREINT |= PREINT_60MHZ;
PREFRAC |= PREFRAC_60MHZ;

// Inicia timer
CCR |= CCR_ENABLE | CCR_RESET;
CCR &= ~CCR_RESET;
}

/*
Devuelve los minutos y segundos almacenados por el RTC
*/
void RTC_leer(uint8_t *min, uint8_t *seg) {
*min = MIN;
*seg = SEC;
}

/*
Devuelve el tiempo transcurrido en segundos
*/
uint32_t __SWI_1 (void) {
uint8_t min, seg;
RTC_leer(&min, &seg);

```

```

    return 60 * min + seg;
}

/*****
/* WD */

/*
    Inicia el contador Watchdog con el tiempo dado para reiniciar.

    sec:    Número de segundos a transcurrir antes de reiniciar el programa
*/
void WD_init(uint32_t sec) {
    // Establece el tiempo máximo antes de reiniciar
    WDTC = TIME_SEC * sec;
    WDMOD = WD_ENABLE | WD_RESET;

    // Inicia cuenta atrás
    WD_feed();
}

/*
    Alimenta al WD para reiniciar la cuenta atrás para reiniciar el programa.

    La secuencia de alimentación se debe realizar en exclusión mútua
*/
void WD_feed() {
    // Inicio sección crítica
    uint8_t irq = read_IRQ_bit(), fiq = read_FIQ_bit();
    if (!irq) disable_irq_fiq(); else if (!fiq) disable_fiq();

    // Secuencia de alimentación
    WDFEED = 0xaa;
    WDFEED = 0x55;

    // Final sección crítica
    if (!irq) enable_irq_fiq(); else if (!fiq) enable_fiq();
}

```

Anexo A.4.2: gpio.c

```

/*
    Devuelve una máscara para leer/escribir en GPIO.

    bit_inicial:    Primer bit desde donde empieza la máscara
    num_bits:        Longitud de la máscara

    La máscara puede quedar truncada.
*/
uint32_t GPIO_mascara(uint8_t bit_inicial, uint8_t num_bits) {
    uint32_t mask = 0;

```

```

    for (uint8_t i = num_bits; i > 0; i--) { mask <= 1; mask |= 0x1; }

    return mask << bit_inicial;
}

/*
    Inicializa todos los pines de GPIO
*/
void GPIO_iniciar() { IOPIN = 0x0; }

/*
    Lee una serie de pines de GPIO.
    bit_inicial:    Primer bit desde donde empieza la lectura
    num_bits:       Número de bits a leer
*/
uint32_t GPIO_leer(uint8_t bit_inicial, uint8_t num_bits)
{ return (IOPIN & GPIO_mascara(bit_inicial, num_bits)) >> bit_inicial; }

/*
    Escribe en una serie de pines de GPIO.

    bit_inicial:    Primer bit desde donde empieza la escritura
    num_bits:       Número de bits sobre los que escribir
    valor:          Valor a escribir
    No se escribirá sobre pines de entrada.
*/
void GPIO_escribir(uint8_t bit_inicial, uint8_t num_bits, uint32_t valor) {
    IOCLR |= (GPIO_mascara(0, num_bits) & ~valor) << bit_inicial;
    IOSET |= (GPIO_mascara(0, num_bits) & valor) << bit_inicial;
}

/*
    Marca una serie de pines de GPIO como entrada.

    bit_inicial:    Primer bit desde donde empezar a marcar
    num_bits:       Número de bits marcados

    El usuario podrá cambiar el estado del pin, pero no el programa.
*/
void GPIO_marcar_entrada(uint8_t bit_inicial, uint8_t num_bits)
{ IODIR &= ~GPIO_mascara(bit_inicial, num_bits); }

/*
    Marca una serie de pines de GPIO como salida.

    bit_inicial:    Primer bit desde donde empezar a marcar
    num_bits:       Número de bits marcados

    El usuario no podrá cambiar el estado del pin, pero el programa si.
*/
void GPIO_marcar_salida(uint8_t bit_inicial, uint8_t num_bits)
{ IODIR |= GPIO_mascara(bit_inicial, num_bits); }

```

Anexo A.4.3: power.c

