MEMORIA PRÁCTICAS 2 Y 3

Reversi8 en una placa con procesador ARM

Fernando Peña Bes, 756012
Pedro José Pérez García, 756642

Proyecto Hardware

Universidad de Zaragoza, 20 de enero de 2020

1. Resumen ejecutivo

El objetivo de estas dos prácticas se centraba en los periféricos y su gestión, y otra en el juego final y la lógica de este.

En la Práctica 2, los esfuerzos se han centrado en gestionar la entrada y salida de la placa de forma que se pudieran procesar las entradas a través de los botones, y se respondiera al usuario de alguna forma, que en este caso ha sido a través del 8-led de la placa.

En concreto, se pedía implementar una cola de depuración donde los eventos de los diferentes periféricos se iban encolando (por lo general encolan un evento cada vez que interrumpen o cada cierto número de interrupciones), que además alojara también el momento del tiempo donde se han encolado los diferentes eventos. Para esto se ha reservado el `timer2`, que se leerá cada vez que se encole un evento y no ha cambiado desde la primera práctica. Estos eventos son desencolados y procesados por un bucle infinito, en la función `reversi_main()`.

Los siguientes pasos requerían implementar un latido en el led izquierdo de la placa, un parpadeo con una frecuencia fijada a 4Hz, que se ha logrado mediante el `timer0`. Posteriormente, se ha desarrollado una serie de funciones, alojadas en `button.c` y `button.h` que permiten controlar los pulsadores de la placa. Como el objetivo principal era la interacción entre 8-led y botones, un evento de pulsación de los botones aumenta o decrementa el número o símbolo mostrado en este, a través de las funciones de `8led.c` y `8led.h`. Como los pulsadores presentan rebotes en la señal eléctrica que mandan a la hora de generar interrupciones y generan muchas más interrupciones de las debidas, también se ha implementado el autómata de estados `botones_antirebotes` que se encarga de filtrar estos rebotes.

Además, se ha desarrollado un mecanismo de gestión de excepciones que muestra qué excepción se ha producido en el 8-led, parpadeando con el código de esta.

Para la práctica 3, el objetivo final era terminar el juego y ser capaces de ejecutarlo en la placa, ya flasheada. La mayoría del trabajo ha consistido en modificar lo ya creado en la práctica 2, como el comportamiento de los botones. Por ello, ahora permiten elegir la fila y la columna donde se quiere efectuar el movimiento del jugador. Para la gestión de la lógica del juego reversi se ha creado el autómata `jugada_por_botones`, y se han creado funciones para gestionar la pantalla táctil, ya que la jugada se confirma con tocar en el touchpad. Esto implicaba desarrollar otro autómata sencillo que filtrara posibles rebotes en la pantalla.

El autómata del juego especifica que se parte de una pantalla donde se muestran las reglas básicas, y al tocar la pantalla se empieza a jugar en el tablero al reversi, mostrando además información de tiempos de juego y procesamiento varios, como el tiempo total de partida en segundos, o el tiempo total empleado en cálculos, en microsegundos. Cuando la partida termina, se muestra una pequeña pantalla informando del resultado final, y si se vuelve a pulsar, la partida se reinicia. Esto implica que también se ha implementado una serie de funciones que permiten mostrar, borrar y modificar los elementos en pantalla con facilidad. Se consigue a través de transferencias de DMA, cuya gestión se ha tenido en cuenta en el autómata de la lógica del juego.

Por último, se ha hecho que la placa cambie a modo usuario antes de ejecutar el juego, y el `timer2` ahora funciona mediante FIQ ya que necesita algo más de prioridad que el resto de periféricos. Para jugar al juego final, se ha flasheado el programa compilado con -O3, de forma que ya no se precisa cargarlo cada vez.

En general, el resultado final es muy satisfactorio ya que se ha logrado cumplir en el tiempo establecido con todas las especificaciones requeridas, incluso haciendo apartados opcionales que se detallarán en posteriores secciones de esta memoria. También se ha logrado un alto grado de entendimiento del funcionamiento de la placa de desarrollo, de los modos de usuario y de la gestión de eventos, interrupciones y excepciones.

Índice

2. Introducción

Las prácticas siguen con la línea de las anteriores, donde se debe proseguir con el desarrollo del juego Reversi en la placa Embest S3CEDV40. En esta ocasión se debían preparar los periféricos, para posteriormente acoplarlos a la gestión del juego.

La primera práctica de estas dos se centraba en los periféricos y la creación de una estructura y un gestor que se encarguen de controlar que estos son correctamente atendidos, mientras que la segunda consistía en acoplar esa gestión a una lógica de juego, y a la parte visual por pantalla, para jugar enteramente en la placa.

2.1 Entorno de trabajo

Se ha utilizado Eclipse, junto con las herramientas gcc de compilación cruzada. La placa (Embest S3CEV40) se ha utilizado con un soporte especial a través del puerto JTAG que permite la ejecución paso a paso y acceso en tiempo real al estado del procesador y memoria.

Para poder depurar el código fuera del laboratorio, instalamos el entorno en nuestros ordenadores personales y usamos el plug-in de eclipse para simular procesadores ARM7TDMI.

3. Objetivos

La finalidad de estas prácticas es múltiple, ya que por un lado se trata de comprender cómo trabaja un procesador ARM con los diferentes modos de ejecución, sus particularidades y las excepciones que desencadenan que se cambie a uno o a otro, junto con el tratamiento de estas.

Y por otro lado se pretende aprender a gestionar la entrada y salida de los dispositivos conectados a la placa, teniendo en cuenta que esta gestión se tiene que llevar de forma paralela y concurrente entre varios periféricos, por lo que también es importante desarrollar habilidad para depurar el código y para configurarlos bien, ya que editar mal los valores de un registro de depuración puede llevar a perder tiempo valioso en depurar. Como son prácticas distintas y se pide cambiar el código y las funcionalidades de la placa entre una y otra, se tiene que lograr desarrollar un código modular y correctamente comentado, para poder cambiar cosas concretas rápidamente sin afectar demasiado al funcionamiento del resto del sistema.

También se pretendía aprender a utilizar la pantalla como periférico de salida, haciendo hincapié en las transmisiones de DMA y entendiendo cómo es necesario respetar los tiempos que conlleva una transmisión de este tipo, por lo que se volvía necesario buscar estrategias de ahorro, o de economizar el número y el tamaño de estas.

Adicionalmente, por la propia naturaleza de la asignatura, en todo momento hay que saber gestionar bien el tiempo, llevando correctamente un control de las tareas hechas y pendientes de hacer, y aprovechando las sesiones de laboratorio con la placa lo máximo posible. Esta gestión del tiempo es esencial para poder haber llegado a las fechas de entrega establecidas.

4. Estructura del proyecto

El proyecto final está formado por una serie de módulos enlazados entre sí. En el siguiente esquema se muestran todos ellos, ordenados por nivel de abstracción. A continuación del este, se incluye una descripción de cada uno junto a los ficheros que los forman.

Main	Reversi main					
Reversi 8						
Jugada por botones						
Cola de depuración	Rutinas excepciones					
Botones anitrebores	Tp antirebotes	Teclado antirebotes	Bitmap	Elementos pantalla		
8led	Lcd	Button	Timer	Tp	Keyboard	Led

- **Main** (`main.c`). Contiene la inicialización de los dispositivos, el cambio a modo usuario y la llamada a Reversi main para iniciar el juego.
- **Reversi main** (`reversi_main.c` y `reversi_main.h`). Contiene la inicialización del juego y el bucle principal del juego que atiende a eventos producidos por los temporizadores o por la interacción del usuario.
- **Reversi 8** (`reversi8_2019.c` y `reversi8_2019.h`). Implementa la lógica y almacena el estado del juego.
- **Jugada por botones** (`jugada_por_botones.c` y `jugada_por_botones.h`). Gestiona la lógica de la interacción el juego.
- **Cola de depuración** (`cola_depuracion.c`, `cola_depuracion.h`, `codigos_eventos.h`). Implementa la cola circular que se utiliza para guardar los diferentes eventos que llegan de los periféricos.
- **Rutinas excepciones.** (`rutinas_excepciones.c` y `rutinas_excepciones.h`). Incluyen las rutinas de tratamiento de las excepciones `DAbort`, `Undefined` y `SWI`.
- **Botones antirebotes** (`botones_antirebores.c` y `botones_antirebotes.h`). Implementa la máquina de estados que elimina los rebotes de los botones.
- **Tp antirebotes** (`tsp_antirebotes.c` y `tsp_antirebotes.h`). Implementa la máquina de estados que elimina los rebotes del touchpad.
- **Bitmap** (`Bmp.c` y `Bmp.h`). Contiene funciones para mostrar bitmaps en la pantalla LCD.
- **Elementos pantalla** (`elementos_pantalla.c` y `elementos_pantalla.h`). Contiene una colección de funciones que permiten dibujar en la pantalla LCD los elementos del juego.
- **Teclado antirebotes** (`teclado_antirebores.c` y `teclado_antirebotes.h`). Implementa la máquina de estados que elimina los rebotes del teclado.
- **8led** (`8led.c` y `8led.h`). Contiene las funciones de control del display de 8 segmentos.
- **Lcd** (`lcd.c` y `lcd.h`). Contiene las funciones de control de la pantalla LCD.
- **Button** (`button.c` y `button.h`). Contiene las funciones de manejo de los pulsadores.
- **Timer** (`timer.c`, `timer.h`, `timer2.c` y `timer2.h`). Contiene las funciones para el control de `timer1` y `timer2`.
- **Tp** (`tp.c` y `tp.h`). Contiene las funciones de control de la pantalla táctil.
- **Keyboard** (`keyboard.c` y `keyboard.h`). Contiene las funciones de control del teclado.
- **Led** (`led.c` y `led.h`). Contiene las funciones de control de los LED de la placa.

En el directorio `common` se incluyen todos los ficheros necesarios para la compilación e inicialización de la placa. Dentro de ese directorio se encuentra `44binit_flashear.asm`, que contiene el fichero `44binit.asm` modificado para poder flashear el programa en la placa.

5. Metodología Práctica 2

Se van a comentar y discutir aspectos clave del desarrollo del proyecto, como la estructura de los ficheros que lo componen, la evolución de estos con las diferentes prácticas, y el procedimiento de diseño seguido a lo largo de las sesiones, las decisiones tomadas, su implementación, los problemas encontrados y las soluciones que se les han dado.

Pasos, diseño e implementación realizados en la práctica 2

Ahora se van a describir las decisiones de diseño, junto con su justificación y fundamentos teóricos en los que estas se puedan basar. De ahí se pasará a concretar la implementación en código C y Ensamblador ARM en algunos casos, junto con los problemas encontrados y las soluciones que se les han dado.

Se van a explicar uno a uno y de forma ordenada los pasos solicitados por los profesores en el enunciado de las prácticas, junto con los apartados opcionales realizados. Se comenzará por la práctica 2, en orden cronológico, y posteriormente la 3, con las partes nuevas y también aquellas que hayan sufrido cambios.

5.1 Primer paso: Modificaciones al código de la primera práctica

Lo primero que se hizo nada más recibir retroalimentación de los profesores del código de la primera entrega, fue arreglar y modificar aquellos fragmentos del código que se hubieran señalado como problemáticos.

Solo se detectó un problema menor: el primero se encontraba en el código del timer2, ya que en `timer2_leer()` se podía leer un valor de tiempo erróneo. Esto se debe a que primero se leen las interrupciones que ha hecho el timer y se multiplican por el número de microsegundos que pasan en este lapso, 2048, y a ese valor se le suma la cuenta del registro interno del timer, para obtener la mayor precisión posible.

De esta forma, en los ciclos de procesador que se dan entre la lectura del número de interrupciones y la del número de ticks se puede incrementar el número de interrupciones, y por tanto se reinicia el de ticks en el registro interno, dando lugar a incongruencias. Por ejemplo, podemos leer una interrupción y 64 ticks, que serían 1 * 2048 + 64 / 32 = 2050 mirosegundos, sin saber que el valor real de ticks era, por ejemplo, 120, porque mientras se leía la variable del número de interrupciones el número de ticks ha llegado al máximo y el timer ha interrumpido, teniendo como valor real 2 interrupciones en vez de una, y los 64 ticks leídos.

La solución es simple, ya que cuando se tienen leídas las variables se comprueba que no haya cambiado el número de interrupciones en ese periodo de tiempo. De haber cambiado, simplemente nos quedamos con los valores más actualizados. El código modificado se encuentra a continuación.

```
unsigned int timer2_leer(void)
{
    unsigned int num_int_1 = timer2_num_int;
    unsigned int num_int_2 = timer2_num_int;
    // Con esto, evitamos posibles incrementos no deseados en timer2_num_int
    if (num_int_2 > num_int_1)
    {
        return num_int_2 * PERIOD_INT + (rTCNTB2 - rTCNTO2) / CYCLES_EACH_MICROSEC;
        // Si queremos optimizar, como la multiplicación es por 2048, se pueden mover
        // los bits 16 lugares a la izquierda y en la división, al ser por 32, se
        // pueden mover 5 a la derecha.
    }
    else
    {
        return num_int_1 * PERIOD_INT + (rTCNTB2 - rTCNTO2) /  CYCLES_EACH_MICROSEC
    }
}
```

Código 1 - Función `timer2_leer()` modificada

5.2 Tratamiento de excepciones

La primera funcionalidad a implementar como parte de la práctica 2 fue el correcto tratamiento de las diferentes excepciones que se pueden dar durante la ejecución. Se requería tratar diferentes tipos, como las Software Interrupt (SWI), Instrucciones indefinidas (Undef) y Data Aborts (Daborts). En caso de darse una de estas excepciones, se debía detener la ejecución normal y el 8-led de la placa debía parpadear, informando del código de la excepción. Además se tenía que indicar de alguna manera el tipo y la instrucción que ha causado la excepción.

Las excepciones se producen cuando el procesador detecta algo anormal durante la ejecución de un programa, como por ejemplo un acceso a memoria no alineado. Hay diferentes tipos de excepciones ya que interesa diferenciar las causas, y darles diferentes tratamientos, por eso cada excepción se ejecuta en un modo diferente, y tienen prioridades entre sí. Por ejemplo, las SWI tienen la menor prioridad dado que no se dan de forma natural y deben ser provocadas por el programador con su instrucción correspondiente.

Ante una excepción, el procesador ARM debe preservar el estado completo del procesador {R0-R14}, y guarda la dirección de retorno en el LR del registro del modo de la excepción, y hace igual con la palabra de estado y el SPSR del modo de la excepción. Desactiva las IRQ, y dependiendo del tipo de

Prioridad	Excepción	Modo	Vector
1	Reset	SVC	0x00
2	Data Abort	Abort	0x10
3	FIQ	FIQ	0x1C
4	IRQ	IRQ	0x18
6	Prefetch Abort	Abort	0x0C
7	Instruccion no definida	Undef	0x04
8	SWI	SVC	0x08

Tabla 1 - Excepciones del procesador ARM

excepción, las FIQ. Por último, como las excepciones son autovectorizadas, salta el PC a la dirección indicada en la componente correspondiente del vector. Como se indica en la tabla 1, si tenemos una excepción SWI, el programa saltará a la dirección alojada en la dirección 0x08 del vector (0xc7fff00 + 0x08)

Con esto en mente, el diseño del tratamiento de excepciones se facilita, ya que lo único que hay que hacer es modificar el vector de excepciones de SWI, Data Abort y Undefined de forma que apunten a su función de tratamiento.

Como el comportamiento que se buscaba era similar entre las diferentes excepciones, en vez de definir tres funciones de tratamiento que hicieran prácticamente lo mismo con pequeños cambios, se decidió diseñar una función común con pequeños cambios para cada excepción.

5.2.1 Implementación en lenguaje C

Para implementar todo esto en lenguaje C se han definido `rutinas_excepciones.c` y `rutinas_excepciones.h`. En estos ficheros se han definido las funciones `Gestion_excepciones_init()`, encargada de inicializar el vector de excepciones, haciendo que se salte a la función de gestión, `Gestion_excepciones()`. Esta función lee del CPSR para saber en qué modo de ejecución estamos, y así sabe qué excepción se ha dado, para poner en el 8-led un número que parpadea según la excepción. Como también se necesita saber el tipo de excepción y dónde se ha producido, se tienen dos variables llamadas `hay_excepcion` y `causa_fallo`, donde la primera almacena un valor de un enum que informa de que no hay, o del tipo de la que hay, y la segunda almacena el valor de LR del modo de excepción, al que se resta en función del modo, ya que la dirección real varía de modo a modo. El código se muestra a continuación:

```
/*--- variables globales del módulo (hacen falta?)---*/
volatile static int hay_excepcion;
volatile uint32_t causa_fallo;
/*--- Declaraciones de las diferentes rutinas para tratamiento de excepciones ---*/
volatile void Gestion_excepciones(void) __attribute__((interrupt("ABORT")));
volatile void Gestion_excepciones(void) __attribute__((interrupt("SWI")));
volatile void Gestion_excepciones(void) __attribute__((interrupt("UNDEF")));

//Ignorar warning, sí que devuelve lo que debe, pero sin hacer return en C como tal
volatile uint32_t __get_CPSR()                                    //Devuelve CPSR
        {
                __asm volatile ("MRS r0, CPSR");
                __asm volatile ("bx lr");
```

```
        }

void __attribute__((optimize("O0"))) parpadear_error(int caracter)
//8-led se queda parpadeando con el código del error
{
        //según lo definido en el enum del .h
        while(1)
        {
                D8Led_symbol(caracter);
                Delay(2500);
                D8Led_symbol(16);
                Delay(1250);
        }
}

volatile void Gestion_excepciones(void)
{
        asm volatile(" mov %0,lr\n" : "=r" (causa_fallo));
        volatile uint32_t cpsr = __get_CPSR();
        if((cpsr & 0x0000001F) == 0x1b)              //Si Estamos en modo Undef
        {
                hay_excepcion = UNDEF;
                causa_fallo = causa_fallo - 4;
                parpadear_error(UNDEF);
        }
        else if((cpsr & 0x0000001F) == 0x13)         //Si Estamos en modo SVC de SWI
        {
                hay_excepcion = SWI;
                causa_fallo = causa_fallo - 4;
                parpadear_error(SWI);
        }
        else                                         //Estamos en modo ABORT
        {
                hay_excepcion = DABORT;
                causa_fallo = causa_fallo - 8;
                parpadear_error(DABORT);
        }
}

//Inicializa la gestión de las excepciones de los tipos UNDEF, SWi y DABORT

volatile void Gestion_excepciones_init(void)
{
        pISR_DABORT = (int) Gestion_excepciones;
        pISR_SWI = (int) Gestion_excepciones;
        pISR_UNDEF = (int) Gestion_excepciones;
        hay_excepcion = NO_EXCEPT;
}
```

Código 2 - Detalle de la gestión de excepciones

El enum que define los códigos, alojado en el .h es el siguiente:

```
//Valores posibles que puede tomar la variable compartida hay_excepcion
enum {NO_EXCEPT = 0, SWI = 1, DABORT = 2, UNDEF = 3};
```

Código 3 - Enum con los códigos de las excepciones

Por tanto, en el 8-led se mostrará un 1, un 2 o un 3 dependiendo de si es un SWI, un Data Abort o un Undef. Y las variables ya mencionadas, si se exploran con el visor de memoria de Eclipse, contienen la dirección de la instrucción que la causó. Se puede ver con el desensamblado de Eclipse.

