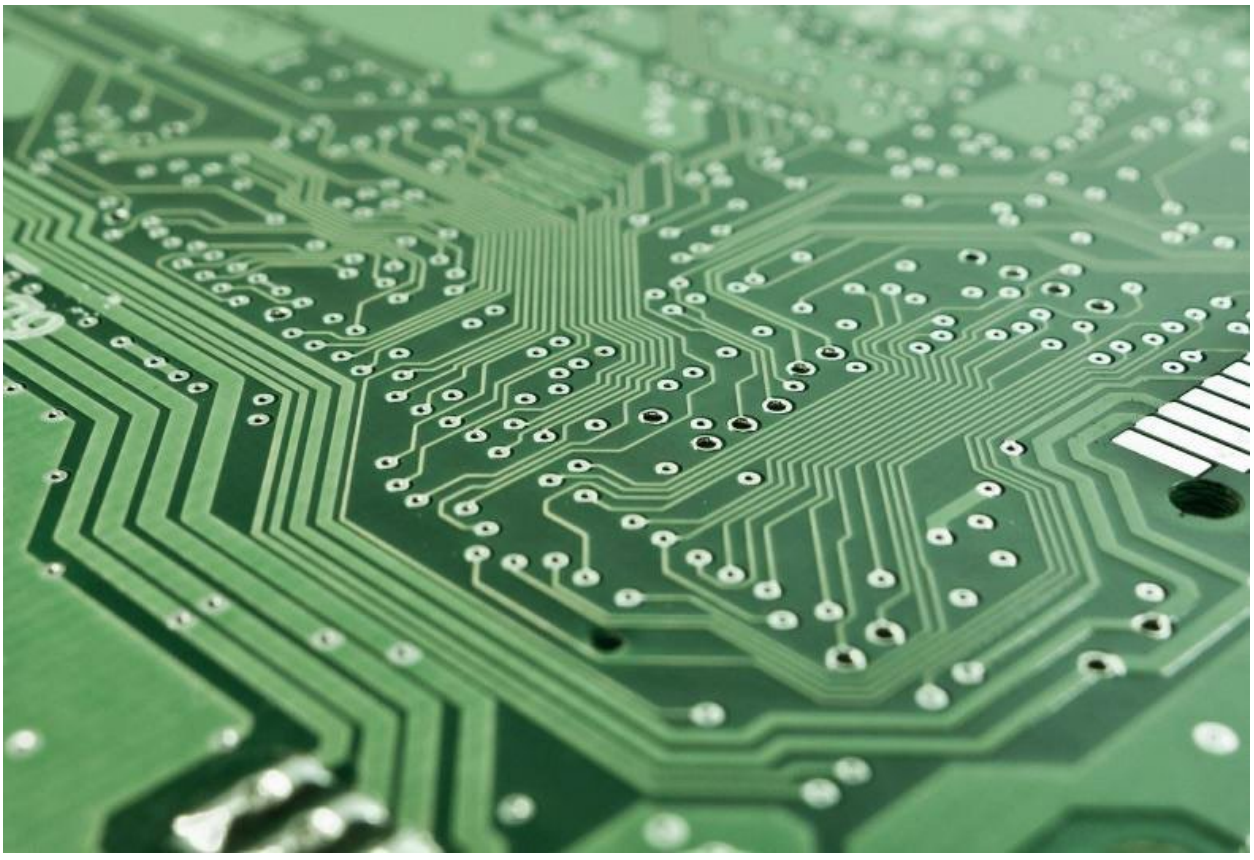
ENTRADA/SALIDA, INTERFAZ GRÁFICA Y PLATAFORMA AUTÓNOMA

Proyecto Hardware: Prácticas 2 y 3



Autores: **Irene Fumanal Lacoma** **758325**

Sergio García Esteban **755844**

Índice

Introducción

Este proyecto se basa en el desarrollo integral del juego Reversi en la placa de prototipado Embest S3CEV40.

El Reversi es un juego de tablero entre dos personas. Cada jugador tiene asignado un color, y por turnos, van colocando fichas en el tablero de tamaño 8x8, intentando voltear las fichas del rival. El ganador es aquel jugador que tiene más fichas colocadas en el tablero al final de la partida.

La placa Embest S3CEV40 es una placa de desarrollo ARM de todo-propósito, que proporciona una interfaz JTAG que nos permite tener control total sobre el CPU para probar nuestro software.