```
enum {
    PCON_IDLE      = 0x1,
    PCON_SLEEP     = 0x2,
    WAKE_EINT_1_2   = (0x3 << 1)
};

/*
    Reinicia el PLL tras salir del modo dormir
*/
void reiniciar_pll(void);

/*
    Coloca el procesador en modo reposo
*/
void power_idle() {
    // Entra en reposo
    PCON |= PCON_IDLE;
}

/*
    Coloca el procesador en modo dormir
*/
void power_down() {
    // Permite despertar procesador al activar EINT1 o EINT2
    EXTWAKE |= WAKE_EINT_1_2;
    // Entra en power-down
    PCON |= PCON_SLEEP;
    reiniciar_pll();
}
```

Anexo A.4.4: boton.c

```
// Constantes para interrupciones vectorizadas
enum {
    POS_EINT1 = 15, // Interrupción 15
    POS_EINT2 = 16, // Interrupción 16

    INT_EINT1 = (0x1 << POS_EINT1), // Bit 15
    INT_EINT2 = (0x1 << POS_EINT2), // Bit 16

    VIC_CNTL_ENABLE = 0x20
};

// Constantes para EINT y PINSEL
enum {
    PIN_14_0 = (0x3 << 28),
    PIN_14_2 = (0x2 << 28),

    //PIN_15_0 = (0x3 << 30),
}
```

```

        //PIN_15_2 = (0x2 << 30),

        EXTINT_1 = (0x1 << 1),
        EXTINT_2 = (0x1 << 2)
};

/*****
/* EINT1 */

/*
    Rutina de servicio para EINT1
*/
void eint1_RSI (void) __irq {
    static volatile uint32_t eint1_count = 0;

    // Trata la interrupción
    eint1_count++;
    cola_enqueue_eventos(BOTON1, eint1_count, 0);
    // Baja interrupción y deshabilita EINT1 temporalmente
    VICIntEnClr |= INT_EINT1;
    EXTINT |= EXTINT_1;
    VICVectAddr= 0;
}

/*
    Inicializa la interrupción externa EINT1
*/
void eint1_iniciar() {
    // Selecciona GPIO(14) como EINT1
    PINSEL0 &= ~PIN_14_0;
    PINSEL0 |= PIN_14_2;

    // Activa EXTINT1 en VIC[2]
    VICVectAddr2 = (uint32_t) eint1_RSI;
    VICVectCntl2 = VIC_CNTL_ENABLE | POS_EINT1 ;

    eint1_habilitar();
}

/*
    Habilita EINT1 para poder interrumpir
*/
void eint1_habilitar() {
    VICIntEnable |= INT_EINT1;
}

/*
    Obtiene el estado de la interrupción EINT1
*/
uint32_t eint1_leer() {
    EXTINT |= EXTINT_1;
    VICVectAddr = 0;
    return ((EXTINT & EXTINT_1) >> 1);
}

*****/

```

```

/* EINT2 */

/*
    Rutina de servicio para EINT2
*/
void eint2_RSI (void) __irq {
    static volatile uint32_t eint2_count = 0;

    // Trata la interrupción
    eint2_count++;
    cola_encolar_eventos(BOTON2, eint2_count, 0);
    // Baja interrupción y deshabilita EINT1 temporalmente
    VICIntEnClr |= INT_EINT2;
    EXTINT |= EXTINT_2;
    VICVectAddr= 0;
}

/*
    Inicializa la interrupción externa EINT2
*/
void eint2_iniciar() {
    // Selecciona GPIO(15) como EINT2
    PINSEL0 &= ~(0xc0000000);
    PINSEL0 |= (0x80000000);

    // Activa EXTINT3 en VIC[3]
    VICVectAddr3 = (uint32_t) eint2_RSI;
    VICVectCntl3 = VIC_CNTL_ENABLE | POS_EINT2 ;

    eint2_habilitar();
}

/*
    Habilita EINT2 para poder interrumpir
*/
void eint2_habilitar() {
    VICIntEnable |= INT_EINT2;
}