Por último, al realizar pruebas con la optimización máxima, -O3, se observó que no parpadeaba el 8-led. Se descubrió que la función `Delay()`, hecha por los profesores, y usada para el parpadeo del 8-led se optimizaba y se eliminaba su ejecución. Por eso, tanto en esa función como en `parpadear_error()` se ha añadido la directiva `optimize("O0")`, como se puede observar en el Código 2. Esta directiva permite indicar al compilador que esas dos funciones no se deben optimizar.

Para probar las diferentes excepciones, simplemente se han usado unas pocas líneas que aparecen comentadas en la función Main:

```
/// Pruebas de excepciones ///
asm volatile ("SWI 0x55");       //lanzar SWI
asm volatile ("mov r3, #3");
asm volatile ("ldr r2,[r3]");          //Forzar DABORT
asm volatile (".word 0xe7f000f0\n");//Forzar UNDEF
```

Código 4 - Código para forzar y probar diferentes excepciones

La primera línea es la instrucción que lanza la excepción, en la segunda y tercera se intenta un acceso a memoria no alineado (0x3) para forzar el Data Abort, y en la tercera se utiliza una instrucción inexistente. Al probarlas, funcionan correctamente y el 8-led parpadea con el código como se espera.

5.3 Cola de depuración

En siguiente lugar, se debía implementar una cola en la que se pudieran introducir eventos como las interrupciones de los periféricos que debieran ser tratadas de alguna forma en el juego, ya que en la versión final se van a tener numerosos periféricos interrumpiendo, y la cola ayudará a depurar correctamente e identificar fallos.

Se pedía además que se gestionara como una lista circular, alojada al final del espacio de memoria, antes de las pilas de los diferentes modos de ejecución. En ella se van a introducir los códigos que identifiquen los eventos, así como el momento del tiempo en el que se han introducido esos eventos.

De cara a la implementación, lo primero ha sido revisar los ficheros de configuración de la carpeta `common` en busca de alguna dirección de memoria que indique dónde ubicar la cola. En el fichero `44binit.asm` de esta carpeta se encontró la dirección de la base de cada una de las pilas, mientras que en 44blib.c se menciona que el heap termina justo encima de las pilas.

Como no se va a utilizar el heap, ni sus funciones de gestión ya que no se va a tratar con memoria dinámica en ningún momento, es buen lugar para colocar la cola de depuración, a una distancia razonable de las pilas para prevenir desbordamientos y sobreescrituras.

```
.equ   UserStack,    _ISR_STARTADDRESS-0xf00          /* c7ff000 */
.equ   SVCStack,     _ISR_STARTADDRESS-0xf00+256      /* c7ff100 */
.equ   UndefStack,   _ISR_STARTADDRESS-0xf00+256*2    /* c7ff200 */
.equ   AbortStack,   _ISR_STARTADDRESS-0xf00+256*3    /* c7ff300 */
.equ   IRQStack,     _ISR_STARTADDRESS-0xf00+256*4    /* c7ff400 */
.equ   FIQStack,     _ISR_STARTADDRESS-0xf00+256*5    /* c7ff500 */
```

Código 5 - Direcciones de las pilas según `44binit.asm`

```
#define HEAPEND   ( ISR_STARTADDRESS-STACKSIZE-0x500) // =0xc7ff000//
```

Código 6 - Definición del límite del heap en 44blib.c

Como se ve en los fragmentos de código, el heap está en las direcciones inmediatamente anteriores a las pilas de los modos. Si se observa la documentación aportada por los profesores se concluye que el mapa de memoria de un programa cargado en la placa tendrá por tanto un aspecto similar al de la Figura 3.

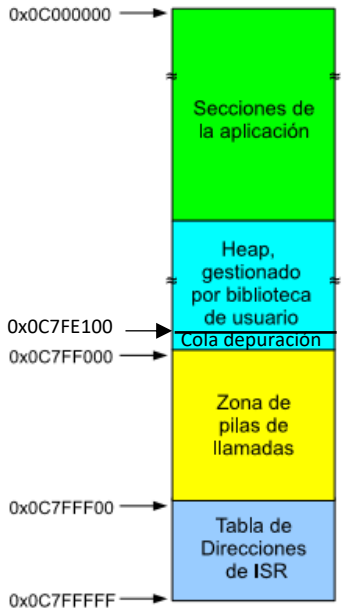

Figura 1 - Mapa de memoria, incluyendo la cola de depuración

La pila de depuración se decide colocar 4 veces más separada de lo que están las pilas entre sí, por precaución. Esto es, en la dirección **`0x0c7fe100`.** Así se asegura dejar suficiente espacio a las pilas, de forma que no vayan a sobrescribir esta cola.

5.3.1 Implementación en lenguaje C

A la hora de implementar esta estructura de datos se ha decidido que no se le iba a dar un tamaño fijo, puesto que podría interesar modificarlo, como a la hora de probar, donde interesa ver cuanto antes que no desborda por ningún lado, sin esperar a llenar una gran cantidad de elementos.

La función encargada de inicializar esta estructura es `cola_depuracion_inicializar()` y se le pasa como parámetro el tamaño que se le desea dar. Hay que tener en cuenta que, por ejemplo, si se le da tamaño 7, significa que caben 7 pares `<evento, timestamp>`, cada uno de 32 bits. Este tamaño asignado se guarda en la variable `maxAsignado`.

La cola se gestiona principalmente con dos punteros a entero de 32 bits, `cima` y `base`, que apuntan a los elementos en última y primera posición de la cola, respectivamente. Como particularidad, se tiene que cima en realidad no apunta al último, sino a la posición de justo encima, directamente donde habría que insertar el siguiente elemento. El número de elementos almacenados se cuenta con la variable `numElem`.

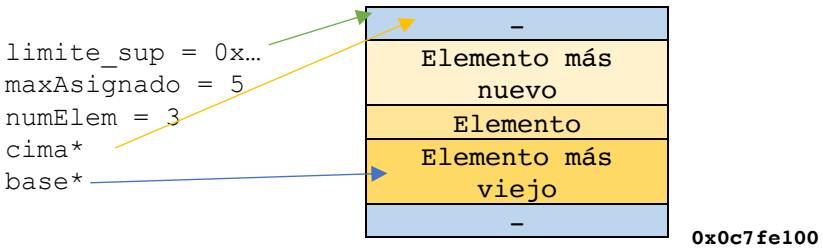

Figura 2 - Esquema de la cola de depuración

También se han desarrollado las funciones `push_debug()` y `pop_debug()`. Como su nombre indica, introducen y sacan elementos de la cola. Para introducir elementos se procesan sus dos parámetros, el entero de 8 bits correspondiente al código del evento y el entero de 32, con la información adicional del evento. La propia función se encarga de crear el entero a apilar, haciendo que el

evento sean los bits de mayor peso, y los 24 restantes son los de menor peso del campo de información de este, y apila también otro entero de 32 bits guarda la lectura del `timer2` que será su marca de tiempo. También gestiona los punteros de forma circular para evitar desbordamientos.

Esta función se llama en todas las rutinas de interrupción de periféricos, para notificar del evento al bucle principal del juego, que tratará los eventos. El propio timer2 es la excepción, ya que sobrecargaría la cola con información, y solo se usa como elemento auxiliar a la estructura.

(Posteriormente se modificará para que lea también de él al mostrar el profiling por pantalla al jugar)

Por su lado, pop_debug() hace lo necesario para separar el código del evento y su información auxiliar, y os devuelve. Se le pasan dos punteros cuyos valores asigna a los del evento e información extra. También gestiona los punteros de la cola adecuadamente para que nada falle.

```
static volatile uint32_t * cima;
static volatile uint32_t * base;
static volatile int limite_sup;
static volatile int numElem;
static volatile int maxAsignado;
//Cada elemento de la cola es una tupla <Evento,Timestamp> que ocupa 8 Bytes
//Por lo que se pasa como parámetro el número de tuplas que se desean almacenar.
void cola_depuracion_inicializar(int maxElem)
{
	cima = (uint32_t*)_COLA_DEP_STARTADDRESS;
	base = (uint32_t*)_COLA_DEP_STARTADDRESS;
	limite_sup = _COLA_DEP_STARTADDRESS - 8 * maxElem;
	numElem = 0;
	maxAsignado = maxElem;
	timer2_inicializar();
	timer2_empezar();
}
/* Devuelve, por separado, el ID del evento y la información adicional de este,
 * tal y como le fueron suministrados a push_debug */
void push_debug(uint8_t ID_evento, uint32_t auxData)
{
	uint32_t dato = (uint32_t) ID_evento << 24;
	auxData &= 0x00FFFFFF;
	dato |= auxData;
	if(numElem < maxAsignado)	//Si no hay que gestionar de forma circular
	{
		//encolar en cima y cima -=8 y numElem++
		if((int)cima <= limite_sup)
		{
			cima = (uint32_t *)_COLA_DEP_STARTADDRESS;
		}
		*cima = dato;
		cima -= 1;
		*cima = timer2_leer();
		cima -= 1;
		numElem += 1;
	}
	else
	{
		//base-=8, encolar en base vieja y si base se ha desbordado, corregimos
		cima = base;
		base -= 2;
		*cima = dato;
		cima -= 1;
		*cima = timer2_leer();
		cima -= 1;
		if((int)base <= limite_sup)
		{
			base = (uint32_t *)_COLA_DEP_STARTADDRESS;
		}
	}
}
//Subimos la base y restamos un elemento de la cola.
//Si desborda la base por el límite superior, la corregimos
void pop_debug(uint8_t *ID_evento, uint32_t  *auxData)
{
	if(!esVacia())
```

```
        {
                uint32_t * aux = base;
                base -= 2;
                if((int)base <= limite_sup)
                {
                        base = (uint32_t *)_COLA_DEP_STARTADDRESS;
                }
                numElem -= 1;
                uint32_t dato = *aux;
                *auxData = dato & 0x00FFFFFF;
                dato &= 0xFF000000;
                *ID_evento = dato >> 24;
        }
}
int esVacia(void)
{
        return (numElem == 0);
}
```

Código 7 - Funciones para la gestión de la cola de depuración

5.3.2 Códigos de los eventos

Para mantener el código lo más limpio posible, se han reunido todos los posibles eventos a tratar en un único fichero, llamado codigos_eventos.h, que consta de un enum con todos los códigos y los campos de información reunidos, de forma que el código utilice nombres como ev_bot_der y no 0xFF. Así, todas las funciones donde se necesite hacer referencia a un evento, se incluye este fichero, en vez de redefinir todo el rato, algo básico si se desea un código lo más modular posible.

```
/*--- declaracion los códigos en un enum ---*/

//Los códigos de 8 bits representan eventos y empiezan por ev_, y los de 24 represen-
tan info adicional
enum {
    ev_tick_timer0    = 0xFF,
    ev_button_int     = 0xBB,
    ev_tsp            = 0x11,
    ev_keyboard       = 0x22,
    /*ev_autoincr     = 0xCC,        //Ya no se usa al haber cambiado la forma en que
                                     //se comunican antirebotes y jugada_por_botones
    ev_fila           = 0x55,        //Se deja por si en el futuro se puede recuperar
    ev_columna        = 0x44,        //El motivo del cambio es que la cola es
                                     //concurrente, y se ha decidido no apilar eventos
    ev_ready          = 0x33,*/      //que no sean necesarios, para aliviar su carga
    ev_finLCD         = 0x69,
    tecla_0           = 0x00000070,
    tecla_1           = 0x00000071,
    no_info           = 0xAAAAAAAA,
    button_izq        = 0x0000000F,
    button_der        = 0x0000000E
        };
```

Código 8 - Códigos de los diferentes eventos a tratar, versión final de la práctica 3

El comentario en verde del código se ha dejado con la intención de hacer ver que originalmente se pretendían apilar más eventos pero dado que la cola puede ser concurrente, se ha decidido darle la mínima carga posible. En vez de usar el sistema de eventos, esos módulos se comunicarán directamente entre sí. Para más detalles acerca de la interacción entre los módulos y periféricos, se incluye un esquema en la Figura 12.

5.4 Integración de juego y `reversi_main()`

Con la cola de depuración terminada, el siguiente paso que se ha dado en la práctica 2 ha sido crear un bucle cuya función va a ser atender en orden todos los eventos generados por los periféricos, de forma ininterrumpida, por lo que es un bucle infinito. Se trata de la función **`reversi_main()`**,

que estará construida en los ficheros con igual nombre y extensiones .c y .h, y va a ser la encargada de comunicar la mayoría de los módulos entre sí.

Su funcionamiento viene explicado en el enunciado de la práctica y es sencillo de entender. Es un bucle que perpetuamente está comprobando si hay eventos que tratar en la cola de depuración, y de ser así, ejecuta un comportamiento concreto para cada evento hasta haber atendido todos los eventos que tenga pendientes. Por ejemplo, si llega un evento de que el `timer0` ha interrumpido, se debe ejecutar una función (entre otras) que se dedica a gestionar el latido del Led, y si hay que parpadear o no (el latido se explica con detalle en la siguiente sección, 4.5). Por el contrario, si no hay eventos pendientes de tratamiento y la cola está vacía, se ejecuta una función vacía para esperar a que llegue algún evento.

También se han implementado otras funciones en el módulo, como la de inicialización, encargada de poner en marcha todos los autómatas y controladores de los periféricos y la lógica del juego, así como de inicializar algunas variables necesarias.

5.4.1 Código en C

```
/*--- Código de funciones ---*/
void reversi_main_inicializar()
{
	//Inicializar las variables que hagan falta para proesar bien los eventos
	timer_init();
	cuenta_int_t_juego = 0;
	estado_led1 = 0;
	cuenta_int_latido = 0;		//Cada 7 u 8 hay que cambiar el led izquierdo
	led1_off();			//El led empieza apagado
	botones_antirebotes_inicializar();
	tsp_antirebotes_inicializar();
	init_keyboard();
	tec_antirebotes_inicializar();
	inicializar_jugada_botones();
}

			. . .

void reversi_main()
{
	reversi_main_inicializar();
	while(1)
	{
		while(!esVacia())
		{
			//Desencolar para poder procesar la información
			uint8_t evento;
			uint32_t info;
			pop_debug(&evento, &info);
			switch(evento)
			{
			case ev_tick_timer0 : //Atender eventos de timer0
				. . .
				break;
			case ev_button_int :  //Atender eventos de los botones
				if(info == button_izq)
				{	//Botón izquierdo
					. . .
				}
				else
				{	//Botón derecho
					. . .
				}
				break;
			case ev_finLCD:
				termina_DMA();
				break;
			case ev_tsp:
				. . .
				break;
			case ev_keyboard:
				. . .
```

```
                        break;
                    default : //Si es otra cosa desconocida, no lo atendemos
                        break;
                    }
                }
                dormir_procesador();
        }
}
```

Código 9 - Resumen del funcionamiento del bucle en `reversi_main()`

Se han recortado algunos detalles, ya que no aportan mucho. Si se desea ver todos los detalles de cómo se trata cada evento, se puede consultar en el código entregado.

En la zona de declaración de variables del código entregado se puede ver una explicación detallada del por qué de esas variables que aparentemente no aportan nada como `cuenta_int_t_juego`, junto con otras dos funciones que se han omitido aquí. En las secciones correspondientes al latido y al autómata de la lógica del juego se comentan y explican.

5.5 Latido en el led de la izquierda

Para tener alguna referencia visual fácil de identificar, se ha hecho que el led izquierdo de la placa parpadee con una frecuencia de 4Hz, entendiendo por parpadear hacer el ciclo completo de cambiar de encendido a apagado, y otra vez a encendido, o viceversa.

Era obligatorio implementar este comportamiento con el `timer0`, que debía interrumpir a razón de 60 veces por segundo. Por tanto, se va a explicar primero la configuración del timer0 y luego cómo se ha conseguido la frecuencia del latido a partir de las 60 interrupciones por segundo.

5.5.1 Recalibrar `timer0` a 60 interrupciones por segundo

Tal como ya se ha explicado, se debe conseguir que timer0 interrumpa 60 veces cada segundo. O lo que es lo mismo, una vez cada 16.6 milisegundos o 166 décimas de milisegundo. Se elige dar al temporizador una resolución de 1e-4 segundos, de forma que el registro de ticks internos deberá contar desde 166 hasta 0.

La fórmula donde se explica cómo configurar los registros internos del temporizador en función de la precisión que se desea obtener es la siguiente:

$$F = MCLK/\left((\text{valor de pre-escalado} + 1)(\text{valor del divisor})\right)$$

Fórmula 1 - Precisión del temporizador en función de los valores de sus registros

Si F = 10000 , es la frecuencia de ticks internos del timer y MCLK = 64e6, es la frecuencia del procesador (64 MHz), como divisor solo se pueden poner los valores 2, 4, 8, 16 y 32, y el preescalado debe ser entre 0 y 255, se toma 199 como valor de preescalado (hay que recordar que se le suma 1) y un divisor de 32.

Finalmente, se modifican los registros con los nuevos valores.

```
/* Configura el Timer0 a resolución de 1e-4 segundos para el latido */
rTCFG0 &= 0xffffff00;
rTCFG0 |= 0xc7;       // Ajusta el preescalado del timer 0 a 199 (200) para el latido
rTCFG1 &= 0xffffff0;
rTCFG1 |= 0xfffffff7;  // Selecciona la entrada del mux que proporciona el reloj 0.
                       // La 1xx (0x...7) corresponde a un divisor de 1/32.
rTCNTB0 = 166;         // valor inicial de cuenta. Con esto hay una interrupción cada
                       //  1/60 de segundo.
rTCMPB0 = 0;           // Valor de comparación
```

Código 10 - Valores de registros de configuración de `timer0`

Además, las interrupciones ahora solamente generan un evento que se encola. Se le dará tratamiento al ser atendida por `reversi_main()` que llamará a las funciones que hagan falta de los módulos necesarios.

```
void timer_ISR(void)
{
        push_debug(ev_tick_timer0, no_info);
        /* borrar bit en I_ISPC para desactivar la solicitud de interrupción*/
        rI_ISPC |= BIT_TIMER0;
}
```

Código 11 - Rutina de interrupción de `timer0`

5.5.2 Parpadeo del led izquierdo a 4Hz

Con el problema del timer ya solucionado, queda pasar de una resolución de 60Hz a una de 4. En realidad, hay que **cambiar el estado del led** con el doble de frecuencia por lo ya comentado de que 4 son las veces que hay que hacer los dos cambios, no uno. Entonces, 60/(4 x 2) nos da que cada 7.5 interrupciones del temporizador se debe cambiar de estado. En vez de coger un valor u otro y generar imprecisiones que se irían arrastrando, se ha optado por hacer un cambio cada 7 y otro cada 8.