Estudiamos las excepciones, interrupciones y modos de procesador de ARM a fondo, para ser capaces de capturar las interrupciones de los dispositivos de la placa. Desarrollamos una cola circular de depuración para gestionar las interrupciones como eventos concurrentes asíncronos y procesarlos en orden de entrada a la cola.

Entre los dispositivos que proporciona nuestra placa, los elegidos para interaccionar con el usuario son los botones y el touchpad como entrada, y la pantalla y los leds como salida. En este proyecto desarrollamos software de configuración y control de estos dispositivos externos y de dispositivos internos como timers, DMA o A/D conversor.

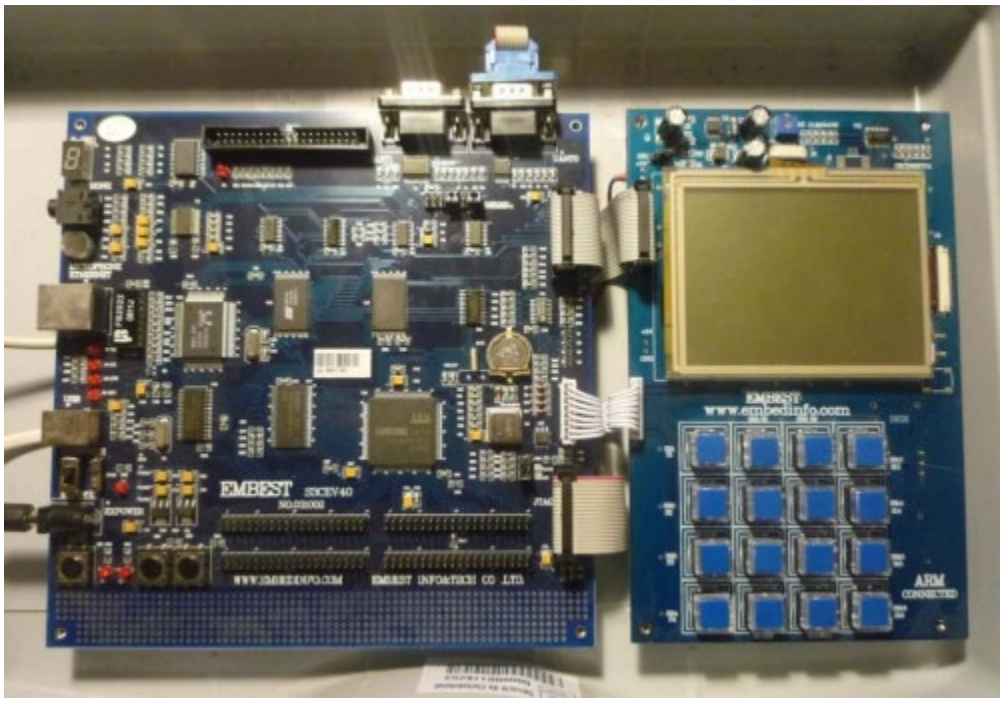
Diseñamos la lógica de nuestro programa de tal forma que reaccione a la interacción del usuario (eventos), procesando una partida de Reversi y generando la salida deseada a través de los leds y la pantalla LCD.

Por último, estudiamos las funciones de inicialización del sistema, el LinkerScript y la memoria Flash-ROM de la placa para hacer autónomo nuestro sistema, cargando nuestro

programa en la placa y pudiendo ejecutarlo sin necesidad de estar conectada a un ordenador.

1. Entrada/Salida

La placa Embest S3CEV40 conecta el microcontrolador S3C44B0X con un conjunto de dispositivos, internos y externos. Todos ellos son accesibles mediante puertos genéricos (GPIO o A/D conversor), interfaces específicas (controlador LCD) o mapeados totalmente o parcialmente en memoria (8 led).



Gestión E/S en C

Para configurar un dispositivo, hay que acceder y escribir en sus registros de configuración. Cada dispositivo tiene sus registros y sus parámetros a configurar.

Algunos dispositivos pueden ser configurados para producir interrupciones (IRQ o FIQ). Al producirse la interrupción, se ejecutará su Rutina de Tratamiento de Interrupción (ISR). La ISR es una función previamente escrita y asignada a ese dispositivo en concreto (p.e. pISR_TIMER0). En la ejecución de la ISR, comunicamos que la interrupción ha sido atendida escribiendo el *bit pending* (I_ISPC o F_ISPC). Además, las interrupciones de este

dispositivo tienen que estar activadas en el registro *INTMSK* y debe estar asignado el tipo de interrupción en el registro *INTMOD (*IRQ o FIQ*)*.