/*
    Obtiene el estado de la interrupción EINT2
*/
uint32_t eint2_leer() {
    EXTINT |= EXTINT_2;
    VICVectAddr = 0;
    return ((EXTINT & EXTINT_2) >> 2);
}

```

Anexo A.4.5: uart.c

```
// Constantes para configurar la UART
enum {
    PIN_0_0 = (0x3 << 0),
    PIN_0_1 = (0x1 << 0),

    PIN_1_0 = (0x3 << 2),
    PIN_1_1 = (0x1 << 2),

    LCR_8BIT = 0x03,          // Caracteres de 8 bits
    LCR_DLAB = 0x80,          // Activar latch de tasa de símbolos

    IER_EN_RDA = 0x1,         // Habilitar interrupciones por datos disponibles
    IER_EN_THRE = 0x2,        // Habilitar interrupciones por registro de
                                // transmisión vacío
    UART0_BAUDRATE = 97,      // Tasa de baudios para la frecuencia dada

    IIR_INT_ID = 0xe,         // Máscara para identificación de interrupciones
    IIR_RDA = 0x5,            // Interrupcion de recepción
    IIR_THRE = 0x2            // Interrupción de fin de transmisión
};

// Constantes para configurar el VIC
enum {
    POS_UART0 = 6,
    INT_UART0 = (0x1 << POS_UART0),

    VIC_CNTL_ENABLE = 0x20
};

/*
    Rutina de servicio para UART0
*/
void uart0_RSI (void) __irq {
    static volatile uint32_t uart0_read_count = 0;
    static volatile uint32_t uart0_write_count = 0;

    uint32_t flag = U0IIR & IIR_INT_ID;
    // Lectura de UART
    if (flag & IIR_RDA) {
        uart0_read_count++;
        uint32_t data = U0RBR;
        cola_encolar_eventos(UART0_LEER, uart0_read_count, U0RBR);
    }
    // Escritura a UART
    else if (flag & IIR_THRE) {
        uart0_write_count++;
        cola_encolar_eventos(UART0_ESCRIBIR, uart0_write_count, 0);
    }
    VICVectAddr = 0;
}

/*
    Inicializa el UART0
*/>
```

```

*/
void uart0_init() {
    PINSEL0 &= ~PIN_0_0;
    PINSEL0 |= PIN_0_1;
    PINSEL0 &= ~PIN_1_0;
    PINSEL0 |= PIN_1_1;

    U0LCR |= LCR_8BIT | LCR_DLAB;
    U0DLL = UART0_BAUDRATE;
    U0LCR &= ~LCR_DLAB;

    VICVectAddr0 = (uint32_t) uart0_RSI;
    VICVectCntl0 |= VIC_CNTL_ENABLE | POS_UART0;

    U0IER |= IER_EN_RDA | IER_EN_THRE;
    VICIntEnable |= INT_UART0;
}

/*
    Escribe un caracter inmediatamente en la UART0

    ch: caracter a escribir
*/
void uart0_echo(uint8_t ch) {
    U0THR = ch;
}

/*
    Encola una cadena de caracteres al búfer asociado a la UART0. Si el búfer\
    estaba vacío al añadir, se escribe el primer caracter a transmitir

    ch: Cadena de caracteres. Debe terminar en NULL
*/
void uart0_enviar_array(char* ch) {
    uint8_t estabaVacio = buffer_vacio();
    buffer_anyadir(ch);
    if (estabaVacio) U0THR = buffer_siguiete();
}

/*
    Si el bufer no está vacío, transmite a UART0 el siguiente caracter
*/
void uart0_continuar_envio() {
    if (!buffer_vacio()) {
        U0THR = buffer_siguiete();
    }
}

```


Anexo A.4.6: irq_control.h

```
// Habilita interrupciones rápidas
void __swi(0xff) enable_fiq(void);

// Habilita interrupciones rápidas
void __swi(0xfe) disable_fiq(void);

// Habilita interrupciones estandar y rápidas
void __swi(0xfd) enable_irq_fiq(void);

// Habilita interrupciones estandar y rápidas
void __swi(0xfc) disable_irq_fiq(void);