Aquí entra en juego una de las funciones que no se habían comentado anteriormente en `reversi_main()`. Esta función es `Latido_ev_new_tick()`. Básicamente lleva una cuenta de cuántas veces ha interrumpido, y en función del número de veces que gestiona la variable `cuenta_int_latido`, y de si toca que sea cada 7 o cada 8 (se codifica junto con el estado del led en `estado_led1`), cambia el estado del led con las funciones proporcionadas en led.c y led.h. También reinicia la cuenta del número de interrupciones a 0 cada vez que cambia el estado.

El número de interrupciones necesarias para cambiar el led se ha decidido codificar en un enum bajo los nombres `ticks_latido_A` y `ticks_latido_B`

A continuación, se muestra la implementación concreta de lo ya descrito.

```
/*--- variables ---*/
static enum {ticks_latido_A = 7, ticks_latido_B = 8, ticks_segundo_de_juego = 60};
static int cuenta_int_latido;
static char estado_led1;

. . .

void Latido_ev_new_tick(void)
{
        cuenta_int_latido += 1;
        if(estado_led1 == 0)  //Unas veces 7 y otras 8, 7.5 de media
        {
                if(cuenta_int_latido == ticks_latido_A)
                {
                        estado_led1 = 1;
                        led1_on();
                        cuenta_int_latido = 0;
                }
        }
```

```
        else
        {
                if(cuenta_int_latido == ticks_latido_B)
                {
                        estado_led1 = 0;
                        led1_off();
                        cuenta_int_latido = 0;
                }
        }
}
```

Código 12 - Función para conseguir la correcta frecuencia de latido en el led

5.6 Comportamiento de los botones y el 8-led

En este apartado se va a describir cómo se ha logrado el comportamiento principal de la placa para la práctica 2. El 8-led de la placa debe mostrar números del 0 a la F en hexadecimal, que se irá cambiando según se pulsen los botones de la placa. Al principio, se muestra una 'F' de fila y con el botón izquierdo se aumenta el número deseado, que irá de 0 a 9, y si se pasa volverá a ser un 0. Al pulsar el botón derecho, aparecerá una 'C' de columna y se repetirá el mismo proceso para elegir un número en el 8-led, solo que al pulsar el botón derecho esta vez de reinicia el proceso mostrando la 'F' de fila.

En el siguiente apartado se darán más detalles acerca de cómo se ha logrado este comportamiento con un autómata de estados. Ahora se va a focalizar la atención en los botones y el 8-led como periféricos, y en el filtrado de los rebotes del 8-led con la ayuda de una máquina de estados y un temporizador.

5.6.1 Ficheros `button` y `8led`

Entre los ficheros fuentes iniciales se proporcionaron, unos ficheros que no se habían usado para nada hasta el momento: `button.h`, `button.c`, `8led.c` y `8led.h`.

También se proporcionan los ficheros `led.c` y `led.h` pero no hacía falta modificar nada en ellos.

En el caso de estos últimos, al estudiarlos junto con la documentación, se vio rápidamente que no era necesario arreglar nada, puesto que el 8led está controlado por hardware y únicamente se escribe en un registro mapeado en memoria el valor que corresponda con los segmentos (o punto decimal) que se quieren encender y apagar. Para encender se pone ese bit a 0 y para apagar, se pone a 1. Lo único que se modificó fue el añadido de un nuevo símbolo al enum de símbolos predefinidos, llamado 'blank', que apaga todo el 8-led, para hacerlo parpadear con más facilidad en el caso de la gestión de las excepciones, sin números mágicos 'colgando' por el código.

En el caso de los ficheros que controlan los pulsadores de la placa, sí que había cosas que cambiar. Solamente partiendo de la implementación que se nos daba, había que cambiar valores de registros para que solo afectaran a la línea de los botones (EINT4567), o cambiar la forma de asignación (por ejemplo, AND bit a bit en vez de una asignación directa de valor para no sobrescribir valores que no se deberían tocar)

Configuración de los registros de control de los botones

Para que los botones fueran utilizables había que cambiar los valores de la mayoría de las configuraciones del fichero inicial. Los botones van conectados a la línea de interrupción EINT4567, por lo que 4 dispositivos compartirán la línea (los pulsadores/botones son las líneas 6 y 7). Si se desea saber cuál de estos dispositivos ha generado la interrupción, se debe leer del registro `EXTINTPND`. Como los otros dos periféricos no interrumpen ni se utilizan en el proyecto, se puede comprobar si el valor leído es un 4 para la línea 6 , botón izquierdo, o un 8, línea 7, botón derecho.

La configuración completa de los registros correspondientes a los botones se lleva a cabo en la función `button_iniciar()` como se observa en el siguiente fragmento de código:

```
void button_iniciar(void)
{
        /* Configuracion del controlador de interrupciones pensando SOLO en usar bits
         * 6 y 7 para los pulsadores. Estos registros están definidos en 44b.h */

        rI_ISPC    |= BIT_EINT4567;   // Borra INTPND escribiendo 1s en I_ISPC
        rEXTINTPND = 0xf;            // Borra EXTINTPND con 1s en el propio registro
        rINTMOD    &= ~(BIT_EINT4567);// Configura la linea EINT4567 como de tipo IRQ
        rINTCON    &= 0x1;           // Habilita int. vectorizadas y linea IRQ(FIQ no)
        rINTMSK    &= ~(BIT_EINT4567); // habilitamos int. linea eint4567 en v.masc.

        /* Establece la rutina de servicio para Eint4567 */
        pISR_EINT4567 = (int) button_ISR;

        /* Configuracion del puerto G */
        rPCONG  |= 0xf000;           // Establece la funcion de los pines (EINT6-7)
        rPUPG   &= 0x3f;             // Habilita el "pull up" de los pines 6 y 7, de
                                     //  los pulsadores
        rEXTINT &= 0x00ffffff;       //
        rEXTINT |= 0x22000000;       // Configura las lineas de int. de los pulsadores
                                     // como de flanco de bajada

        /* Por precaucion, se vuelven a borrar los bits de INTPND y EXTINTPND */
        rEXTINTPND = 0xf;            // borra los bits en EXTINTPND
        rI_ISPC   |= BIT_EINT4567;   // borra el bit pendiente en INTPND
}
```

Código 13 - Configuración de los registros de los pulsadores

En los comentarios del código se detallan las operaciones realizadas, como, por ejemplo, que hace falta habilitar la resistencia “pull-up” de los pulsadores para el correcto funcionamiento de los botones.

Funciones para manejo de los botones y rutina de interrupción

Lo siguiente fue desarrollar las funciones especificadas en el enunciado, que permiten controlar y gestionar el tratamiento de las interrupciones de los botones, la gestión asociada a los eventos generados, y el filtrado de los rebotes. Las funciones a desarrollar, que interactúan con el hardware, fueron:

- **`enum estado_button {button_none, button_iz, button_dr}`**
  Estados que pueden tener los botones, enum definido en `button.h`
- **`button_iniciar()`**
  Ya comentada, responsable de inicializar los registros de configuración
- **`button_resetear()`**
  Permite readmitir interrupciones de la línea EINT4567, después de haber sido desactivada para el filtrado de los rebotes.
- **`enum estado_button button_estado()`**
  Devuelve el estado actual de los botones. En teoría ambos no pueden estar pulsados a la vez, pero si se da este caso, se da prioridad al derecho, por poner alguno y cubrir este caso.

Estas funciones fueron colocadas en `button.c`, ya que interactúan directamente con el hardware de la placa, a diferencia de las otras 2 que se comentarán a continuación. Se muestra más abajo su implementación en C, junto con la rutina de interrupción.

La rutina lee de `EXTINTPND` para saber qué botón se ha pulsado, y en función de ello, encola un evento `ev_button_int` donde lo que varía es el campo de información adicional, según el botón pulsado. La función `button_estado()` tiene un comportamiento similar, pero en vez de encolar un evento, devuelve un `enum estado_button`. Además, no lee de las interrupciones pendientes,

sino que lo hace de `PDATG`, el registro de datos del puerto G, donde también se puede ver si los botones están pulsados o no. De este registro de datos interesa leer los bits de mayor peso. Por último, `button_resetear()` borra las interrupciones pendientes, por precaución, y rehabilita las interrupciones de la línea, modificando la máscara de interrupciones. (Se deshabilitarán al tratar los rebotes, se explica en la siguiente sección)

```
void button_ISR(void) __attribute__((interrupt("IRQ")));
/*--- codigo de funciones ---*/
void button_ISR(void)
{
        rINTMSK  |= (BIT_EINT4567);   //Deshabilitamos interrupcion linea eint4567
        volatile int which_int = rEXTINTPND;
        rEXTINTPND |= 0xa;            // borra los bits 6 y 7 en EXTINTPND
        rI_ISPC   |= BIT_EINT4567;    // borra el bit pendiente en INTPND

        /* Identificar la interrupcion (hay dos pulsadores)*/
        switch(which_int)
                {
                        case 4:         //boton 6, izquierdo
                                push_debug(ev_button_int, button_izq);
                                break;
                        case 8:         //boton 7, derecho
                                push_debug(ev_button_int, button_der);
                                break;
                        default:
                                break;
                }
        /* Finalizar ISR */
        rEXTINTPND |= 0xa;            // borra los bits 6 y 7 en EXTINTPND
        rI_ISPC   |= BIT_EINT4567;    // borra el bit pendiente en INTPND
}
void button_resetear(void)            //Reactiva interrupciones y deja button listo
{
        /* Por precaucion, se vuelven a borrar los bits de INTPND y EXTINTPND */
        rEXTINTPND = 0xf;             // borra los bits en EXTINTPND
        rI_ISPC   |= BIT_EINT4567;    // borra el bit pendiente en INTPND
        rINTMSK    &= ~(BIT_EINT4567); // rehabilitamos interrupcion linea eint4567 en
}
/* Devuelve el estado de los botones */
// Se supone que nunca están los dos botones pulsados a la vez
enum estado_button button_estado(void)
{
        rPCONG  &= 0x0fff;
        int input_GPort = rPDATG;
        rPCONG  |= 0xf000;

        if ((input_GPort & 0x40) == 0) {
                return button_iz;
        }
        else if ((input_GPort & 0x80) == 0) {
                return button_dr;
        }
        else if ((input_GPort & 0xc0) == 0) {
                return button_dr;
                // Si los dos botones estuvieran a 1 (suponemos que no pasa) se
                //  devuelve que está pulsado el derecho
        }
        else {
                return button_none;
        }
}
```

Código 14 - Implementación de las funciones de button.c

5.6.2 Filtrado de rebotes en los pulsadores

Una vez se tiene la gestión del dispositivo hardware, aparece un nuevo problema: rebotes en la señal eléctrica. Esto se debe a la onda cuadrada del pulsador, donde la tensión oscila antes de estabilizarse al ser pulsado, y al ser soltado.

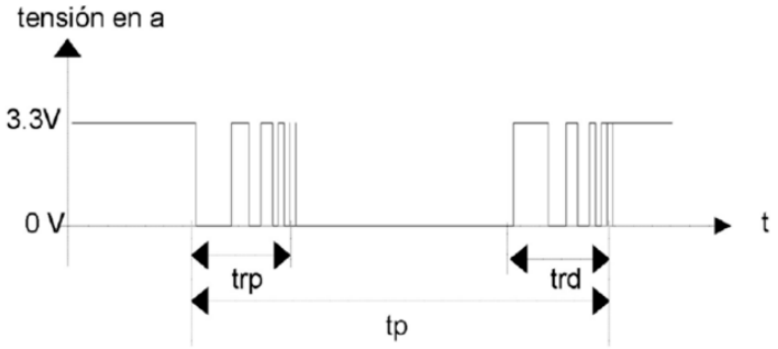

Figura 3 - Rebotes en la señal eléctrica de los pulsadores

Siguiendo el esquema planteado en el enunciado, se plantea un autómata de estados para tratar estos rebotes en la señal. El comportamiento a implementar es el descrito en el enunciado:

1. Se encola un evento, informando de qué botón ha sido pulsado
2. Se inhabilita la línea de interrupción de los botones
3. Se espera un tiempo trp, medido en ticks de un timer para filtrar los rebotes de entrada.
4. Pasado ese tiempo, se monitoriza cada 30ms para ver cuándo se suelta
5. Se espera otro tiempo trd, medido en ticks de un timer para filtrar los rebotes de salida.
6. Se rehabilitan interrupciones de los botones

Diseño del autómata de filtrado de rebotes y autoincremento

Siguiendo estas indicaciones, se diseña un autómata de estados que tenga en cuenta las funciones que se requiere implementar, con alguna ligera modificación del esquema planteado. Por ejemplo, se desactiva la línea de interrupciones del botón lo antes posible, nada más entrar en la rutina, en vez de esperar a tratar el evento encolado. Esto se puede observar en el código de la rutina en `button.c` (Código 13)

Pero, por lo general, se distinguen 4 estados, uno inicial, otro donde se espera el tiempo trp, otro donde se encuesta, y un último donde se espera el tiempo correspondiente a trd, antes de restablecer las interrupciones. El autómata final se muestra en la siguiente Figura. Si no se puede visualizar bien, junto con esta entrega se incluyen en un fichero pdf, donde se ven con mayor resolución.

Aprovechando este apartado, se incluye el diseño correspondiente al apartado opcional del autoincremento. El objetivo era conseguir que al mantener pulsado el botón izquierdo más de un tercio de segundo, se incrementara el valor del 8-led cada 180 milisegundos. Esto se puede lograr si se logra contar cuándo se mantiene pulsado el botón **izquierdo** durante un tercio de segundo, lo cual, active alguna variable que permita identificar cuándo pasar a contar intervalos de 180 milisegundos, y que esta otra variable de la cuenta actualice el valor del 8-led, y reinicie su cuenta de forma cíclica, hasta que se deje de pulsar. Por tanto, como se mostrará en el autómata y en el código, parecen casos a tener en cuenta dentro de los estados de cuenta de trp (empezar a contar 180ms), y de encuesta cada 30ms (el resto).

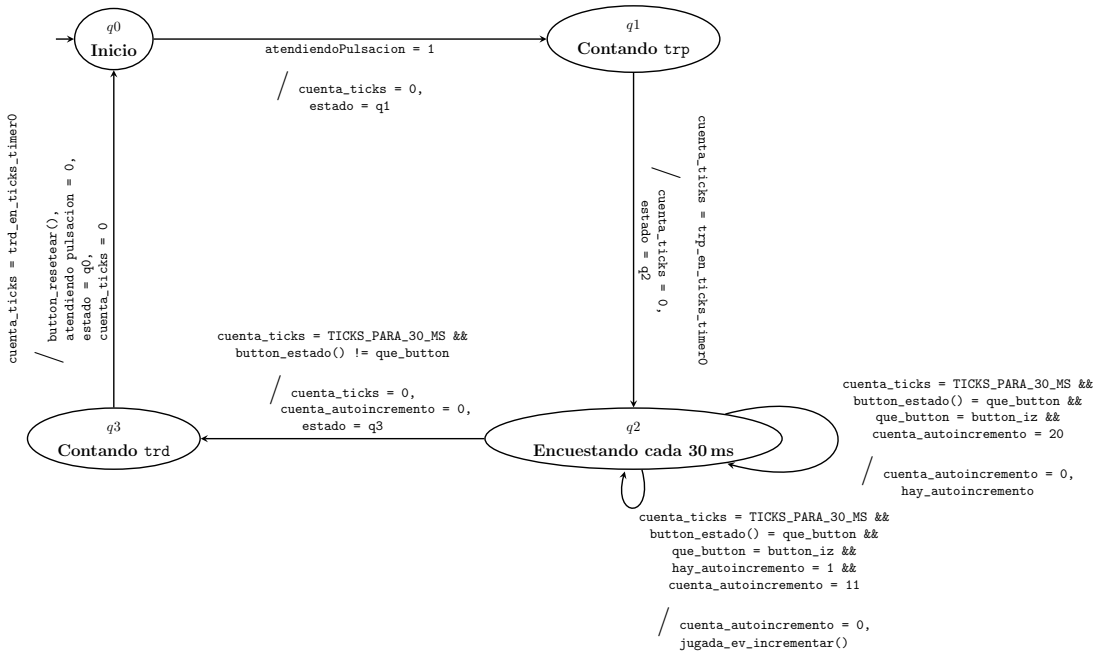

Figura 4 - Autómata de estados de `botones_antirebotes`, versión de la práctica 2

Implementación del autómata antirebotes en lenguaje C

Este autómata se implementa en el módulo `botones_antirebotes.c`. Se han tomado unas cuantas decisiones de cara a la implementación en C, como que haya una variable que actúa de guarda, `atendiendoPulsacion`, de forma que, si ya está atendiendo una pulsación, no se atenderá en el autómata otra, puesto que seguramente será algún rebote que se ha logrado 'colar' antes de desactivar las interrupciones. El estado se guarda en un enum que puede tener 4 valores, uno por cada estado del autómata, y las transiciones se implementan en una misma función, llamada `antirebotes()`, que en función del estado y las demás variables, como la cuenta de ticks del timer, decide qué acciones tomar y a qué estados saltar.