Timer 2

El S3C44B0X dispone de 6 temporizadores de 16 bits y la frecuencia de cada contador se deriva de la del reloj del sistema.

Hemos configurado el Timer 2 con el objetivo de medir tiempos, con la mayor precisión y el mínimo número de interrupciones. Desarrollamos una función (timer2_leer) que nos devuelve el tiempo en microsegundos y garantiza el resultado correcto si llega una interrupción del timer2 durante la lectura.

Timer 0

A parte del Timer 2 para medir tiempos, vamos a necesitar un Timer que produzca interrupciones 60 veces por segundo.

Configuración del Timer 0

Para conseguir interrupciones del Timer 0 a 60 Hz hemos configurado los siguientes parámetros.

$$Timer0_{freq(Hz)} = \frac{CPU_{freq(Hz)}}{(\,preescaler + 1) * divider * CUENTA_{INICIAL}}$$

Utilizando divisor 2, hemos seleccionado el menor preescalado que permita obtener 60 Hz con un valor de CUENTA_INICIAL menor a 2^{16} (65536).

$$60(Hz) = \frac{64000000(Hz)}{(8\ + 1) *\ 2 * 59259}$$

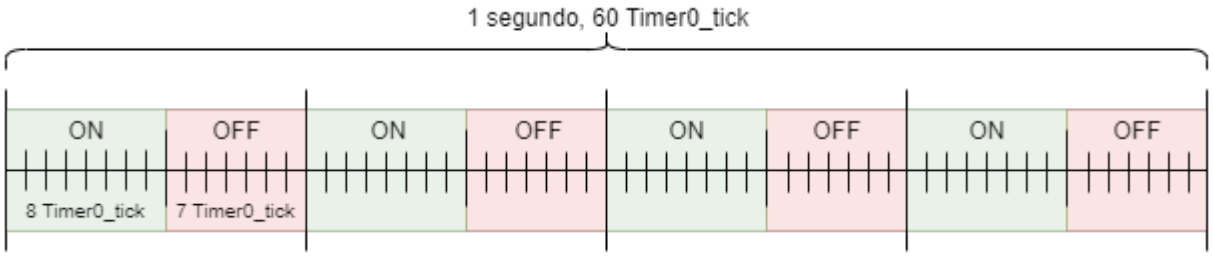
Timer 0 producirá una interrupción cada 16,6 milisegundos (60 Hz).

Función del Timer 0

Este dispositivo genera eventos en nuestra cola de depuración (apartado 3).

La principal función de este Timer es mostrar que el programa sigue vivo. Para ello hemos desarrollado una función que interacciona con los LEDs de la placa, así el usuario podrá ver si el programa sigue vivo (los LEDs están parpadeando).

La función se ejecuta 60 veces por segundo (60 Hz) y queremos que el LED izquierdo parpadee 4 veces por segundo (4 Hz).



Más adelante se podrá utilizar el Timer0 para medir tiempos en múltiplos de 16,6 ms (ev_timer0).

LEDs y 8 LED

Los LEDs están conectados al puerto B del GPIO, en concreto, al pin PB9 el LED izquierdo y al pin PB10 el LED derecho. Como todos los pines del GPIO, tiene su registro de control para configurar la funcionalidad del pin (PCONB[9] = 0 -> PB9 = output) y el registro de datos que permite escribir/leer un valor (PDATB[9] = 0 -> LED izquierdo encendido).

El 8 LED no está conectado al GPIO, está mapeado en memoria, en concreto, en la dirección 0x02140000. En esta dirección se escribirán 8 bits que determinarán que segmentos LED se iluminarán, dibujando un número.

Botones

Los botones están conectados al puerto G del GPIO, en concreto, al pin PG6 el botón izquierdo y al pin PG7 el botón derecho.

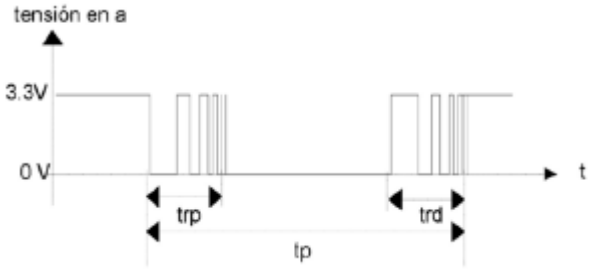
Configuración de los botones

En el registro de control (PCONG) debe ser configurada la funcionalidad EINT para que genere interrupciones en ambos pines. El registro de datos (PDATG) nos permitirá crear una función que identifique si hay un botón pulsado o no y cual.