// Devuelve el valor del bit IRQ
uint8_t __swi(0xfb) read_IRQ_bit(void);

// Devuelve el valor del bit FIQ
uint8_t __swi(0xfa) read_FIQ_bit(void);
```

Anexo A.4.7: pll.s

```
; Descripción:
; Implementación del reinicio del PLL tras despertar del estado
; power-down tal y como se configura durante el startup

AREA datos, DATA

; Definiciones de constantes para gestionar PLL
PLL_BASE EQU 0xE01FC080 ; PLL Base Address
PLLCON_OFS EQU 0x00 ; PLL Control Offset
PLLCFG_OFS EQU 0x04 ; PLL Configuration Offset
PLLSTAT_OFS EQU 0x08 ; PLL Status Offset
PLLFEED_OFS EQU 0x0C ; PLL Feed Offset
PLLCON_PLLE EQU (1<<0) ; PLL Enable
PLLCON_PLLC EQU (1<<1) ; PLL Connect
PLLCFG_MSEL EQU (0x1F<<0) ; PLL Multiplier
PLLCFG_PSEL EQU (0x03<<5) ; PLL Divider
PLLSTAT_PLOCK EQU (1<<10) ; PLL Lock Status

; // <e> PLL Setup
; // <o1.0..4> MSEL: PLL Multiplier Selection
; // <1-32><#-1>
; // <i> M Value
; // <o1.5..6> PSEL: PLL Divider Selection
; // <0=> 1 <1=> 2 <2=> 4 <3=> 8
; // <i> P Value
; // </e>
PLL_SETUP EQU 1
PLLCFG_Val EQU 0x00000024
```

```

AREA codigo, CODE

EXPORT reiniciar_pll

reiniciar_pll
    LDR    R0, =PLL_BASE
    MOV    R1, #0xAA
    MOV    R2, #0x55

; Configure and Enable PLL
    MOV    R3, #PLLCFG_Val
    STR    R3, [R0, #PLLCFG_OFS]
    MOV    R3, #PLLCON_PLLE
    STR    R3, [R0, #PLLCON_OFS]
    STR    R1, [R0, #PLLFEED_OFS]
    STR    R2, [R0, #PLLFEED_OFS]

; Wait until PLL Locked
PLL_Loop    LDR    R3, [R0, #PLLSTAT_OFS]
            ANDS   R3, R3, #PLLSTAT_PLOCK
            BEQ    PLL_Loop

; Switch to PLL Clock
    MOV    R3, #(PLLCON_PLLE:OR:PLLCON_PLLC)
    STR    R3, [R0, #PLLCON_OFS]
    STR    R1, [R0, #PLLFEED_OFS]
    STR    R2, [R0, #PLLFEED_OFS]

    BX     r14

END

```

Anexo A.5: Gestores

En este apartado del anexo se encuentra el código de los gestores, que se encargan de controlar la información obtenida desde los periféricos, y su correcta utilización. También se incluye el fichero que contiene los distintos tipos de alarmas que se utilizan en este proyecto.

Anexo A.5.1: G_Alarm.c

```
// Constantes
enum {
    NUM_ALARMAS = 8
};

// Booleanos
enum {
    ERROR = 0xFF
};

// Tipo alarma
typedef struct AlarmaGestor {
    uint8_t    mensaje;    // Mensaje programado
    uint8_t    periodico;  // Alarma periódica
    uint32_t    retardo;    // Retardo programado (ms)
    uint32_t    restante;  // Tiempo restante (ms)
} alarma ;

// Vector de alarmas
static __align(8) alarma alarmas[NUM_ALARMAS];

static uint8_t alarmas_activas = 0; // Cuenta de alarmas activas

/*
    Pre: ---
    Post: Devuelve (bit_inicial + num_bits - 1, bit_inicial) = 1
*/
uint32_t alarma_mascara(uint8_t bit_inicial, uint8_t num_bits) {
    uint32_t mask = 0;

    for (uint8_t i = num_bits; i > 0; i--) { mask <<= 1; mask |= 0x1; }

    return mask << bit_inicial;
}

/*
    Pre: ---
    Post: Devuelve índice al alarma dado el tipo de mensaje programado
         Devuelve ERROR si no hay alarmas con ese tipo de mensaje
*/
uint8_t alarma_obtener(uint8_t mensaje) {
    for (int8_t i = 0; i < NUM_ALARMAS; i++)
        if (alarmas[i].mensaje == mensaje)
            return i;
}
```