En el código en C se pueden ver también comentarios que ayudan a entender el comportamiento del autómata. Otra decisión tomada de cara a la implementación ha sido elegir con qué temporizador medir los tiempos trp, trd y los de control de los autoincrementos. Se ha decidido que sea timer0, puesto que ya estaba configurado, así no se sobrecarga con más eventos o interrupciones. Como timer0 interrumpe 60 veces por segundo, se necesitan 20 ticks para el tercio de segundo y unos 11 aproximadamente, para los 180 milisegundos que controlan los autoincrementos. Por defecto, se han establecido trp y trd en 4 ticks. Y los 30 milisegundos de la encuesta serán aproximadamente 11 ticks.

```
static enum {TICKS_PARA_30MS = 2,trp_en_ticks_timer0 = 4, trd_en_ticks_timer0 = 4,
                    int_timer0_enable_autoincr = 20, int_timer0_autoincr = 11};
/*Los valores de trp y trd se pueden cambiar en función de la placa para controlar
 * mejor los rebotes.
 *Si seguimos pulsando después de 1/3 de segundo (= 20 interrupciones de
 * timer0 * 1/60 * seg. = 1/3 seg.), hay que empezar a autoincrementar.
 * Por eso int_timer0_enable_autoincr = 20
 * Hay que incrementar cada 180ms, que en interrupciones de timer0 es
 * 180/16 ~= 11 <- int_timer0_autoincr.
*/
```

Código 15 - Tiempos del autómata en ticks de timer0

```
static enum estados{Inicio, contando_trp, encuestando, contando_trd} maquina_estados;
static int atendiendoPulsacion;
static int cuenta_ticks = 0;                    //Cuenta ticks de timer0 para gestionar
                                                //      los retardos de rebotes y de en
                                                //      cuesta al botón
static enum estado_button que_button;           //Para saber qué botón hemos pulsado y
                                                //      saber cuál mirar a la hora de com
                                                //      probar si se ha soltado
static int cuenta_autoincremento;               //Cuenta ticks de timer0 para gestionar
                                                //      el autoincremento (pulsado > 1/3s
                                                //      y el incr. cada 180ms)
static int hay_autoincremento;                  //Variable auxiliar de cuenta_autoincre
                                                //      mento que sirve para ver cuándo
                                                //      estamos contando 1/3s o 180ms.
                                                //      Puede valer 0 o 1.
void botones_antirebotes_inicializar()
{
        maquina_estados = Inicio;
        cuenta_autoincremento = 0;
        hay_autoincremento = 0;
        atendiendoPulsacion = 0;//Inicialmente no se está atendiendo ninguna pulsación
}

void antirebotes(void)
{
        switch(maquina_estados)
        {
                case Inicio :
                        if(atendiendoPulsacion)
                        {
                                cuenta_ticks = 0;
                                maquina_estados = contando_trp;
                        }
                        break;
                case contando_trp :
                        if(cuenta_ticks == trp_en_ticks_timer0)
                        {       //Volvemos a usar timer0 para encuestar cada 30ms a1 bo
                                //      tón, esperando que deje de estar oprimido
                                cuenta_ticks = 0;
                                maquina_estados = encuestando;
                        }
                        break;
                case encuestando:
                        if(cuenta_ticks >= TICKS_PARA_30MS)
                        {//Si es hora de encuestar al botón
                                if(button_estado() != que_button)
                                {        //y vemos que ya no está presionado
                                        cuenta_ticks = 0;
                                        cuenta_autoincremento = 0;
                                        hay_autoincremento = 0;
                                        maquina_estados = contando_trd;
                                }
                                else if(que_button == button_iz)
                                if(cuenta_autoincremento == int_timer0_enable_autoincr)
                                {       //Tras mantener pulsado durante 1/3 de segundo
                                        //  autoincremento cada 180ms a partir de ahora
                                        cuenta_autoincremento = 0;
                                        hay_autoincremento = 1;
                                        jugada_ev_incrementar();
                                }
                                if(hay_autoincremento && cuenta_autoincremento ==
                                                                        int_timer0_autoincr)
//11 ticks de timer0 son 182.6 milisegs ~= 180 ms, para incrementar de forma auto
                                {
                                        cuenta_autoincremento = 0;
                                        //push_debug(ev_autoincr, no_info);
                                        //Lo hago como una llamada directa a
                                        //jugada_por_botones porque la cola es
                                        //concurrente y así se usa el mínimo posible
                                        jugada_ev_incrementar();
                                }
                        }
                        break;
                default: //Si estamos en contando_trd
                        if(cuenta_ticks == trd_en_ticks_timer0)
                        {       //Si ha pasado trd, rehabilitamos interrupciones botón y
                                //  volvemos a admitir el procesado de otras pulsaciones
```

```
                                button_resetear();
                                atendiendoPulsacion = 0;
                                maquina_estados = Inicio;
                                cuenta_ticks = 0;
                        }
                        break;
        }
}
```

Código 16 - Función `antirebotes()` que implementa el comportamiento del autómata

También, para encapsular todas las acciones a realizar en caso de que llegue un tick de timer0, se ha implementado una función, **`button_ev_tick0()`** que será llamada desde el bucle gestor de eventos cuando desencole un evento de este temporizador. Dicha función controla todas las actualizaciones de variables y guardas necesarias, y avisa a la función del autómata para procesarlas, que se encarga de cambiar de estado y hacer las acciones, o no, si no le corresponde actuar.

```
void button_ev_tick0(void)
{       //Solo se incrementa el contador si es útil para la máquina de estados
        //      por tanto, si no estamos atendiendo ningún evento de pulsación no
        //      nos molestamos en hacer nada
        if(atendiendoPulsacion)
        {
                cuenta_ticks += 1;
                cuenta_autoincremento += 1;
                antirebotes();
                //Se avisa a la máquina de estados del cambio en las variables
                //      por si se activa alguna transición
        }
}
```

Código 17 - Detalle de la función `button_ev_tick0()`

Se ha procedido de igual manera, pero con los botones, de forma que si no se estaba atendiendo otra pulsación anteriormente, se comienza el tratamiento de los rebotes de esta, inicializando el autómata. Se llama **`button_ev_pulsacion()`** y necesita como parámetro el botón pulsado para inicializar correctamente el autómata. Ambas funciones completan las funciones requeridas por los profesores en el enunciado.

```
void button_ev_pulsacion(enum estado_button boton)
{       //Solo se hace algo si no se está gestionando otra pulsación
        //      así aseguramos que solo se gestiona un evento de pulsación a la vez
        if(!atendiendoPulsacion)
        {
                que_button = boton;
                atendiendoPulsacion = 1;
                antirebotes();                  //Se avisa a la máquina de estados para
                                                // que inicie su ejecución
        }
}
```

Código 18 - Detalle de la función `button_ev_pulsacion()`

Con esto, ya se tienen los botones funcionando y sin rebotes, el latido, y el 8-led, pero cada componente funciona de forma independiente a los demás. Ahora queda unirlos bajo una lógica común que los administre y coordine.

5.7 Vamos a jugar: `jugada_por_botones`

El último paso de los obligatorios para la segunda práctica consiste en integrar el funcionamiento de los botones y el 8-led a través del módulo `jugada_por_botones`, que ha sido implementado en los ficheros .c y .h de igual nombre. Este módulo coordinará esos dispositivos, y permitirá una

aproximación a lo que se conseguirá en la tercera práctica, elegir fila y columna de nuestra jugada a través de los botones de la placa.

Dicha implementación se conseguirá a través de otra máquina de estados. Su funcionamiento es el siguiente:

1. El 8-led muestra una 'F' de fila y cuando se pulse el botón izquierdo muestra un 1
2. El botón izquierdo incrementa el número de la fila, de 1 a 8 y vuelta al 0
3. El botón derecho confirma esa fila y muestra por el 8-led una 'C' de columna
4. Se repite igual la selección para la columna
5. Se llama a `reversi_procesar`, que procesa nuestra jugada, y se reinicia el ciclo.

La función encargada de procesar nuestra jugada aún no está implementada, se detallará su funcionamiento más adelante. (Ver Sección 4.7.3)

5.7.1 Diseño y diagrama de estados

Conocidas las especificaciones, se diseña un autómata de 4 estados, uno inicial, donde se ha puesto la 'F' y se espera al botón izquierdo, otro donde se elige la fila, otro donde se ha confirmado con el botón derecho y se espera al botón izquierdo para quitar la 'C' de la pantalla, y uno más donde se elige la columna y se confirma, después del cual se vuelve a principio, procesando la jugada y poniendo otra vez más la 'F'. El autómata de estados diseñado, con parte de la implementación se muestra a continuación.

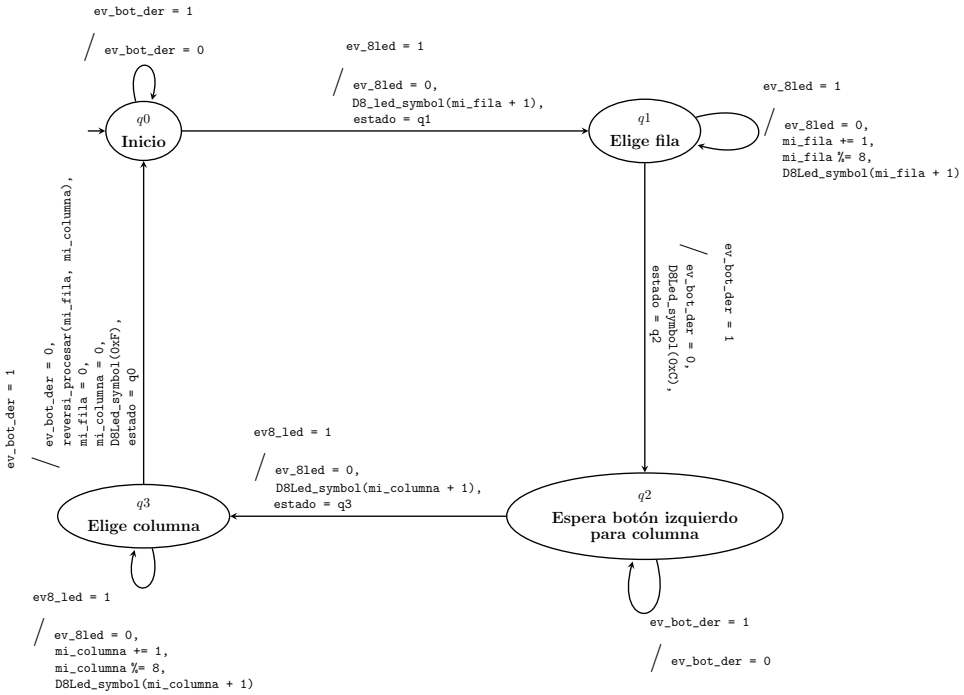

Figura 5 - Autómata de estados de `jugada_por_botones`

5.7.2 Implementación en lenguaje C

Para la implementación de la máquina de estados, se ha seguido una estrategia igual a la anteriormente descrita para el filtrado de los rebotes, es decir, se utilizan variables que codifican las entradas que pueden hacer cambiar de estado al autómata, el cual se codifica con una variable enum. Además, el comportamiento de la máquina de estados queda definido en una única función, llamada `jugada_por_botones()`.

También se cuenta con funciones auxiliares, que son las que se llamarán desde el bucle de gestión de eventos en `reversi_main()` para cambiar los valores necesarios de las variables, y avisan a la función del autómata, que comprueba si debe realizar alguna acción con el nuevo estado de las variables. Posteriormente, para la práctica 3, las acciones a realizar desde la máquina de estados se volvieron demasiado numerosas como para dejar todo en una única función, por lo que se añadirán más funciones auxiliares, pero la idea seguirá siendo la misma. A continuación, se incluye el código correspondiente a la implementación en C de la versión de la práctica 2:

```
/*--- variables del módulo ---*/
static enum estados{q0, q1, q2, q3} jugada_botones;
static int ev_8led = 0;                         //Indica cuándo hay que actualizar el
                                                // valor que aparece en el 8led,
                                                // por nueva pulsación del botón
                                                // izquierdo o por autoincremento
static int ev_bot_der = 0;                      //Flag que indica pulsación botón derecho
static char mi_fila = 0;                        //La fila donde queremos poner ficha
static char mi_columna = 0;                     //La columna donde queremos poner ficha

void inicializar_jugada_botones()
{
        jugada_botones = q0;
        mi_fila = 0;
        mi_columna = 0;
        ev_8led = 0;
        ev_bot_der = 0;
        D8Led_symbol(0xF);
}
void jugada_por_botones()
{
        switch(jugada_botones)
        {
        case q0:                                //Estado inicial, 'F'en 8-led espera b.iz
                if(ev_bot_der == 1)
                {
                        ev_bot_der = 0;
                }
                if(ev_8led == 1)                //Pone el 0 y cambia de estado
                {
                        ev_8led = 0;
                        D8Led_symbol(mi_columna+1);
                        jugada_botones = q1;
                }
                break;
        case q1:                                //Selección fila y espera botón derecho
                if(ev_bot_der == 1)
                {
                        ev_bot_der = 0;
                        D8Led_symbol(0xC);
                        jugada_botones = q2;
                }
                if(ev_8led == 1)
                {
                        ev_8led = 0;
                        mi_fila +=1;
                        mi_fila %= 8;           //Si llega a 9 se reinicia modulo 8
                        D8Led_symbol(mi_fila+1);
                }
                break;
        case q2:                                //Pone la 'C' y espera botón izquierdo
                if(ev_bot_der == 1)
                {
                        ev_bot_der = 0;
                }
                if(ev_8led == 1)                //Se pone el 0 y cambia de estado
                {
                        ev_8led = 0;
                        D8Led_symbol(mi_columna+1);
                        jugada_botones = q3;
                }
                break;
        default:                                //q3, selección columna y espera bot der
                if(ev_bot_der == 1)             //Cuando confirman jugada
                {
                        ev_bot_der = 0;
```

```
                        reversi_procesar(mi_fila,mi_columna);       //Procesa la jugada
                        mi_fila = 0;
                        mi_columna = 0;
                        D8Led_symbol(0xF);
                        jugada_botones = q0;                        //Reinicia autómata
                }
                if(ev_8led == 1)              //Actualiza 8led modulo 8
                {
                        ev_8led = 0;
                        mi_columna +=1;
                        mi_columna %= 8;
                        D8Led_symbol(mi_columna+1);
                }
                break;
        }
}

void jugada_ev_der()
{
        ev_bot_der = 1;
        jugada_por_botones();
}

void jugada_ev_incrementar()
{
        ev_8led = 1;
        jugada_por_botones();
}
```

Código 19 - Implementación del autómata de `jugada_por_botones` en C, práctica 2

A pesar de que el nombre de los estados no sea muy descriptivo, en versiones posteriores de la tercera práctica se aprovechó que había que rehacer buena parte de este módulo para dar nombres más descriptivos a los estados del autómata.

5.7.3 Unión con reversi8.c

Finalmente, para unir toda la lógica del juego con el módulo `reversi8.c` que proporcionaron originalmente los profesores, se ha implementado una función más en ese fichero, `reversi_procesar()`, que recibe como parámetros la fila y la columna de la jugada a procesar y se los pasa a las variables del módulo, y pone la señal interna de ready a 1.

Por último, avisa a `reversi8()` para que vea que ready está a 1, y procese la jugada. Además, se ha añadido a la declaración de las variables la palabra clave **static** para su correcto funcionamiento.

La implementación resulta sencilla:

```
. . .

static volatile char fila=0, columna=0, ready=0;
void reversi_procesar(char f, char c)
{
        fila = f;
        columna = c;
        ready = 1;
        reversi8();
}

. . .
```

Código 20 - Implementación de `reversi_procesar()`

5.8 Apartados opcionales de la práctica 2

A continuación se va a explicar el diseño y la implementación de los apartados opcionales de la práctica 2, a excepción del autoincremento, el cual ya se ha explicado aprovechando la sección del autómata de `botones_antirebotes` (Sección 4.6.2)

5.8.1 Estudio del linker script y fallos detectados

El fichero de linker `ld_script.ld` proporcionado es correcto. Podría haber aparecido un problema si no se hubiera puesto la directiva `ALIGN(4)` después de colocar la sección `.text` (correspondiente al código del programa) en memoria, ya que, si tuviera instrucciones THUMB, la sección `.data` (correspondiente a variable inicializadas) podría comenzar desalineada.

En el fichero 44blib.c, se detectó un fallo relativo al linker script. En puntero que utiliza la función `malloc(unsigned nbyte)`, llamado `mallocPt`, se inicializa con la dirección `Image_RW_Limit` del linker script. Si en la sección `.bss` de la memoria, que contiene las variables inicializadas, hubiera datos, estos serían sobrescritos al utilizar dicha función. La solución sería inicializar el puntero con la dirección `Image_ZI_Limit`.

6. Metodología Práctica 3

Ahora que se ha explicado todo el trabajo realizado durante la segunda práctica, se puede ofrecer una visión más global y esquemática de cómo interactúan entre sí los módulos desarrollados, y cómo se logra la máxima modularidad posible para el tratamiento de los diferentes eventos que generan los periféricos conectados a la placa.

Para ello, se muestra en la Figura 8 un diagrama con los módulos, donde las flechas amarillas indican cuándo un módulo emplea las funciones de otro, y las flechas grises indican que ese módulo genera un evento. También se explica de forma resumida todo lo que ocurre en el sistema desde una pulsación del botón izquierdo hasta que se actualiza el valor mostrado en el 8-led. Se excluye timer2, ya que tan solo lo utiliza

Diagrama de interacción entre los componentes desarrollados en la práctica 2

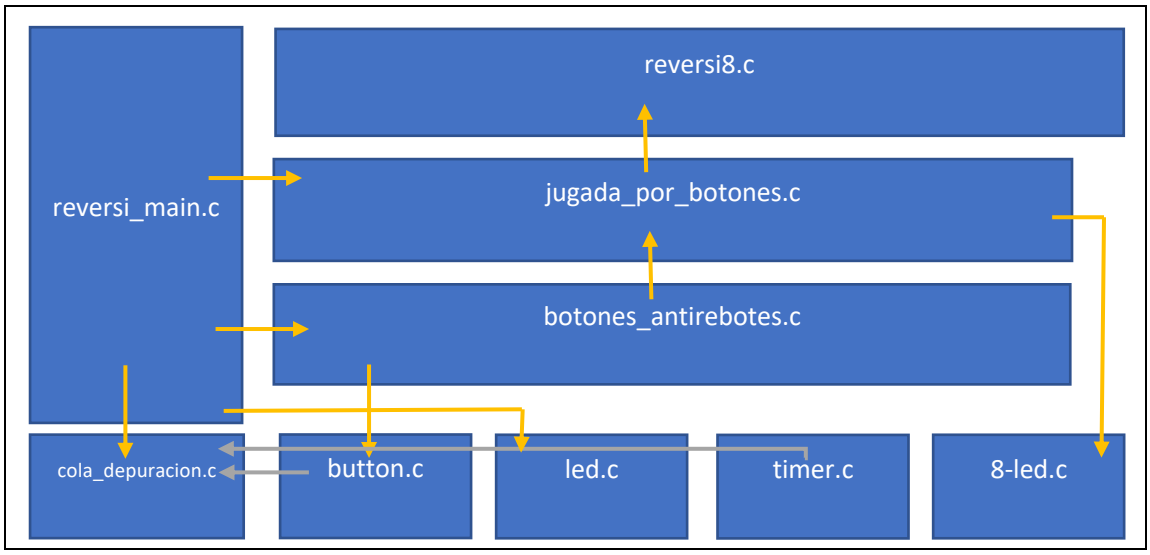

Figura 6 - Esquema de interacción entre los módulos desarrollados en la práctica 2

Ejemplo: pulsación de un botón

Con el fin de ofrecer un ejemplo que resuma gran parte de lo que se ha explicado hasta ahora, correspondiente al desarrollo de la práctica 2, se va a explicar todo lo que ocurre en la placa desde que se pulsa el botón izquierdo, para ofrecer una visión más amplia de cómo los diferentes módulos se pasan los eventos, o los tratan. El comportamiento de la placa nada más pulsar el botón izquierdo sería el siguiente:

1. Se entra en la interrupción del botón, que desactiva las interrupciones casi al instante, detecta qué botón se ha pulsado y encola un evento de pulsación, con el campo de información adicional indicando que ha sido el botón izquierdo. La cola de depuración, aparte del evento introducirá la marca de tiempo resultante de leer de timer2.