El puerto G tiene una particularidad con respecto al resto de puertos del GPIO, los pines EINT7654 comparten línea en el controlador de interrupciones. Por lo tanto, el registro EXTINTPND de 4 bits se utilizará para bajar el *bit pending*, además de bajarlo en I_ISPC. También permite mediante el registro EXTINT configurar el modo de señalización de las líneas externas de interrupción.

Rebotes de los botones

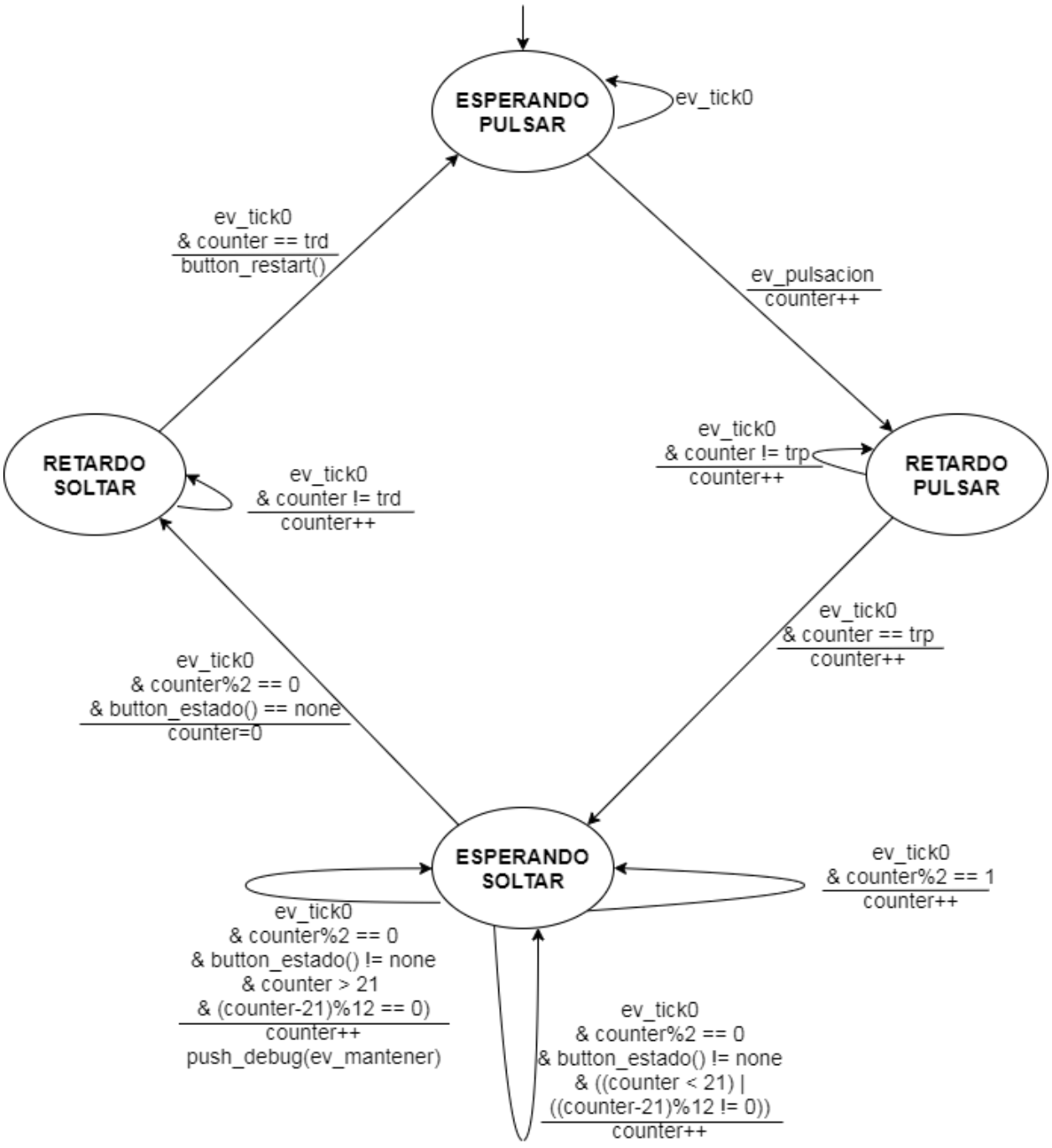
Al pulsar los botones se produce una señal oscilante, causante de que se produzca más de una interrupción en cada pulsación.