```

    return ERROR;
}

/*
    Pre: ---
    Post: Inicializa el gestor de alarmas, incluyendo los periféricos Timer0 con
          el periodo dado y Timer1
*/
void alarma_iniciar(uint32_t alarm_periodo, uint32_t wd_periodo) {
    // Contador de tiempo
    temporizador_iniciar();
    temporizador_empezar();

    // Temporizador
    temporizador_reloj(alarm_periodo);

    // RTC
    RTC_init();

    // WD
    WD_init(wd_periodo);
}

/*
    Pre: i < NUM_ALARMAS
    Post: Añade una alarma en el índice i con los parámetros dados
*/
void alarma_crear(uint8_t i, uint8_t mensaje, uint8_t periodico,
                  uint32_t retardo) {
    alarmas[i].mensaje      = mensaje;
    alarmas[i].periodico    = periodico;
    alarmas[i].retardo      = retardo;
    alarmas[i].restante     = retardo;
}

/*
    Pre: i < NUM_ALARMAS
    Post: Elimina el alarmam en el índice i
*/
void alarma_eliminar(uint8_t i) {
    alarmas[i].mensaje      = MSG_INDEFINIDO;
    alarmas[i].periodico    = 0;
    alarmas[i].retardo      = 0;
    alarmas[i].restante     = 0;
}

/*
    Pre: alarmas_activas < NUM_ALARMAS, recibido mensaje SET_ALARMA
    Post: Programa, reprograma o cancela la alarma dada por el mensaje dado:

        Si hay una alarma programada para el mensaje dado:
        - Si retardo = 0 -> Cancela
        - Si no -> Reprograma con el retardo dado
        En caso contrario, programa una alarma con los parámetros del mensaje

```

```

*/
void alarma_programar(uint32_t msg) {

    // Obtiene los parámetros del mensaje
    uint8_t mensaje      = ((msg & alarma_mascara(24, 8)) >> 24);
    uint8_t periodico    = ((msg & alarma_mascara(23, 1)) >> 23);
    uint32_t retardo     = (msg & alarma_mascara(0, 23));

    // Busca si la alarma ya esta programada
    uint8_t i = alarma_obtener(mensaje);
    if (i != ERROR) {
        // Reprograma la alarma
        if (retardo != 0) {
            alarma_crear(i, mensaje, periodico, retardo);
        }
        // Cancela la alarma
        else {
            alarma_eliminar(i);
            alarmas_activas--;
        }
    }
    // Si no esta y hay espacio para mas alarmas, la introduce
    else if (alarmas_activas < NUM_ALARMAS && retardo != 0) {
        alarma_crear(alarma_obtener(0), mensaje, periodico, retardo);
        alarmas_activas++;
    }
}

/*
Pre: Recibido evento T0_PERIODO
Post: Para todas las alarmas programadas, actualiza el tiempo hasta vencer
      la alarma. Si el tiempo a vencer es menor que el periodo
      transcurrido, encola el mensaje programado
*/
void alarma_refrescar(uint32_t periodo) {
    // Actualiza todas las alarmas activas
    for (uint8_t i = 0; i < NUM_ALARMAS; i++) if (alarmas[i].mensaje != MSG_INDEFINIDO)
    {

        // El alarma va a vencer, encola el mensaje
        if (alarmas[i].restante <= periodo) {
            cola_msg(alarmas[i].mensaje, 0);

            // Es periodica, reiniciar
            if (alarmas[i].periodico == 1)
                alarmas[i].restante = alarmas[i].retardo;
            // No es periodico, eliminar
            else {
                alarma_eliminar(i);
                alarmas_activas--;
            }
        }
        // El alarma no vence, actualiza cuenta
        else { alarmas[i].restante -= periodo; }
    }
}

```

Anexo A.5.2: G_Energia.c

```
/*
    Ordena al procesador a entrar en modo reposo (idle) en caso de no haber
    ninguna tarea por hacer en el planificador.