2. El bucle de **reversi_main()** detecta que la cola de eventos ya no está vacía, por lo que desencola este evento, y ve que es una pulsación del botón. Reacciona a este evento, llamando a las funciones que tratarán el evento: por un lado, **button_ev_pulsacion(),** que se encargará de iniciar el tratamiento de los rebotes causados en la señal eléctrica del pulsador, y por otro lado, avisa a **jugada_ev_incrementar()**, en el autómata de jugada por botones que gestionará el comportamiento del 8-led.

3. Por un lado, **jugada_ev_incrementar()** pondrá la variable que actúa de flag de pulsación del botón izquierdo a 1, y llamará a la función del autómata, **jugada_por_botones()**. La función del autómata verá, por ejemplo, que con el estado en el que está y esa entrada debe actuar. Para ello, bajará el flag a 0 de nuevo, pondrá un 1 en el 8-led (suponiendo que por ejemplo estaba mostrando la F de fila) y cambiará su estado interno.

4. Por otro lado, **button_ev_pulsacion** comprobará que no hay ninguna otra pulsación siendo tratada, y comenzará a tratar esta. Para ello, la variable **que_button** informará de que es el botón izquierdo y se lamará a la función principal del autómata, **antirebotes()**.

5. El timer0 irá interrumpiendo, y generando eventos. En reversi_main() se tratan, lo que incluye una llamada a **button_ev_tick0()**. Esta función el módulo de filtrado de rebotes comprueba que se está atendiendo una pulsación, y como es así, incrementa la variable interna que cuenta el número de ticks de timer0, para luego avisar al autómata.

6. El autómata cambiará de estado cuando haya pasado el tiempo (número de ticks de timer0) correspondiente a trp.

7. Estos eventos del timer0 se seguirán tratando de la misma forma, hasta que en el siguiente estado hayan pasado los ticks correspondientes a 30 milisegundos. Entonces, el autómata empezará a encuestar al botón para ver su estado, a través de la función del módulo de los botones, **que_button().** Cuando se detecte que ya no se está pulsando el botón que le han dicho al autómata que se había pulsado, cambiará de estado.

8. En el siguiente estado, procede igual que en el paso 5 y en el 6, pero contando ticks para trd.

9. Cuando haya pasado este número de ticks de timer0, el autómata volverá al estado inicial, invocando primero a **button_resetear()** para permitir de nuevo interrupciones desde esa línea. También reinicia sus variables al estado inicial y pone a 0 su variable que indica si se está atendiendo una pulsación, para permitir que se gestionen los rebotes del resto de pulsaciones a partir de ese momento.

Pasos, diseño e implementación realizados en la práctica 3

A partir de este punto, la memoria se centrará en describir el proceso de diseño e implementación de los apartados que componen la práctica 3. Se intentará resumir más los apartados en los que solamente haya que cambiar la implementación desarrollada en la práctica 2, puesto que la idea clave del diseño ya se ha explicado, y se centrará en los cambios. En aquellos apartados que sean propios de esta práctica, se comentará el diseño junto con la implementación.

Como ya se ha comentado, el objetivo de la práctica es desarrollar una plataforma autónoma, por lo que la mayoría de los pasos irán orientados a conseguir que se pueda jugar enteramente en la placa, eligiendo movimientos con esta y sin tener que cargar constantemente el programa del juego en ella, para lo que se flasheará.

6.1 Pasar a modo usuario

Lo primero que se implementó fue hacer que el juego debía ejecutarse en modo usuario. Hasta ahora, como al iniciar la placa se ejecutaba un reset, se estaba corriendo en modo supervisor. El cambio de modo se produce antes de entrar en el bucle principal de `reversi_main()`.

El apartado en sí es sencillo, dado que solamente hay que obtener y modificar el CPSR, y luego volver a guardarlo.

La única dificultad del apartado podría estar en la pila del modo usuario, ya que comienza al final del heap. Podría ser que al haber colocado la pila de depuración lo suficientemente cerca del heap, esta desbordara y sobrescribiera los valores de la pila, si crece lo suficiente, provocando fallos catastróficos en los retornos de las funciones. Se tuvo precaución en la implementación de la cola de depuración, y se colocó a la distancia suficiente, 4 veces la separación normal entre pilas dejada por defecto, para asegurar que esto no ocurría.

6.1.1 Implementación en lenguaje C

La función `pasar_a_usuario()` se ha definido en el módulo main.c, y se llama desde el propio `Main()`, justo antes de entrar en el bucle del juego, `reversi_main()`. Se hace uso del ensamblador *inline* para cambiar de modo, obteniendo CPSR en una variable, que se modifica con operaciones *bitwise* en C, y se devuelve al CPSR, con la instrucción en ensamblador.

```
static volatile inline void pasar_a_user()
{
	uint32_t _cpsr;
	asm volatile (" mrs %0, cpsr\n" : "=r" (_cpsr));
	_cpsr &= 0xFFFFFFE0;		//El modo son los 5 bits de menos peso
	_cpsr |= 0x00000010;		//0x10 = modo usuario<
	asm volatile (" msr cpsr,%0\n" :: "r" (_cpsr));
	asm volatile (" ldr sp, =0xc7ff000 ");
	return;
}
```

Código 21 - Función para pasar a modo de ejecución de usuario

6.2 Interrupciones FIQ

Otro de los primeros pasos que se han dado de cara a la implementación de la práctica 3 ha sido cambiar el modo en el que interrumpe timer2, de forma que sea una interrupción rápida, ya que el modo FIQ cuenta con mayor prioridad. Para ello, hay que modificar ligeramente la declaración de la función de interrupción del timer2, indicando que ahora es una FIQ. Además, al inicializar ahora hay que asignarle el valor a `pISR_FIQ`, que se encarga de este tipo de interrupciones.

Como se comenta en el enunciado, sería un problema si hubiera diferentes fuentes de interrupción FIQ porque estas interrupciones no están autovectorizadas y habría por tanto que distinguir entre las fuentes de interrupción para atenderlas de una forma u otra. Pero, solo se tiene una fuente de interrupción por lo que no es necesario.

También se ha tenido cuidado con los diferentes registros de control de los periféricos, porque en las funciones de inicialización se modifica INTCON, responsable de habilitar FIQ e IRQ. Afortunadamente, se hicieron bien las demás asignaciones de valor en las inicializaciones y no ha habido problemas. También se debía tener cuidado en esta función al modificar INTCON ya que había que habilitar FIQ, pero sin modificar el valor que hubiera en el bit correspondiente a las FIQ.

6.2.1 Implementación en lenguaje C

El código modificado para que timer2 interrumpa por FIQ se muestra a continuación (Código 22):

```
void timer2_ISR(void) __attribute__((interrupt("FIQ")));

. . .

void timer2_inicializar(void)
{
        /* Configuraion controlador de interrupciones */
        rINTMOD |= BIT_TIMER2; //Configura la línea del timer2 como FIQ (1 FIQ, 0 IRQ)
        rINTCON &= 0x6; // Habilita int. vectorizadas y la línea FIQ, no toca IRQ
        rINTMSK &= ~(BIT_TIMER2); // Habilitamos vec. máscaras de interrupción Timer0
        /* Establece la rutina de servicio para TIMER2 */
        pISR_FIQ = (unsigned) timer2_ISR;
        /* Configura el Timer2 */
        rTCFG0 &= 0xFFFF00FF; // ajusta el preescalado a 0
        rTCFG1 &= 0xFFFFF0FF; // selecciona la entrada del mux que proporciona el
                              // reloj. La 00 corresponde a un divisor de 1/2.
}
```

Código 22 - Código modificado de timer2 para que interrumpa por FIQ

6.3 Juego en pantalla

El mayor peso de la práctica recae sobre lograr que el juego se muestre por pantalla, y lograr confirmar la jugada al tocarla. Para el desarrollo de este apartado se han tenido que redefinir algunos módulos ya implementados, utilizar algunos ya proporcionados por los profesores, y definir otros nuevos.

Además, como la tarea de lograr jugar a través de la pantalla es demasiado extensa, se ha tratado de dividir las explicaciones en diferentes apartados correspondientes a tareas menores, que entre todas logran hacer que sea posible jugar a través de la pantalla de la placa.

6.3.1 Desarrollo de un módulo para mostrar elementos del juego en pantalla

En este apartado, lo primero que se intentó fue mostrar elementos por pantalla, y comprender el funcionamiento de las funciones incluidas en `lcd.c` y `lcd.h`. Estas funciones se dedican a mostrar las formas más básicas por pantalla, como líneas, cuadrados o texto ASCII, en dos tamaños diferentes.

También había funciones de inicialización y para limpiar la pantalla otra función que ponía en marcha una transferencia de DMA. Y es que para poner algo en pantalla, estas funciones lo ponen en una zona de la memoria, denominada pantalla virtual, que con un DMA se mueve a otra zona de memoria, donde el controlador de la pantalla se encarga del resto de tareas necesarias para mostrar la información correctamente. Esto se tendrá en cuenta posteriormente, en la lógica del juego (4.11.3)

Diseño del módulo

Una vez comprendido el funcionamiento de la pantalla, se llegó a la conclusión de que lo mejor sería actualizar los elementos del juego según se necesitara, no todos a la vez. Por tanto, se ha dividido la pantalla principal del juego en secciones, para crear funciones diferentes que se encarguen de cada una de ellas. Estas funciones constituirán un módulo aparte, que se servirá de las funciones más básicas de `lcd.`

Las diferentes zonas de la pantalla consideradas han sido:

- Tablero (cuadrícula y números de filas y columnas, no necesitan refresco)
- Textos de profiling e información durante el juego (no necesitan refresco)
- Información de profiling (valores de los parámetros a mostrar por pantalla, hay refresco)
- Pantalla de reglas (no necesita refresco)
- Pantallas de victoria, derrota y empate (no necesitan refresco)
- Pintar puntuación final (auxiliar a las anteriores)
- Pintar fichas (necesitará coordenadas en la cuadrícula, color de ficha y hay que actualizar)

Todos aquellos elementos que necesiten refresco, o algún tipo de actualización necesitarán una función que borre los elementos previos, antes de pintar los nuevos.

La pantalla principal del juego contendrá el tablero, las fichas, textos de información y los datos de profiling.

Implementación en lenguaje C

Este conjunto de funciones se decide implementar en dos nuevos ficheros, `elementos_pantalla.c` y `elementos_pantalla.h.` Además, en las etapas finales del desarrollo, se utilizaron bitmaps para representar las fichas, puesto que en las demás versiones tan solo se utilizaban letras de diferentes colores dentro de la escala de grises.

Surge un pequeño problema con el profiling, ya que los valores son numéricos, y se necesitaban pasar a cadenas de texto para que la pantalla pudiera representarlos correctamente. Se probó incluyendo librerías por defecto de C pero no dio resultado, por lo que se definió una función `itoa()` que hiciera este trabajo. Como necesita dos funciones auxiliares, se colocó en un módulo propio, en los ficheros `funciones_itoa.c` y `funciones_itoa.h.`

El código de este módulo es extenso, por lo que se solo se muestra parte de él, para ilustrar cómo utiliza las funciones más básicas del módulo que fue proporcionado al principio.

```
void pintar_cuadricula()
{
        volatile int i;
        volatile int j;
        for(i = 0; i < num_filas; i++)
        {
                for(j = 0; j < num_columnas; j++)
                {
                        Lcd_Draw_Box(100+25*j,20+25*i,125+25*j,45+25*i,BLACK);
                }
        }
}

void pintar_numeros_tablero()
{
        int i;
        char buffer[15];
        for(i = 0; i < num_filas; i++)
        {
                Lcd_DspAscII8x16(110 + 25*i, 0,BLACK,itoa(i+1, buffer,10));// base 10
```

```
                Lcd_DspAscII8x16(310,25 + 25*i,BLACK,itoa(i+1, buffer,10));
        }
}
void pintar_textos()
{
        . . .
}

void pintar_ficha(int fila, int col, enum estado_casilla color)
{
        switch(color)
        {
                case FICHA_BLANCA:
                        BitmapView(101+25*col, 21+25*fila, Stru_Bitmap_fichaBlanca);
                        break;
                case FICHA_NEGRA:
                        BitmapView(101+25*col, 21+25*fila, Stru_Bitmap_fichaNegra);
                        break;
                case FICHA_GRIS:
                        BitmapView(101+25*col, 21+25*fila, Stru_Bitmap_fichaGris);
                        break;
                default:
                        break;
        }
}

void borrar_ficha(int f, int c)
{
        LcdClrRect((100+25*c) + 1, (20+25*f) + 1, (125+25*c) - 1, (45+25*f) - 1, 0x0);
}

//Mueve en el LCD la ficha gris a partir de la fila y columna en la que está (0-7)
void mover_gris(int fila, int columna, int fila_anterior_gris, int columna_ante-
rior_gris)
{
        borrar_ficha(fila_anterior_gris, columna_anterior_gris);
        pintar_ficha(fila, columna, FICHA_GRIS);
        Lcd_Dma_Trans();
}

void pintar_profiling(int t_total, int t_calc, int t_pvolteo, int veces_pvolteo)
{
        char buffer[15];        //15 por darle un tamaño decente

        //Se borra la zona de pantalla virtual donde están los numeritos del profiling
        //      que si no funciona :)
        LcdClrRect(20, 35, 75, 50, 0x0000);
        LcdClrRect(20, 85, 95, 100, 0x0000);
        LcdClrRect(20, 135, 95, 150, 0x0000);
        LcdClrRect(20, 185, 75, 200, 0x0000);

        itoa(t_total, buffer, 10);    //10 es para indicar que pasamos a base 10
        Lcd_DspAscII8x16(20,35,BLACK,buffer);
        itoa(t_calc, buffer, 10);
        Lcd_DspAscII8x16(20,85,BLACK,buffer);
        itoa(t_pvolteo, buffer, 10);
        Lcd_DspAscII8x16(20,135,BLACK,buffer);
        itoa(veces_pvolteo, buffer, 10);
        Lcd_DspAscII8x16(20,185,BLACK,buffer);
}

void pintar_jugando()
{
        Lcd_Clr();
        pintar_cuadricula();
        pintar_numeros_tablero();
        pintar_textos();
        pintar_profiling(0,0,0,0);
        pintar_ficha(0,0,3);  //Pintar la ficha gris en 0,0
        pintar_ficha(3,3,FICHA_BLANCA);//Pintar fichas blancas iniciales en 3,3 y 4,4
        pintar_ficha(4,4,FICHA_BLANCA);
        pintar_ficha(3,4,FICHA_NEGRA);//Pintar las fichas negras en 3,4 y 4,3
        pintar_ficha(4,3,FICHA_NEGRA);
        Lcd_Dma_Trans();
}
. . .
void pintar_reglas()
```

```
{
        Lcd_Active_Clr();
        Lcd_DspAscII8x16(110,1,BLACK,"REVERSI 8");
        Lcd_DspAscII6x8(70,20,BLACK,"- A HIDEO KOJIMA GAME -");
        . . .
        Lcd_Dma_Trans();
}

void pintar_resultados(int blancas, int negras) {
        volatile char buffer[15];
        Lcd_DspAscII8x16(30,80,BLACK,"Tus fichas:");
        Lcd_DspAscII8x16(30,100,BLACK,"Fichas de CPU:");
        itoa(negras, buffer, 10);     //10 es para indicar que pasamos a base 10
        Lcd_DspAscII8x16(126,80,BLACK,buffer);
        itoa(blancas, buffer, 10);    //10 es para indicar que pasamos a base 10
        Lcd_DspAscII8x16(150,100,BLACK,buffer);
        Lcd_DspAscII8x16(100,190,BLACK,"TOCA LA PANTALLA");
        Lcd_DspAscII8x16(80,210,BLACK,"PARA REINICIAR LA PARTIDA");
}
void pintar_fin_victoria(int blancas, int negras)
{
        . . .
}
void pintar_fin_derrota(int blancas, int negras)
{
        . . .
}
void pintar_fin_empate(int blancas, int negras)
{
        . . .
}
void iniciar_DMA()
{
        Lcd_Dma_Trans();
}
```

Código 23 - Código para mostrar los elementos del juego por pantalla

6.3.2 TouchPad TSP, interrupciones y filtrado de rebotes

A continuación, se debía configurar el otro aspecto clave para esta tercera práctica, la pantalla táctil. Tras estudiar el proyecto de ejemplo y examinar los ficheros `tp.c` y `tp.h`, se configuraron adecuadamente las funciones de inicialización y se sustituyó todo el código de la rutina de interrupción por un apilado de un nuevo tipo de evento, `ev_tsp`, que será atendido como corresponda. (Se explica en el apartado 4.11.3).

Además, la pantalla puede presentar rebotes en la señal eléctrica, que se percibieron durante las pruebas iniciales aunque apenas se manifestaban. Para su eliminación se ha adoptado una estrategia parecida a la tomada con los botones, otro autómata de estados.

Configuración de las interrupciones de la pantalla táctil

Se eliminó todo el código de la rutina, ya que se decidió que no se iban a implementar botones táctiles en la pantalla para simplificar las operaciones a realizar al atender este evento.