Para solucionar este problema, cuando se produce la primera interrupción desactivamos las interrupciones de los botones en la ISR y hemos desarrollado el siguiente software para reactivarlas cuando se hayan cumplido los retardos y se haya soltado el botón.

Hemos utilizado el Timer 0 para gestionar los retardos en múltiplos de 16 ms. También hemos desarrollado una serie de funciones de gestión de los botones para activar y desactivar interrupciones y comprobar si el botón está pulsado.

```
enum{ MS_30 = 2, MS_333 = 21, MS_180 = 12 };     //VALORES PARA TRANSFORMAR DE TIMER0_TICK A MS
enum{ TRP = 12, TRD = 2 };                       //VALORES DE LOS RETARDOS EN TIMER0_TICKS
```



Función de los botones

Este dispositivo genera eventos en nuestra cola de depuración (apartado 3), almacenados desde la ISR.

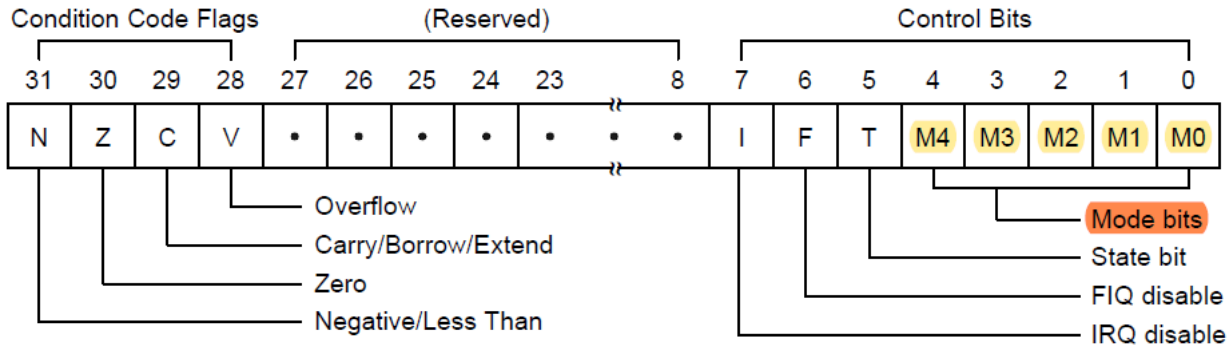
El usuario utilizará los botones para seleccionar fila y columna en el juego y confirmar el movimiento (ev_pulsación).

Como apartado opcional propuesto en el enunciado, los botones se podrán mantener para sumar fila y columna con una sola pulsación (ev_mantener). Una pulsación de más de 1/3 segundo (21 timer0_ticks) irá incrementando el número cada 180ms (12 timer0_ticks). Se puede observar en la figura del apartado anterior.

2. Modos del procesador y excepciones

Modos del procesador

En el CPSR existen 5 bits que determinan el modo de operación. Este modo de operación dicta, entre otras cosas, los registros disponibles para el programador.



Estos son los 7 modos que existen en esta arquitectura. Salta a la vista que 7 modos pueden ser representados con 3 bits, pero se representan con 5 bits por compatibilidad ARM.

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

Excepciones

Cuando una excepción salta, se detiene el flujo principal de ejecución y se ejecuta una parte del código (Handler) previamente asignada a la excepción que salta.

Estas son las excepciones de la arquitectura ordenadas por prioridad.

Highest priority:

1. Reset
2. Data abort
3. FIQ
4. IRQ
5. Prefetch abort

Lowest priority:

6. Undefined Instruction, Software interrupt.

En este proyecto vamos a capturar las siguientes excepciones.

Data Abort. Ocurre al acceder a una dirección de memoria no disponible. Undefined Instruction. Ocurre al intentar ejecutar una instrucción que no puede manejar. Software Interrupt. Ocurre al ejecutar la instrucción SWI, se usa para acceder a modo Supervisor.

Tratamiento de excepciones

Hemos desarrollado una única rutina para capturar las 3 excepciones mencionadas. Para ello hemos asignado la rutina a las 3 excepciones en el vector de excepciones.

La rutina identifica la excepción producida leyendo los bits de modo del CPSR. Si es un SWI, mostrará un 1 parpadeando en el 8led, en el caso de Data Abort o Undefined, se mostrará un 2 o un 3 respectivamente.