    El microprocesador detendrá su reloj manteniendo en funcionamiento el de
    los periféricos, que podran reanudar el procesador al interrumpir
*/
void energia_reposo(void) {
    power_idle();
}

/*
    Ordena al procesador a entrar en modo dormir (power-down) al recibir un
    mensaje Energia_Dormir.

    El microprocesador no ejecutará instrucciones ni procesará instrucciones.
    Se reactivará al recibir una interrupción externa EINT1 o EINT2;
*/
void energia_dormir(void){
    power_down();
}
```

Anexo A.5.3: G_IO.c

```
enum {
    PIN_JUGADOR_1    = 2,
    PIN_JUGADOR_2    = 3,
    PIN_COL1         = 3,
    PIN_REALIZADA     = 16,
    PIN_VALIDA       = 17,
    PIN_VICTORIA      = 18,
    PIN_OVERFLOW      = 30,
    PIN_LATIDO        = 31,
};

/*
    Pre: ---
    Post: Inicializa el gestor de entrada-salida, incluyendo el GPIO
*/
void IO_iniciar() {
    GPIO_iniciar();
    GPIO_marcar_salida(PIN_JUGADOR_1, 2);
    GPIO_marcar_salida(PIN_REALIZADA, 3);
    GPIO_marcar_salida(30, 2);
}

/*
    Pre: ---
```

```

    Post: Altera el estado de GPIO para una nueva partida
*/
void IO_nueva_partida() {
    GPIO_escribir(PIN_JUGADOR_1, 2, 0x1); // Indica que inicia jugador 1
    GPIO_escribir(PIN_REALIZADA, 3, 0x0); // Limpia estado partida
}

/*
    Pre: ---
    Post: Si la entrada ha cambiado desde la última entrada, reinicia el
          alarma para dormirse. Devuelve la columna introducida o error
*/
uint8_t IO_leer_entrada() {
    return GPIO_leer(PIN_COL1, 7);
}

// Alterna el estado del latido
void IO_cambiar_latido() {
    if (GPIO_leer(PIN_LATIDO, 1))
        GPIO_escribir(PIN_LATIDO, 1, 0);
    else GPIO_escribir(PIN_LATIDO, 1, 1);
}

// Apaga el indicador de latido
void IO_bajar_latido() {
    GPIO_escribir(PIN_LATIDO, 1, 0);
}

// Apaga el indicador de jugada inválida
void IO_indicar_jugada_valida() {
    GPIO_escribir(PIN_VALIDA, 1, 0);
}

// Enciende el indicador de jugada inválida
void IO_indicar_jugada_invalida() {
    GPIO_escribir(PIN_VALIDA, 1, 1);
}

// Alterna jugador y marca jugada realizada
void IO_jugada_realizada() {
    if (GPIO_leer(PIN_JUGADOR_1, 2) == 0x1) GPIO_escribir(PIN_JUGADOR_1, 2, 2);
    else GPIO_escribir(PIN_JUGADOR_1, 2, 1);

    GPIO_escribir(PIN_REALIZADA, 1, 1);
}

// Apaga el indicador de jugada realizada
void IO_bajar_jugada_realizada() {
    GPIO_escribir(PIN_REALIZADA, 1, 0);
}

// Indica estado de victoria
void IO_victoria() {
    GPIO_escribir(PIN_REALIZADA, 1, 1);
    GPIO_escribir(PIN_VALIDA, 1, 1);
}

```

```

// Indica estado de empate
void IO_empate() {
    GPIO_escribir(PIN_JUGADOR_1, 2, 3);
    GPIO_escribir(PIN_REALIZADA, 1, 1);
    GPIO_escribir(PIN_VICTORIA, 1, 1);
}

// Enciende el indicador de desborde
void IO_marcar_overflow() {
    GPIO_escribir(PIN_OVERFLOW, 1, 1);
}

```

Anexo A.5.4: G_Boton.c