Aunque los primeros días no daba ningún problema, al probar una versión casi definitiva del código se encontró un fallo inusual, ya que con -O3 el programa se perdía en direcciones de memoria aleatorias cada vez que llegaba a la rutina de inicialización de la pantalla táctil. Para evitar esto, la única solución posible fue no optimizar esta función. Apenas tiene impacto al rendimiento ya que la inicialización solo ocurre una vez, al principio de la ejecución del juego. El código se muestra en el fragmento de debajo.

```
void TSInt(void)
{
        rINTMSK |= (BIT_EINT2);       //Deshabilitamos interrupcion linea eint1 en
                                      // vector de máscaras para el antirebotes de tsp
```

```
        push_debug(ev_tsp, no_info);
        rI_ISPC |= BIT_EINT2; //Limpiar bit pendiente en INTPND
        //El código que había aquí debajo ha sido eliminado porque no nos hace falta
}
. . .
void __attribute__ ((optimize("O0"))) TS_init(void)
{
    rINTMOD &= ~(BIT_EINT2);  // Configura las líneas de TSP como de tipo IRQ
    rINTCON &= 0x1; // Habilita int. vectorizadas y linea IRQ (FIQ la deja igual)
    rI_ISPC |= BIT_EINT2;            // clear pending_bit
    rPUPE  = 0x0;                    // Pull up
    rPDATE = 0xb8;                   // should be enabled
    rEXTINT |= 0x200;                // falling edge trigger
    pISR_EINT2=(unsigned *)TSInt;    // set interrupt handler
    rCLKCON = 0x7ff8;                // enable clock
    rADCPSR = 0x1;//0x4;             // A/D prescaler
    rINTMSK &= ~(BIT_EINT2);
    oneTouch = 0;
}
```

Código 24 - Inicialización y rutina de interrupción de la pantalla táctil

Filtrado de los rebotes de la pantalla táctil

Para eliminar los rebotes de la pantalla se ha implementado un autómata, aunque este es más sencillo que el de filtrado de rebotes en los botones. Como la pantalla no se va a mantener tocada, ni interesa considerar ese caso, solo se consideran dos estados: que no está siendo tocada y que está siendo tocada, y por tanto hay que filtrar rebotes. Los rebotes de la señal se filtran desactivando las interrupciones del tsp y esperando una cantidad de tiempo prefijada. Este tiempo se mide como en los casos anteriores, con el timer0. Para ello, se añade al tratamiento de ese evento en `reversi_main()` una invocación a la función de este módulo que vaya a tratarlo.

Como se muestra encima (Código 24) , la inhabilitación de las interrupciones se produce nada más entrar en la rutina de servicio. A continuación se muestran tanto el autómata generado, como parte de su implementación en C.

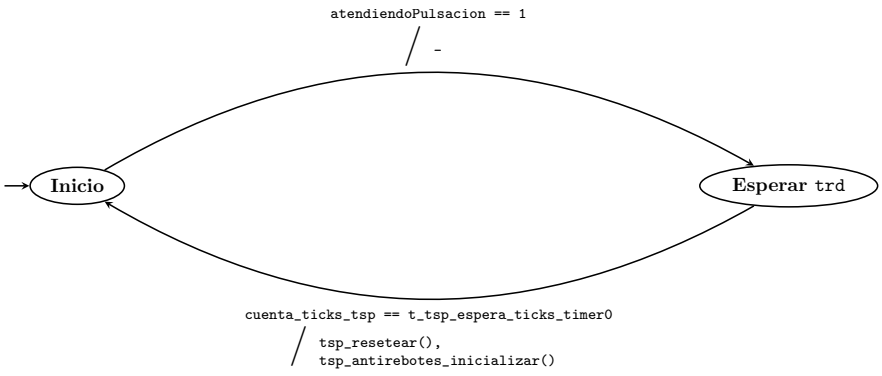

Figura 3: Autómata de `tsp_antirebotes.c`

Figura 7 - Autómata de estados para el filtrado de rebotes en el tsp

El módulo que contiene el autómata implementado en C se ha llamado `tsp_antirebotes`, y funciona de manera similar a los autómatas ya implementados y explicados anteriormente. Al implementarlo, se ha elegido que el tiempo que dure la inhabilitación de las interrupciones sea de medio segundo, unos 30 ticks de timer0.

```
/*--- variables del módulo ---*/
enum {t_espera_ticks_timer0 = 30};
static int cuenta_ticks_tsp = 0;
static int atendiendo_pulsacion_tsp = 0;
/*
   Los valores se pueden cambiar en función de la placa
   Desactivamos las interrupciones durante 30 ticks = 30 * 1/60 seg = 0.5 segundos.
*/
static enum estados_tsp_antirebotes{Inicio, deshabilitadas_int} maquina_estados_tsp;
//Estados de la máquina de estados
void tsp_antirebotes_inicializar()
{
        maquina_estados_tsp = Inicio;
        cuenta_ticks_tsp = 0;
        atendiendo_pulsacion_tsp = 0; //Inicialmente no se está atendiendo
                                      //ninguna pulsación, por eso se inicializa a 0
}

void tsp_antirebotes(void)
{
        switch(maquina_estados_tsp)
        {
                case Inicio :
                        if(atendiendo_pulsacion_tsp)
                        {
                                //Las interrupciones se deshabilitan en IRQ por si acaso
                                maquina_estados_tsp = deshabilitadas_int;
                        }
                        break;
                default:        //Si estamos en deshabilitadas_int
                        if(cuenta_ticks_tsp == t_espera_ticks_timer0)
                        {       //Si ha pasado trd, rehabilitamos interrupciones tsp
                                // y volvemos a admitir procesado de otras pulsaciones
                                tsp_resetear();
                                tsp_antirebotes_inicializar();
                        }
                        break;
        }
}
void tsp_ev_pulsacion()
{
        . . .
}
void tsp_ev_tick0(void)
{
        . . .
}
```

Código 25 - Autómata de estados de `tsp_antirebotes`

6.3.3 Cambios en `reversi8` para obtener datos de profiling

Como se especifica en el enunciado, se pretende mostrar en pantalla durante la partida datos sobre el rendimiento del código de procesamiento del juego, como por ejemplo el tiempo en segundos de partida, el número de invocaciones a `patron_volteo()`, el tiempo que se pierde en estas invocaciones, o el que se pierde en total al procesar las jugadas.

Para ello, hay que partir del código proporcionado por los profesores al principio de la primera práctica en `reversi8_2019.c`. En él se añaden variables que permiten contar el número de veces que se invoca `patron_volteo()`. Y tal como se midió en la práctica 1, la implementación más rápida es la original en C, por lo que será la que se utilice para el juego final.

Para medir el tiempo de estas invocaciones, se ha utilizado el timer2, puesto que está calibrado para ofrecer la precisión deseada (microsegundos). Para realizar estas lecturas, se lee de él al principio y al final de la función, y se actualiza una variable que almacena el tiempo de todas las invocaciones durante el procesamiento de todo ese turno.

Para calcular los tiempos de procesamiento se ha hecho parecido, pero midiendo directamente en el autómata (4.11.4) antes y después de llamar a la función `reversi_procesar()`. La cuenta del tiempo total de juego se ha implementado diferente, con el timer0, y se explica en el apartado del autómata del juego (4.11.4). Se han implementado más funciones y se han realizado más cambios, pero están directamente relacionados con el funcionamiento del juego. Por ello, se detallan en la siguiente sección.

```
static int veces_pv = 0;
static int t_pv = 0;
. . .
int reversi_t_pv(void)
{
        return t_pv;
}
int reversi_veces_pv(void)
{
        return veces_pv;
}
. . .
int patron_volteo(char tablero[][DIM], int *longitud, char FA, char CA, char SF, char
SC, char color)
{
        int t_inicio = timer2_leer();
        int t_final;
        . . .
        if ((. . .))
        {
                t_final = timer2_leer();
                t_pv += (t_final - t_inicio);
                veces_pv++;
                return PATRON_ENCONTRADO;
        }
        else
        {
                t_final = timer2_leer();
                t_pv += (t_final - t_inicio);
                veces_pv++;
                return NO_HAY_PATRON;
        }
}
```

Código 26 - Modificaciones en `reversi8_2019.c` para datos profiling

Nota: Al momento de realizar la memoria, se detectó que `reversi_veces_pv()` y `reversi_t_pv()` devolvían los parámetros cambiados entre sí, por lo que al ejecutar el juego se mostrarían uno en el sitio del otro. Se ha corregido para la entrega final.

6.3.4 Nueva versión de jugada_por_botones y cambios para jugabilidad

Esta nueva versión del autómata es completamente diferente, teniendo que dar soporte a la estructura de partida descrita en el enunciado. Se ha tenido que rehacer por completo el autómata, y se han ido modificando pequeños fragmentos del resto del proyecto para añadir funcionalidades necesarias o de usabilidad.

Implementar autoincremento también en el botón derecho

Se ha tenido que modificar ligeramente `botones_antirebotes` para poder admitir que haya autoincremento en el botón derecho. Su nuevo diseño se encuentra en la fig. 10. Su implementación en C es casi idéntica, tan solo cambian las condiciones, que ahora tienen en cuenta ambos botones.

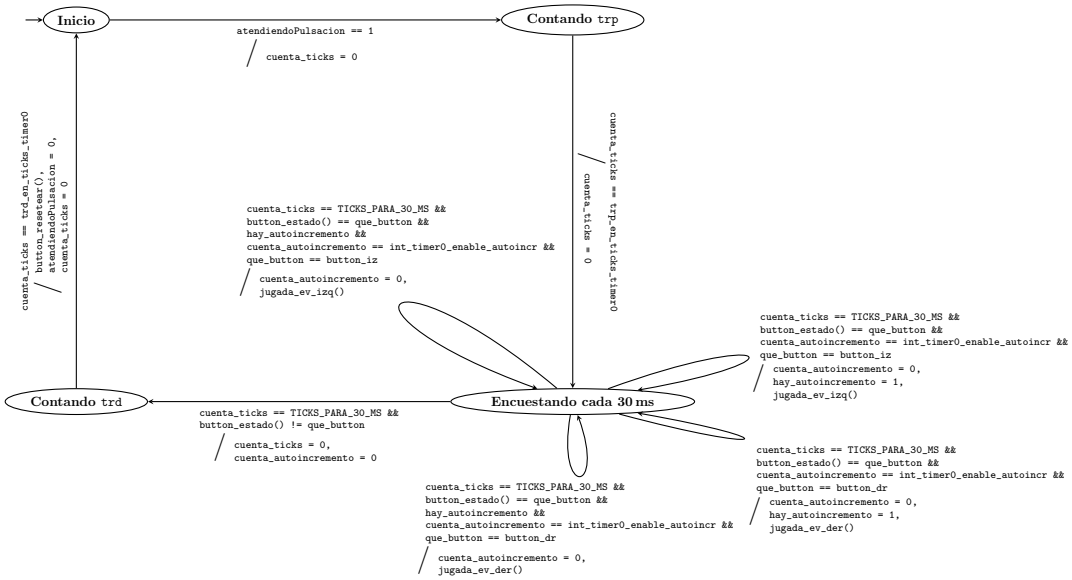

Figura 8 - Diseño final del autómata de botones antirebotes

```
. . .
if(cuenta_autoincremento == int_timer0_enable_autoincr)
        {       //Tras mantener pulsado durante 1/3 de segundo
                //autoincremento cada 180ms a partir de ahora
                cuenta_autoincremento = 0;
                hay_autoincremento = 1;
                if(que_button == button_iz)
                {
                        jugada_ev_izq();
                }
                else if(que_button == button_dr)
                {
                        jugada_ev_der();
                }
        }
        if(hay_autoincremento && cuenta_autoincremento == int_timer0_autoincr)
        {
                cuenta_autoincremento = 0;
                if(que_button == button_iz)
                {
                        jugada_ev_izq();
                }
                else if(que_button == button_dr)
                {
                        jugada_ev_der();
                }
        }
. . .
```

Código 27 - Cambios para el autoincremento de ambos botones

Diseño del autómata y la lógica del juego

El diseño del autómata de estados que rige el comportamiento del juego se basa en las pantallas principales que se encuentran a lo largo de la partida. Es decir, una pantalla de inicio con las reglas, otra con la partida, y otra para indicar que ha terminado.

Partiendo de esta idea, el autómata queda simple, pero como luego se verá, la complejidad se da en las transiciones durante la partida, como al terminar un turno, donde hay que procesar los movimientos realizados en la pantalla por el jugador y hay que recuperar los de la máquina, para

mostrarlos por pantalla. Además, se han diseñado estrategias para lograr que el número de actualizaciones de fichas en pantalla sea mínimo, únicamente lo necesario.

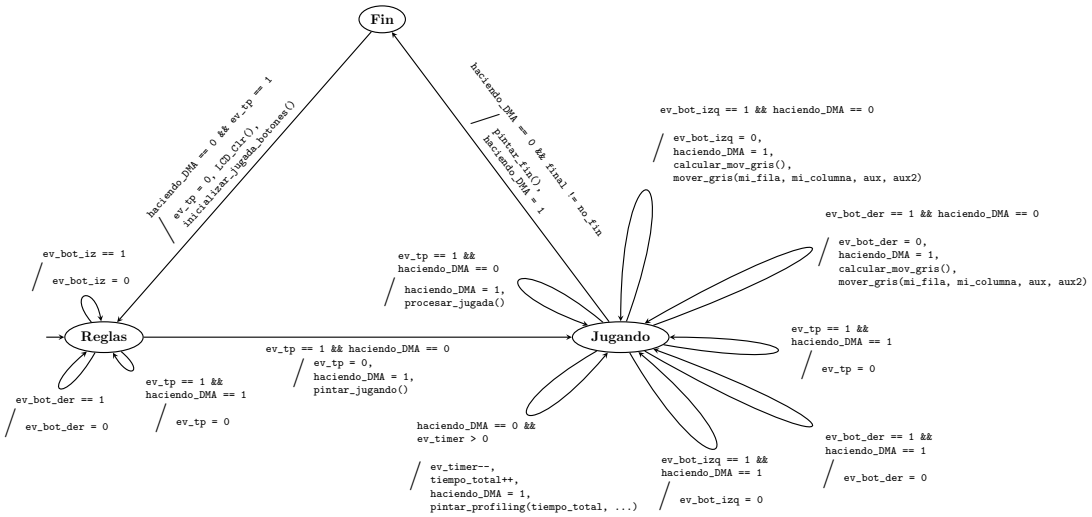

Figura 9 - Nuevo diagrama de estados para `jugada_por_botones`

Todas las actualizaciones en pantalla están sujetas a las transferencias de DMA que se están realizando. Por ejemplo, si llega un aviso de que se debe actualizar el tiempo total de juego pero está en curso otro DMA (por ejemplo, actualizar las fichas en pantalla), el autómata deberá esperar a que la transferencia termine antes de iniciar otra. Esto se debe a que si se cambian los datos que el DMA está moviendo en ese momento se puede ocasionar algún fallo.

Se podría haber diseñado alguna forma de tener más de una actualización en marcha, como designar otra zona de memoria como segundo buffer para nuestras escrituras, o dividir a mano la zona predefinida en los módulos, si por ejemplo, las actualizaciones son más frecuentes en una zona de la pantalla que en el resto. En esta ocasión se ha decidido no implementar nada tan complejo, ya que había que cumplir con los plazos de tiempo establecidos y sin implementarlo se podía conseguir un buen resultado.

Al pulsar la pantalla desde las reglas se empieza a jugar si no hay ningún DMA en marcha. En la pantalla de juego pueden llegar numerosos eventos:

- El jugador toca la pantalla: Si no hay DMA en marcha, se encarga de procesar la jugada y actualiza el tablero en caso de que se haya intentado colocar en una casilla válida. Sea o no sea una casilla válida, actualiza los datos de profiling. También actualiza si la partida debe terminar o no. Se terminaría la partida al intentar mover en el siguiente turno.
- El jugador pulsa un botón: Si no hay transferencias de DMA en activo, mueve la ficha gris de lugar, saltando las que haya en medio. En la implementación se encuentra algún problema con este comportamiento, que se solventa. Se explica más abajo.
- Se debe actualizar el tiempo de juego y no hay DMA en curso. Como este tiempo es en segundos y este debe actualizarse en tiempo real, se calcula de forma diferente a los tiempos de rendimiento y se usa una transición solo para actualizarlo.

Al mostrar la pantalla de fin, se mostrarán las puntuaciones de jugador y máquina, y al volver a pulsar la pantalla se vuelve a la pantalla de reglas, reiniciando el ciclo de una partida.

Aparte del diseño del bucle principal de las mecánicas de juego, se pensó en realizar algunos cambios para mejorar la jugabilidad y la usabilidad. Entre ellos destacan que el jugador no pueda poner ficha en posiciones 'ilegales', es decir, que solo pueda mover si se puede voltear alguna ficha desde esa posición. Otra mejora en la jugabilidad es que la ficha gris salte todas las fichas blancas o negras que se encuentre al moverse en la pantalla, para hacer más rápida la tarea de elegir posición al jugador.

Por último, al principio de cada turno la ficha gris saltará por defecto a la posición {0,0} del tablero, y si esta está ocupada, a la siguiente libre.

Implementación en C del autómata y cambios en reversi8

En la esta sección se detalla la implementación del comportamiento del juego que se acaba de describir, junto con otras implementaciones.

El autómata se ha implementado como el resto de los ya descritos, con variables que actúan de flags para los eventos que llegan, funciones que se llaman desde el tratamiento de eventos en `reversi_main()`, ponen esos flags a 1 y avisan al autómata. Los ficheros siguen siendo `jugada_por_botones.c` y `.h`.

Antes de detallar el código correspondiente al autómata se explican las modificaciones realizadas en el resto del código, para poder acoplar bien el autómata al resto de módulos y que el diseño previo funcione. Estas funciones son utilizadas por el autómata para realizar las funciones que permiten al juego funcionar como debe.