```

/*
    Pre: ---
    Post: Inicializa el gestor de botones, incluyendo las interrupciones
          externas EINT1 y EINT2
*/
void boton_iniciar() {
    eint1_iniciar();
    eint2_iniciar();
}

/*
    Pre: Recibido evento BOTON1
    Post: Encola una petición para programar una alarma periódica que compruebe
          si el botón 1 está pulsado, y encola un mensaje para realizar una
          jugada
*/
void boton1_pulsado() {
    cola_msg(SET_ALARMA, ALARM_CHK_BOTON1_ON_MSG);
}

/*
    Pre: Recibido mensaje CHK_BOTON1
    Post: Si el botón 1 no está pulsado, rehabilita interrupciones y
          encola una alarma para detener las comprobaciones
*/
void boton1_comprobar() {
    if (!eint1_leer()) {
        cola_msg(SET_ALARMA, ALARM_CHK_BOTON1_OFF_MSG);
        eint1_habilitar();
    }
}

/*
    Pre: Recibido evento BOTON2
    Post: Encola una petición para programar una alarma periódica que compruebe
          si el botón 2 está pulsado, y encola un mensaje para iniciar la
          partida
*/

```



```

*/
void boton2_pulsado() {
    cola_msg(SET_ALARMA, ALARM_CHK_BOTON2_ON_MSG);
}

/*
Pre: Recibido mensaje CHK_BOTON2
Post: Si el botón 2 no está pulsado, rehabilita interrupciones y
      encola una alarma para detener las comprobaciones
*/
void boton2_comprobar() {
    if (!eint2_leer()) {
        cola_msg(SET_ALARMA, ALARM_CHK_BOTON2_OFF_MSG);
        eint2_habilitar();
    }
}

```

Anexo A.5.5: G_Serie.c

```

enum{
    BUFSIZE = 3
};

/*****
/* ENTERO <---> CARACTER */

/*
Convierte un dígito a caracter

i: Entero a convertir
*/
char serie_itoa(uint8_t i) { return i + '0'; }

/*
Convierte a caracter y envía a la línea serie un entero.

num: Entero a convertir
*/
void serie_itoa_wide(uint32_t num) {
    char buffer[32] = { 0 };

    uint8_t index = 31;
    uint32_t aux = num;

    do {
        buffer[index--] = serie_itoa(aux % 10);
    } while ((aux /= 10) > 0);

    index++;

    for (uint8_t i = 0; index < 32; i++) {
        buffer[i] = buffer[index];
    }
}

```

```

        buffer[index++] = '\0';
    }

    uart0_enviar_array(buffer);
}

/*
    Convierte un caracter a entero

    i: Caracter a convertir. Debe ser un caracter entre '0' y '9'
*/
uint8_t serie_atoi(char i) { return i - '0'; }

/*****
/* EXPORTADO */

/*
    Inicializa la línea serie
*/
void serie_iniciar() {
    uart0_init();
}

/*
    Interpreta el caracter obtenido de la línea serie

    data: Caracter obtenido
*/
void serie_leer(uint32_t data) {
    static uint8_t buffer[BUFSIZE] = {0};
    static uint8_t index = 0;
    static uint8_t esperando_delim = 1;

    uint8_t ch = data & 0xff;

    uart0_echo(ch);

    if (ch == '#') {
        esperando_delim = 0;
        for (int i = 0; i < BUFSIZE; i++) {
            buffer[i] = 0;
        }
        index = 0;
    }

    else if (!esperando_delim) {
        if (ch == '!') {
            uart0_enviar_array("\n\n");
            if (buffer[0] == 'E' && buffer[1] == 'N' && buffer[2] == 'D') {
                cola_msg(ACABAR, 3);
            }
            else if (buffer[0] == 'N' && buffer[1] == 'E' && buffer[2] == 'W') {
                cola_msg(INICIAR, 0);
            }
            else if ('1' <= buffer[0] && buffer[0] <= '7' && buffer[1] == 0 && buffer[2]
== 0) {

```

```

        cola_msg(CHK_ENTRADA, serie_atoi(buffer[0]));
    }
    else {
        serie_print("Comando incorrecto\n");
    }

    esperando_delim = 1;
}
else if (index < 3) {
    buffer[index] = ch;
    index++;
}
else {
    esperando_delim = 1;
}
}
}

/*
    Escribe el siguiente caracter del búfer de la línea serie
*/
void serie_escribir() {
    uart0_continuar_envio();
}

/*
    Genera un mensaje de bienvenida al iniciar
*/
void serie_pantalla_bienvenida() {
    serie_print("\nBienvenido a Conecta4.\n");
    serie_print("Para jugar, pulse cualquier boton o escriba #NEW!");
    serie_print("Para seleccionar una columna, escriba #C!, donde C es un numero de 1 a 7.");
    serie_print("Tienes 1 segundo para cancelar la jugada. Pulse el boton 1 para cancelarla.");
    serie_print("Pulse el boton 2 o escriba #END! para reiniciar la partida\n");
}

/*
    Genera un mensaje final.

    t_juego: Tiempo tomado en ejecutar el programa
    t_medio: Tiempo medio de procesamiento de mensajes
*/
void serie_mensaje_reinicio(uint32_t t_juego, uint32_t t_medio) {
    uart0_enviar_array("La partida ha durado ");
    serie_itoa_wide(t_juego);
    serie_print(" segundos");
    uart0_enviar_array("Se ha tardado ");
    serie_itoa_wide(t_medio);
    serie_print(" microsegundos de media para procesar un mensaje");
    serie_print("\nPara jugar, pulse cualquier boton o escriba #NEW!");
}

/*
    Envía una cadena cualquiera a la línea serie.

```

```

    linea: Cadena a enviar. Debe acabar en NULL

    Se añade un salto de línea a la línea añadida
*/
void serie_print(char* linea) {
    uart0_enviar_array(linea);
    uart0_enviar_array("\n");
}

```

Anexo A.5.6: alarmas.h

```

enum {
    PERIODICO      = (0x1 << 23),
    NO_PERIODICO   = (0x0 << 23)
};

// Mensajes predefinidos para alarmas
// Formato: MSG (31, 24) | PERIODICO (23) | MS (22, 0)
enum {
    // Comprueba botones cada 10ms
    ALARM_CHK_BOTON1_ON_MSG    = (CHK_BOTON1 << 24) | PERIODICO | 10,
    ALARM_CHK_BOTON2_ON_MSG    = (CHK_BOTON2 << 24) | PERIODICO | 10,

    // Apaga alarmas para botones
    ALARM_CHK_BOTON1_OFF_MSG   = (CHK_BOTON1 << 24) | NO_PERIODICO | 0,
    ALARM_CHK_BOTON2_OFF_MSG   = (CHK_BOTON2 << 24) | NO_PERIODICO | 0,

    ALARM_SIG_ON_MSG           = (SIGUIENTE << 24) | NO_PERIODICO | 1000,
    ALARM_SIG_OFF_MSG          = (SIGUIENTE << 24) | NO_PERIODICO | 0,

    // Alterna estado latido cada 250ms (parpadea cada 0,5 segundos)
    ALARM_PARPADear_MSG        = (PARPADEAR << 24) | PERIODICO | 250,

    // Apaga estado de jugada realizado tras 2 segundos
    ALARM_OFF_REALIZAR_MSG      = (OFF_REALIZAR << 24) | NO_PERIODICO | 2000,

    // Ordena dormir procesador tras 10s
    ALARM_GO_SLEEP_MSG          = (GO_SLEEP << 24) | NO_PERIODICO | 10000,

    // Ordena realimentar el watchdog cada 100ms
    ALARM_FEED_WD_MSG           = (FEED_WD << 24) | PERIODICO | 100
};

```

Anexo B: Ficheros adjuntos

Este documento (p2-3_memoria_778043-Latre_Villacampa_817570-Wozniak.pdf) es parte de la entrega de 2 ficheros que se piden.

Junto a la memoria, se deberá haber entregado un .zip con en proyecto completo (p2-3_proyecto_778043-Latre_Villacampa_817570-Wozniak.zip), que contendrá la totalidad de ficheros del proyecto, que debe funcionar sin ningún tipo de error.