En primer lugar, como las actualizaciones del tiempo de juego en la pantalla deben ser en tiempo real, se utiliza el timer0, que al tratar los eventos del timer0, cada 60 interrupciones se pondrá un flag llamado `ev_timer` a 1. En cuanto esté a 1 y no haya DMA activo, se actualiza el tiempo de juego en pantalla.

```
void tiempo_juego_ev_tick()
{
        cuenta_int_t_juego++;
        if(cuenta_int_t_juego == ticks_segundo_de_juego)
        {
                cuenta_int_t_juego = 0;
                jugada_ev_timer();
        }
}
. . .
case ev_tick_timer0 : //Atender eventos de timer0
                              . . .
                              tiempo_juego_ev_tick();
                              . . .
                              break;
```

Código 28 - Tratamiento en `reversi_main()`

Se iba a tratar el fin de los DMA como un evento más, donde en la interrupción generada al terminar una transferencia se encola un evento `ev_finLCD`, que se trata poniendo a 1 el flag. No obstante, como se requiere la mayor rapidez posible, se desechó esta opción y se optó por colocar una variable **extern** en la cabecera del módulo del lcd, que se aprovechará en el autómata. De esta forma, la interrupción solo cambia el valor de `haciendo_DMA` y el autómata lo verá. Esto fue recomendado por el profesor Darío, cuando se acercó a atender las dudas acerca de la implementación.

```
/*--- define macros---*/
volatile extern int haciendo_DMA;    //Añadido por mí
```

Código 29 - Declaración de la variable como **extern** en `lcd.h`

```
void Zdma0Done(void) {
        rI_ISPC=BIT_ZDMA0;          //clear pending
        //push_debug(ev_finLCD,no_info);
        //Creo que es mejor quitar esto y hacerlo como ha dicho Darío
        haciendo_DMA = 0;
}
```

Código 30 - Interrupción del DMA cuando termina en `lcd.c`

Como ya se ha comentado, se ha hecho que la ficha gris salte las fichas ya colocadas en el juego, pudiéndose colocar solamente sobre casillas vacías. También, cada vez que se coloca una ficha, se devuelve a la gris a la posición {0, 0}. Esto se consigue con las funciones `reiniciar_posición_gris()` y como parte de las funciones que mueven la ficha gris, .No es un comportamiento difícil de implementar, pero en la sección siguiente se verá que puede ser problemático y dará fallos de usabilidad graves. En las funciones que calculan la posición de la ficha gris se implementa el comportamiento **Round-Robin** pedido por los profesores.

```
//Se llama tras hacer un movimiento
//      Devuelve la ficha gris a 0,0 o a la siguiente posición libre del tablero,
//      para hacer la siguiente jugada.
void reiniciar_posicion_gris()
{
        mi_fila = 0;
        mi_columna = 0;
        int i,j;
        int fin_bucle = 0;
        for(i = 0; i < num_filas && fin_bucle == 0; i++)
        {
                for(j = 0; j < num_columnas && fin_bucle == 0; j++)
                {
                        if(tablero_actual[i][j] == CASILLA_VACIA)
                        {
                                mi_fila = i;
                                mi_columna = j;
                                fin_bucle = 1;
                        }
                }
        }
        pintar_ficha(mi_fila, mi_columna, FICHA_GRIS);
}
. . .
//Hace los cálculos necesarios para mover la ficha gris a la derecha en pantalla
void calcular_mov_gris_der()
{
        if((contar_blancas() + contar_negras()) == num_filas * num_columnas)
        //Si no hay que terminar la partida
        {
                contar_fichas_final();
        }
        else
        {
                volatile int aux = mi_fila;
                volatile int aux2 = mi_columna;
                mi_fila = ((mi_fila + 1) % 8);        //Round-Robin, importante
                while((obtener_ficha_en(mi_fila, mi_columna) == FICHA_BLANCA) ||
        (obtener_ficha_en(mi_fila, mi_columna) == FICHA_NEGRA) && mi_fila != aux)
                {
                        mi_fila = ((mi_fila + 1) % 8);
                //8 porque hay 8 columnas en el tablero, [0-7], 0x7 = num_columnas
                }
                //CODIGO QUITADO EN LA MEMORIA PARA EXPLICARLO LUEGO, EN LOS FALLOS//
                while(haciendo_DMA != 0)
                //Espera activa, no se da casi nunca, Dario nos lo aconsejó poner así
                {
                }
                haciendo_DMA = 1;
                //DMA act ficha gris
                mover_gris(mi_fila, mi_columna, aux, aux2); //Mover en la pantalla
        }
}
```

Código 31 - Código para gestionar movimiento de la ficha gris

Para acoplar la lógica del juego con su procesamiento en `reversi8_2019` se ha modificado la función `reversi8()`, que ahora debe procesar el movimiento del jugador, y si no es válido, lo deshace y no calcula el de la CPU. Se apoya en una variable llamada `jugada_valida` que almacena si hay algún patrón para ese movimiento, y en `obtener_jugada_valida()`, que únicamente devuelve el valor de la variable.

```
void reversi8()
{
        ////////////////////////////////////////////////////////////////////
        // Tablero candidatas: se usa para no explorar todas las posiciones
        // sólo se exploran las que están alrededor de las fichas colocadas
        ////////////////////////////////////////////////////////////////////
    jugada_valida = 0;
       veces_pv = 0;
       t_pv = 0;
    if(fin == 0)
    {
        move = 0;
        if(ready)
        {
               ready = 0;
               // si la fila o columna son 8 asumimos que el jugador no puede mover
               if (((fila) != DIM) && ((columna) != DIM))
               {
                       tablero[fila][columna] = FICHA_NEGRA;
                       actualizar_tablero(tablero, fila, columna, FICHA_NEGRA);
                       if(jugada_valida != 1)//Si el movimiento no voltea, es inválido
                       {
                              tablero[fila][columna] = CASILLA_VACIA;
                       }
                       else
                       {
                              actualizar_candidatas(candidatas, fila, columna);
                              move = 1;
                       }
               }
               if(jugada_valida == 1) //Calcular mov. CPU solo si el del jugador OK
               {
                       // escribe el movimiento en las variables globales fila columna
                       done = elegir_mov(candidatas, tablero, &f, &c);
                       if (done == -1)
                       {
                              if (move == 0)
                              {
                                     fin = 1;
                              }
                       }
                       else
                       {
                              tablero[f][c] = FICHA_BLANCA;
                              actualizar_tablero(tablero, f, c, FICHA_BLANCA);
                              actualizar_candidatas(candidatas, f, c);
                       }
               }
        }
    }
    contar(tablero, &blancas, &negras);
}
```

Código 32 - Aspecto final de `reversi8()`

En el autómata, si la jugada no es válida no habrá que actualizar las fichas y solo el profiling.

También, para poder tener la misma versión del tablero en los cálculos y en el módulo del autómata se ha implementado en `reversi8` una función que devuelve el tablero que maneja el juego. Se llama `obtener_tablero()` y con ella es más fácil actualizar el tablero. Se almacenan dos tableros: el del turno actual y el del anterior.

Gracias a esto, para actualizar por pantalla desde el autómata, tenemos `tablero_actual` y `tablero_anterior` con el mismo valor, calculamos en `reversi8()` el nuevo tablero resultante de mover nosotros y la CPU, y lo guardamos en `tablero_actual` con `obtener_tablero()`. Así, se comprueba qué posiciones cambian y solo se actualizan estas, permitiendo minimizar las escrituras en la pantalla virtual y **gastando menos ciclos de CPU**.

```
/* TABLEROS PARA ACTUALIZAR SOLO LAS FICHAS NECESARIAS */
static char tablero_actual[num_filas][num_columnas];
```

```
//Aquí se guardarán las fichas después de que la cpu haga su movimiento
static char tablero_anterior[num_filas][num_columnas];
//Aquí se guardarán las fichas antes de que la cpu haga su movimiento
//Se hará la diferencia, y solo se pintarán aquellas que cambien.
//También, después de esto, anterior = actual, para el próximo turno.
. . .
void actualizar_movimientos_pantalla()
{
        obtener_tablero(tablero_actual);
        int i,j;
        for(i = 0; i < num_filas; i++)
        {
                for(j = 0; j < num_columnas; j++)
                {
                        if(tablero_actual[i][j] != tablero_anterior[i][j])
                        {       //Si algo ha cambiado al mover lo actualizamos
                                borrar_ficha(i,j); //En elementos_pantalla.c
                                pintar_ficha(i, j, tablero_actual[i][j]);
                                //Y de paso actualizamos esto para la siguiente vez
                                tablero_anterior[i][j] = tablero_actual[i][j];
                        }
                }
        }
}
```

Código 33 - Funciones y variables para actualizar las fichas eficientemente

En el autómata se distinguen tres pantallas de fin, una para cada resultado posible (victoria, derrota o empate) Por esto, se ha añadido al autómata un enum con los posibles resultados. Además, para calcular en reversi8 el final obtenido, se utilizaba un flag, 0 o 1. Ahora se ha añadido otra función que devuelve directamente un elemento del tipo del enum para que el autómata lo obtenga más fácilmente. Como este enum, junto con alguna otra variable (dimensiones del tablero) son utilizadas por varios módulos (`reversi8`, `jugada_por_botones`, `elementos_pantalla`), se han colocado en un fichero de cabecera propio llamado `definiciones_juego.h`, para no tener que redefinir en cada sitio, dando modularidad al código.

```
enum estado_casilla{          //Antes estaba en reversi8.c
        CASILLA_VACIA = 0,
        FICHA_BLANCA = 1,
        FICHA_NEGRA = 2,
        FICHA_GRIS = 3
        };
/* Variables para representar en pantalla,
 *  antes en elementos_pantalla.c, las muevo de sitio */
enum{num_filas = 8, num_columnas = 8};
/* Antes en jugada_por_botones.h, lo muevo de sitio */
enum final_partida{no_fin, jugador_gana, cpu_gana, empate};
```

Código 34 - Variables comunes a varios módulos del juego, en `definiciones_juego.h`

En cada pantalla de fin se muestran las fichas de cada jugador, por lo que se ha aprovechado que se lleva la cuenta de blancas y negras por separado, y se han desarrollado dos sencillas funciones en `reversi8_2019` que permiten al autómata obtener los valores.

```
//Devuelven el número de fichas de cada color al final de la partida
//      y solo se cuenta al final, por tanto no hace falta volver a contar
//      Si se quiere contar a mitad de partida, se tendría que contar antes
//      de devolver el número.
int contar_blancas(void){return blancas;}
int contar_negras(void){return negras;
```

Código 35 - Detalle de lo añadido a `reversi8_2019.c`

```
void pintar_fin()
{
	if(final == jugador_gana)
	{
		pintar_fin_victoria(contar_blancas(), contar_negras());
	}
	else if(final == cpu_gana)
	{
		pintar_fin_derrota(contar_blancas(), contar_negras());
	}
	else	//En caso de empate
	{
		pintar_fin_empate(contar_blancas(), contar_negras());
	}
}
```

Código 36 - Función del autómata que usa las definiciones comunes y cuenta fichas por color

Ahora que se han explicado la mayoría de los detalles del diseño del juego, se muestra el código del autómata, contenido en la función `jugada_por_botones`.

```
//El autómata de estados como tal
void jugada_por_botones()
{
	switch(jugada_botones)
	{
	case Reglas:
		if(ev_bot_der > 0)
		{
			ev_bot_der = 0;
		}
		if(ev_bot_izq > 0)
		{
			ev_bot_izq = 0;
		}
		if(ev_tp == 1) //Ignora botones y solo cambia de pantalla con tsp
		{
			ev_tp = 0;
			if(haciendo_DMA == 0)
			{
				haciendo_DMA = 1;
				jugada_botones = Jugando;
				pintar_jugando();
			}
		}
		break;
	case Jugando:
		if(haciendo_DMA == 0 && final != no_fin)
		{
			haciendo_DMA = 1;
			pintar_fin();
			jugada_botones = Fin;
		}
		if(haciendo_DMA == 0 && ev_timer > 0)
		{
			ev_timer--;
			//Actualizar tiempo total
			tiempo_total++;
			//DMA act pantalla con nuevos tiempos
			pintar_profiling(tiempo_total,tiempo_calc, tiempo_pv,veces_pv);
			haciendo_DMA = 1;
			iniciar_DMA();
		}
		if(ev_bot_der == 1)		//Incrementar fila
		{
			ev_bot_der = 0;
			calcular_mov_gris_der();
		}
		if(ev_bot_izq == 1)		//Incrementar columna
		{
			ev_bot_izq = 0;
			calcular_mov_gris_izq();
		}
		if(ev_tp == 1)
```

```
                {
                        ev_tp = 0;
                        if(haciendo_DMA == 0){
                                procesar_jugada();
                        }
                }
                break;
        default:        //case Fin
                if(haciendo_DMA == 0 && ev_tp == 1)
                //Reiniciar partida, restaurando el estado inicial del autómata
                {
                        Lcd_Clr();
                        inicializar_jugada_botones();
                }
                break;
        }
}
```

Código 37 - Implementación del autómata de `jugada_por_botones`

Tanto el proceso de diseño e implementación como el código de este apartado son muy extensos y se han tratado de explicar lo mejor posible y en un orden lo más lógico posible para que se comprenda el trabajo realizado para unir todo y lograr que el juego funcione lo mejor posible. También se han incluido los fragmentos del código más representativos, pero si no es suficiente o no se entiende, se puede consultar el código entregado al completo.

Problemas encontrados al probar y solución

Al probar una versión casi final del juego corriendo sobre la placa se detectaron algunos fallos. Estos fallos eran poco comunes, pero si se daban no se podía seguir jugando en algunos casos. Otros solo eran valores erróneos que se corregían rápido. La mayoría de errores fueron cosas pequeñas o se fueron arreglando según se detectaban, pero hay dos en concreto en los cuales se tuvo que revisar el código exhaustivamente para dar con ellos.

El primero tenía que ver con el comportamiento de la ficha gris. Como salta todas las fichas de la misma fila o columna, puede darse la situación donde toda la fila esté llena menos una casilla. En ese caso, la ficha gris no se mueve. Aunque este comportamiento no es peligroso, si también pasa algo parecido con la columna se podría dar que hubiera casillas libres donde colocar una ficha negra, pero la ficha gris de selección se quedara aislada en esa fila y columna y no pudiera alcanzar ninguna de estas posiciones.

Se optó por una solución sencilla: Si la fila (movimiento horizontal) o la columna (movimiento vertical) de la ficha gris están llenas, la ficha gris se transportará hasta la posición más cercana al final del tablero {8,8}. De esta forma, se garantiza que la ficha gris no se “atascará” nunca.

```
if(mi_fila == aux)//Si ficha gris está atrapada(VER APARTADO DE FALLOS)
{
        volatile int i;
        volatile int j;
        for(i = 0; i < num_filas; i++)
        //se recorre el tablero buscando otra posición libre y
        {
        // se mueve a esa casilla libre para ver si se puede 'liberar'
                for(j = 0; j < num_columnas; j++)
                {
                        if(tablero_actual[i][j] == CASILLA_VACIA)
                        {
                                mi_fila = i;
                                mi_columna = j;
                        }
                }
        }
}
```

Código 38 - Fragmento de actualizar_movimientos_pantalla(), evita bloqueos de la ficha gris

El otro fallo era menor, pero mostraba cosas incorrectas por pantalla y se ha modificado reversi8_2019.c para corregirlo. Al terminar una partida nada más empezarla, la pantalla muestra el número de fichas de cada color, pero mostraba valores incongruentes.

Al analizar el código se ha comprobado que la cuenta de fichas de cada color no se actualizaba hasta que se procesaba una jugada. La solución fue simplemente añadir una invocación a `contar()` durante la inicialización del tablero, cosa que ocurrirá cada vez que comience una partida nueva. Con esto, los resultados siempre serán correctos.

```
void init_table(char tablero[][DIM], char candidatas[][DIM])
{
	. . .
	// Añadido para que se muestre correctamente el número de fichas
	// antes de realizar el primer movimiento
	contar(tablero, &blancas, &negras);
}
```

Código 39 - Detalle de la solución al problema de la cuenta de fichas

6.3.5 Apartado opcional: Uso del teclado

Como apartado opcional, se ha usado el teclado numérico para añadir más controles al juego que mejorarán la jugabilidad. Por ejemplo, a veces es necesario pasar turno porque no se puede mover, y la CPU puede. Se han usado las siguientes teclas:

- `0` para pasar turno y que mueva la CPU.
- `1` para terminar la partida actual.

Estas dos teclas son las dos de arriba a la izquierda del teclado. Para configurarlo, se ha partido del ejemplo proporcionado por el fabricante. Al configurar las interrupciones, y al igual que en los demás dispositivos, se han reconfigurado los registros de tal forma que sólo se modifique la línea que corresponde con el dispositivo configurado. En este caso es el `EINT 1`. También se ha redefinido la rutina de interrupción, de forma que apile un evento en la pila de depuración cuando se pulse alguna de las teclas que se utilizan. En el código 40 se muestra la inicialización del teclado.

```
void init_keyboard()
{
	/* enable interrupt */
	rINTMOD &= ~(BIT_EINT1);	// Configura las linas de TSP como de tipo IRQ
	rINTCON &= 0x1;			// Habilita int. Vect. y linea IRQ (no toca FIQ)
	rI_ISPC |= BIT_EINT1;		// clear pending bit

	/* set EINT1 interrupt handler */
	rINTMSK &= ~(BIT_EINT1);
	pISR_EINT1 = (int)KeyboardInt;

	 /* PORT G */
	rPCONG  |= 0x000c;		// Establece la funcion del pin 1
	rPUPG   &= 0xfd;		// Habilita el "pull up" del pin 1
	rEXTINT &= 0xfffffff7;		//
	rEXTINT |= 0x00000020;		// Configura la línea de interrupción del teclado
					// para que se active en el flanco de bajada
	/* Por precaucion, se vuelven a borrar los bits de INTPND y EXTINTPND */
	rEXTINTPND = 0xf;		// clear EXTINTPND reg
}
```

Código 40 – Configuración del teclado matricial

Físicamente, el teclado está formado por un array de líneas horizontales y verticales junto con 16 interruptores (SB1-SB16), que al ser pulsados conectan una línea horizontal con una vertical. Una vez que se produce la interrupción, para detectar qué tecla se ha pulsado, se utiliza la técnica del

Scanning. Consiste en enviar una tensión baja por una línea horizontal y una alta por el resto de líneas horizontales. Cuando una línea vertical tome tensión baja, la tecla pulsada será la que se encuentre en la intersección entre esa línea vertical y la línea horizontal con tensión baja.

Para evitar rebotes en los botones del teclado, se ha adaptado el código usado para la pantalla anti-rebotes, de forma que una vez que se ha detectado una pulsación, se deshabilitan las interrupciones asociadas al teclado hasta que pase un tiempo. Este tiempo es necesario ajustarlo de la placa utilizada, pero por defecto, se espera durante 0.5 segundos (30 ticks).

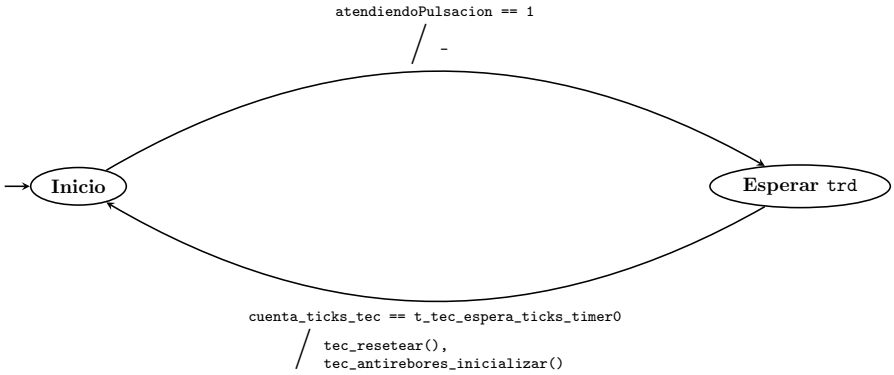

Figura 10 - Autómata de estados para el filtrado de rebotes el teclado

Implementación en lenguaje C

Para añadir las dos nuevas funcionalidades al juego, se ha añadido un nuevo evento que se encolará desde la interrupción del teclado, `ev_keyboard`, y como campo de información puede decir que es la tecla 0 o la 1. En función de la que sea, se tratará poniendo un flag u otro del autómata a 1:

- Si es la tecla 0, se debe pasar turno. Se comprueba que el autómata esté jugando, y de ser así, actúa de forma muy parecida a cuando se toca la pantalla táctil, pero no tiene en cuenta al jugador, solo a la CPU.
- Si es la tecla 1, cuenta las fichas de cada jugador, y en función de ello asigna a fin un valor. Se mostrará la pantalla de fin correspondiente.

```
void jugada_ev_tecla0(void)          //Pasar turno de jugador
{
        if(jugada_botones == Jugando)
        {
                unsigned int delta1 = timer2_leer();
                mover_IA();          //Definida en reversi8.c
                unsigned int delta2 = timer2_leer();
                tiempo_calc += (delta2-delta1);
                . . .
                final = obtener_fin();
        }
}
void contar_fichas_final()    //Qué final poner, si tablero lleno o se aborta partida
{
        int blancas = contar_blancas();
        int negras = contar_negras();
        if(blancas > negras){
                final = cpu_gana;
        }
        else if(blancas < negras){
```

```
                final = jugador_gana;
        }
        else { //En caso de empate
                final = empate;
        }
}
void jugada_ev_tecla1() { //Finalizar la partida de forma prematura
        contar_fichas_final();
        jugada_por_botones();
}
```

Código 41 - Comportamiento de las teclas 0 y 1 en `jugada_por_botones`

En este fragmento de código aparece una nueva función, `mover_IA()`. Está definida en `reversi8_2019.c` y es idéntica a la segunda mitad de la función principal, `reversi8()` por lo que no merece la pena detallar su implementación.

6.4 Plataforma autónoma. Flasheado de la placa con el juego final

Para poder jugar en la placa de forma autónoma, sin que tenga que estar conectada al entorno de desarrollo, se debe cargar el programa en la memoria Flash. Además, es necesario que cada vez que se encienda la placa copie el programa en la memoria RAM, para que las escrituras y lecturas funcionen correctamente tal y como se ha configurado el linker-script.

Se ha modificado el fichero 44binit.asm para gestionar el copiado de los datos en memoria RAM y lanzar el programa. El objetivo es que este proceso se realice cuando se genera una interrupción Reset, así que el nuevo código se ha añadido justo después de desactivar las interrupciones en subrutina `ResetHandler`. En el código 42 se muestra el principio de la rutina de interrupción modificado.

```
#*****************************************************
#*      START                                        *
#*****************************************************
ResetHandler:
    ldr     r0,=WTCON         /* watch dog disable*/
    ldr     r1,=0x0
    str     r1,[r0]

    ldr     r0,=INTMSK
    ldr     r1,=0x07ffffff    /* all interrupt disable */
    str     r1,[r0]

#*****************************************************
#*      Set memory control registers                 *
#*****************************************************

// Eliminado
// Aquí estaba el anterior código de inicialización del controlador memoria

/*********** inicio añadido */

    /* RAM es resuelto por el enlazador como si el programa comenzase en la dirección
       0x0C000000, pero lo cargamos en la flash a partir de la 0x00000000  */
    ldr     r0,=(SMRDATA-0xc000000)
    ldmia   r0,{r1-r13}
    /* establecer valores de los registros del controlador de memoria */
    ldr     r0,=0x01c80000    /* BWSCON Address */
    stmia   r0,{r1-r13}

        LDR r0,=0x0
        LDR r1,=Image_RO_Base
        LDR r3,=Image_ZI_Base

    /* Copiar todo lo que hay desde el comienzo de la flash a la memoria RAM */
    /* El bucle termina cuando la posición destino conincide con Image_ZI_Limit */
```

```
    LoopRw:
        cmp         r1, r3
        ldrcc       r2, [r0], #4
        strcc       r2, [r1], #4
        bcc         LoopRw

    /* código nuevo (Darío) */
            LDR r0, =Image_ZI_Base
            LDR r1, =Image_ZI_Limit
            mov r3, #0
    LoopZI:
            cmp r0, r1
            strcc r3, [r0], #4
            bcc LoopZI
    /* fin código nuevo (Darío) */

/*********** fin añadido */

    #*************************************************
    #*  Set clock control registers                  *
    #*************************************************
```

Código 42 - Rutina de servicio para interrupción Reset

El código añadido sustituye a las instrucciones de inicialización de los registros de memoria. Este código inicializa dichos registros de acuerdo con la configuración del linker-script. Después, realiza un bucle que copia en la RAM la sección de código y variables incializadas del programa, y por último otro que copia las variables no inicializadas. Estas secciones dentro del programa se han definido previamente en el linker-script.

La rutina termina saltando a la función `Main()`, definida en el fichero `main.c`, que previamente se había importado en el fichero ensamblador con la instrucción: `.extern Main`

Una vez realizados estos ajustes, ya se puede recompilar el proyecto y flashear en la placa. Los pasos son los siguientes:[1]

1. Compilar el proyecto y localizar el fichero `P3_PH.elf` en el directorio Debug.
2. Ejecutar el siguiente comando para generar un binario a partir del `.elf`:
   `arm-none-eabi-objcopy -O binary <ruta a P3_PH.elf> P3_PH.bin`
3. El comando se encuentra en el directorio `C:\Programas-Practicas-ISA\EclipseARM\sourcery-g++-lite-arm-2011.03\bin`.
4. Conectar la placa al ordenador.
5. Ejecutar el siguiente comando para copiar el binario en la memoria Flash de la placa:
   `openocd-0.7.0.exe -f test/arm-fdi-ucm.cfg -c "program <ruta a P3_PH.bin> 0x00000000"`
6. El comando se encuentra en el directorio `C:\Programas-Practicas-ISA\EclipseARM\openocd-0.7.0\bin`.
7. Una vez hecho esto, si se reinicia la placa, se debería cargar el juego y entrar en la función `Main()`.

[1] El Proyecto eclipse con el que se trabaja se llama `P3_PH`. Si tuviera otro nombre, bastaría con cambiarlo en los comandos. Los comandos se ejecutan desde los equipos de prácticas, pero se podría realizar en cualquier otro si se tuvieran los programas necesarios para el flasheo.

7. Resultados

Las prácticas 2 y 3 han resultado en sistema autónomo que permite jugar a Reversi contra la placa de desarrollo. La interacción es visual y sencilla.

Cuando el usuario lanza el juego, sale una pantalla inicial (fig. 11), con las reglas del juego y los controles.

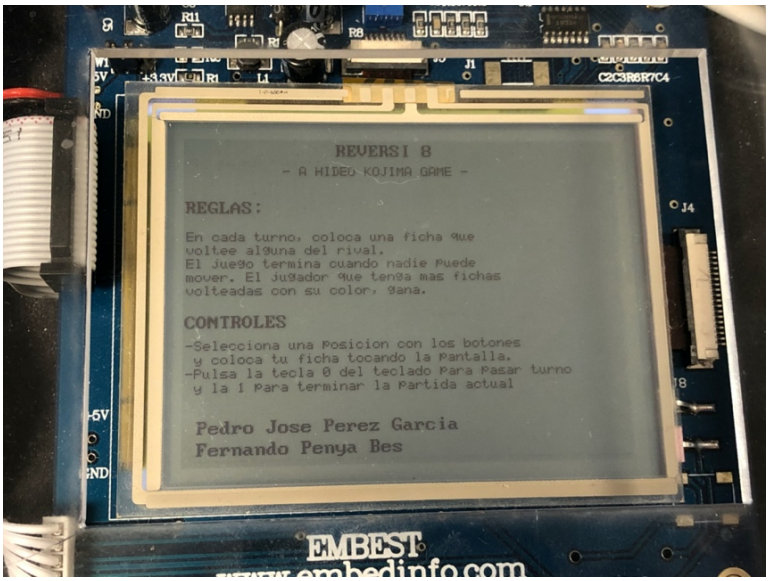

Figura 11 - Pantalla inicial del juego

Cuando el usuario pulsa la pantalla, puede empezar a jugar. El usuario juega con las fichas negras. El usuario puede seleccionar la posición en la que quiere colocar su ficha moviendo la ficha gris que aparece en la pantalla (fig. 12), usando los botones y colocarla tocando la pantalla. La ficha gris se va moviendo por lo huecos libres, de forma que el usuario no pueda colocar su ficha sobre otra. De acuerdo con las reglas del juego, el programa sólo permite al usuario colocar una ficha si voltea alguna del rival.

Una vez colocada la ficha, el programa responde colocando una blanca, pasando de nuevo el turno al usuario.

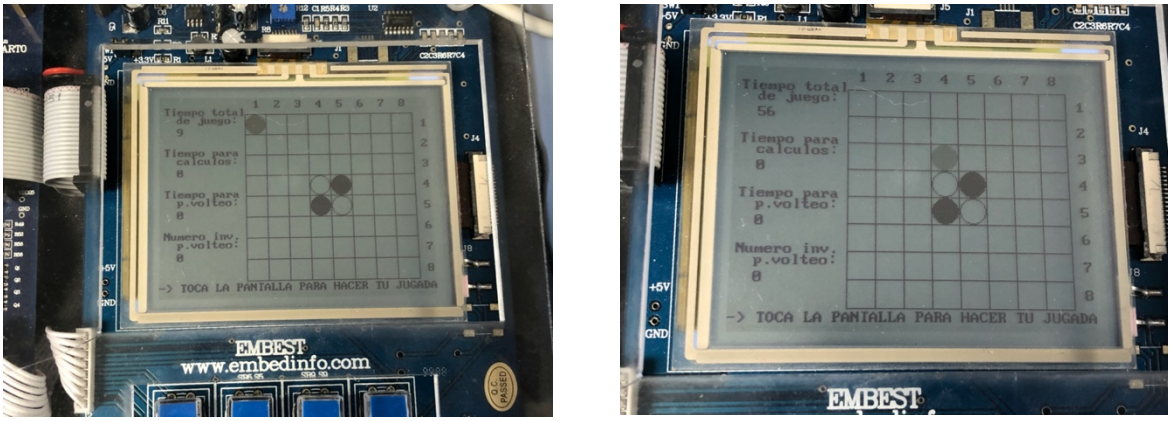

Figura 12 - Movimiento de la ficha gris

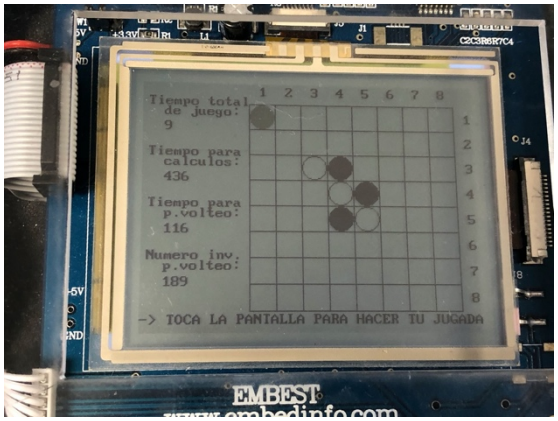

Figura 13 - Respuesta de la placa

La ficha gris aparece por defecto en la primera posición libre del tablero. Si la ficha queda arrinconada, de forma que no se puede mover en la línea actual, salta a la posición libre más cercana en la siguiente línea al pulsar los botones de movimiento (figura 14).

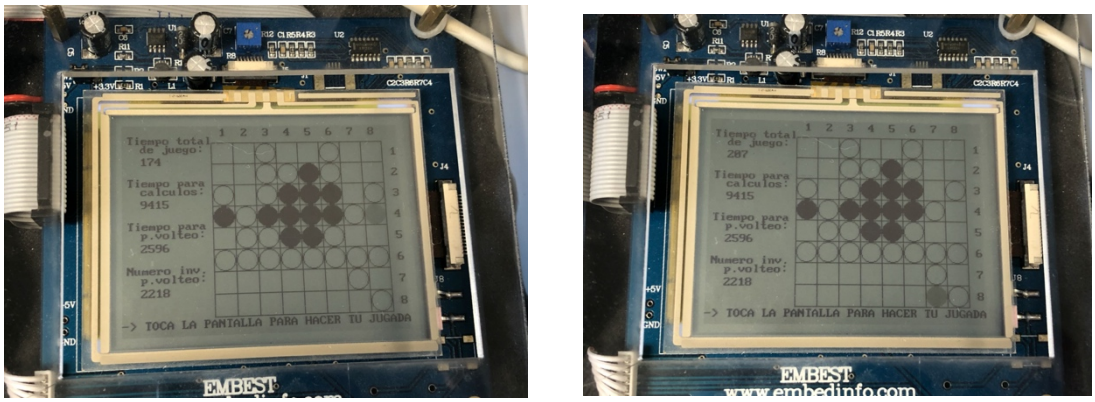

Figura 14 - Movimiento de la ficha gris cuando se encuentra arrinconada

Se ofrece también la posibilidad de pulsar las tecla 0, para pasar turno sin colocar una ficha y 1, para terminar la partida actual.

El juego muestra siempre a la izquierda la información de `profiling`, con el tiempo total de juego, el tiempo total utilizado para los cálculos, el tiempo utilizando en `patron_volteo` y el número de invocaciones a `patron_volteo`.

Cuando la partida termina, muestra al usuario si ha ganado, perdido o empatado, junto al número de fichas colocadas en el tablero (figura 15).

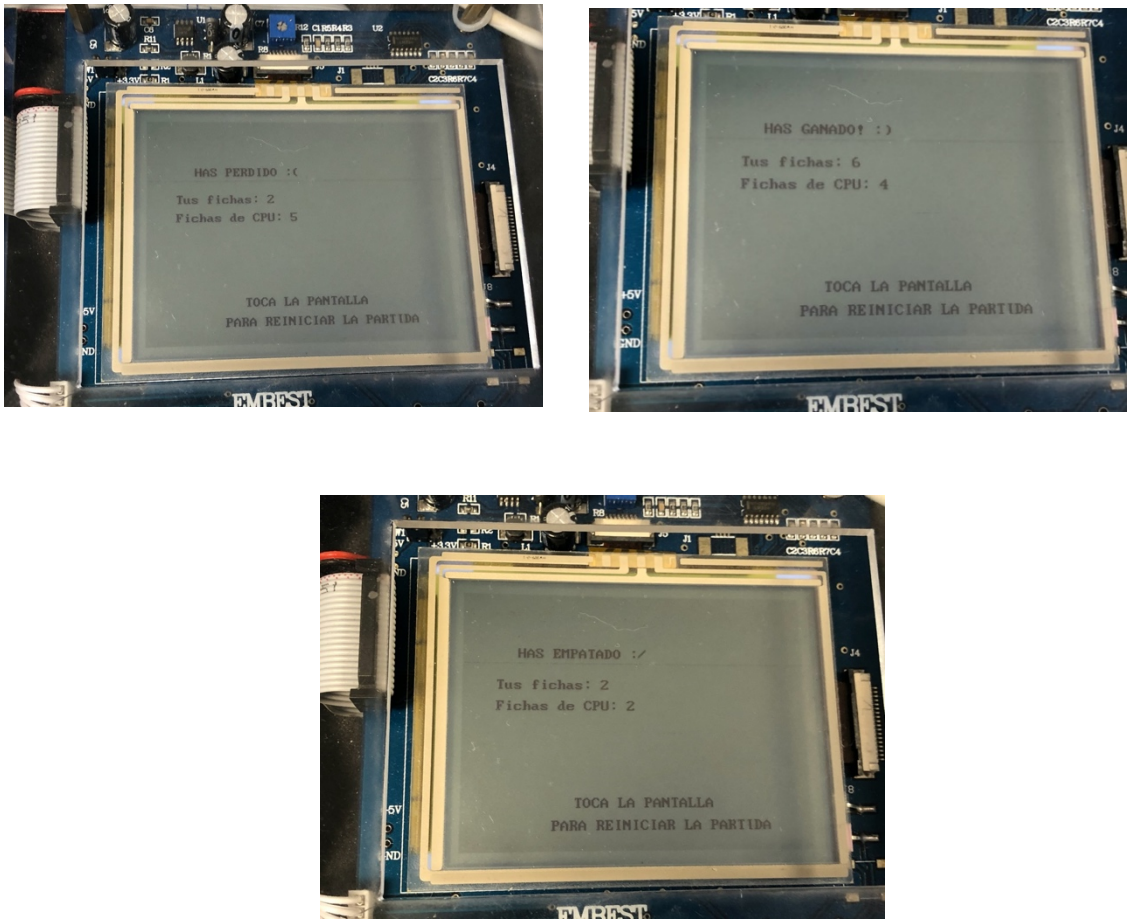

Figura 15 - Pantallas de fin de partida

8. Conclusiones

Gracias a la realización de estas prácticas se ha conseguido una visión más profunda sobre la programación de diferentes dispositivos hardware a bajo nivel para permitir la interacción del usuario con el sistema. Además, se han utilizado técnicas como la eliminación de rebotes para que la experiencia de uso sea satisfactoria.

Ha sido muy importante saber gestionar el tiempo de depuración sobre la placa correctamente, y tener claras las diferentes partes de proyecto creando diagramas y máquinas de estados antes de programar. También hay que destacar la importancia de documentar correctamente el código, para facilitar la tarea de depuración y de puesta en común entre los miembros del equipo.

Aunque el proyecto no ha sido fácil, se ha ganado mucha comprensión en la materia, y como resultado final, se ha construido un sistema con el que un usuario podría interaccionar de forma satisfactoria.

9. Gestión de esfuerzos

En este apartado se incluye una relación aproximada con las horas dedicadas por cada alumno en la realización del proyecto.

- Lectura de documentación: 10 horas
- Realización de diagramas: 5 horas
- Programación: 30 horas
- Depuración: 60 horas
- Redacción de documentación: 20 horas

10. Referencias

[1] GuiaEntorno.pdf

[2] EntradaSalida.pdf

[3] P2-ec.pdf

[4] P3-ec.pdf

[5] um_s3c44box.pdf

[6] S3CEV40_UserGuide.pdf

[7] ARM-Interrupts.pdf

[8] Material de apoyo para la asignatura de Arquitectura y Organización de Computadores 1.

[9] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (consultado: 27/10/2019)

[10]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html (consultado: 27/10/2019)

[11] http://www.scoberlin.de/content/media/http/informatik/gcc_docs/ld_3.html (consultado: 25/11/2019)

[12] Librerías del fabricante de la placa S3CEV40.