Para evitar que el compilador -O3 optimice el retardo, hemos utilizado un atributo en C para que el compilador optimice la función Delay con opción -O0.

```
void Delay(int time) __attribute__((optimize("-O0")));
```

Además de identificar la excepción, identificamos la instrucción en la que se ha producido. En el caso de SWI y Undefined, la dirección de la instrucción causante es la guardada en Link Register (R14). Si la excepción es un Data Abort, a la dirección del LR hay que restarle 8.

	Return Instruction
BL	MOV PC, R14
SWI	MOVS PC, R14_svc
UDEF	MOVS PC, R14_und
FIQ	SUBS PC, R14_fiq, #4
IRQ	SUBS PC, R14_irq, #4
PABT	SUBS PC, R14_abt, #4
DABT	SUBS PC, R14_abt, #8
RESET	NA

Para ejecutar la instrucción siguiente a la causante de la excepción, hay que escribir en el registro PC la dirección de la instrucción causante, tras escribirla se ejecuta automáticamente la instrucción PC=PC+4 y apuntará a la instrucción deseada.

Generación de excepciones

Para comprobar el correcto funcionamiento de la captura de excepciones, hemos utilizado las siguientes instrucciones.

Para generar un Software Interrupt solo tenemos que ejecutar una instrucción SWI.

asm volatile("SWI #0")

Un Data Abort se produce al acceder a una dirección no alineada, en este caso dirección 0x1.

asm volatile("ldr r0, [%1,%2]" :"=r"(x) :"r"(x) ,"r"(y))

Para la Undefined hay que ejecutar una instrucción que no exista. La siguiente está definida en el ARM reference manual:

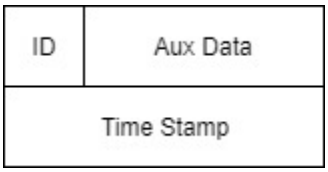
asm volatile(".word 0xe7f000f0\n")

3. Desarrollo de una cola de depuración de eventos

Datos que guarda

La cola está diseñada para guardar eventos. De cada evento guardamos el identificador (ID), datos adicionales (Aux Data) y el instante en el que se ha producido el evento (Time Stamp). Estos tres datos serán implementados en un struct de C.

El *ID* ocupa 8bits, el *Aux Data* 24 bits y el *Time Stamp* es generado por la función timer2_leer() del timer2 y ocupa 32 bits. En total, cada evento ocupa 64 bits (8 Bytes).

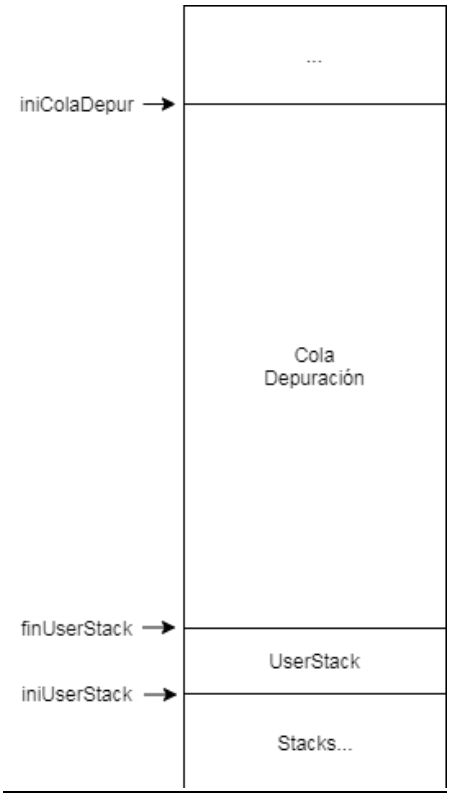


Ubicación en memoria

Queremos que la cola esté ubicada al final del espacio de memoria, antes de las pilas de los distintos modos de usuario. La pila con la dirección menor es la pila de Usuario.

La pila de Usuario empieza en la dirección *_ISR_STARTADDRESS-(0xf00)*, pero es Full Descending y ocupa 256 Bytes, por lo que nuestra cola debe terminar en la dirección *iniUserStack-(256)*.

Hemos implementado una cola circular, con tamaño (configurable) para 256 eventos, 256*8 = 2024 Bytes. La dirección inicial de nuestra cola es *finUserStack-(MAXEVENTOS*8)*.



Función de la cola en el juego

Hemos definido una serie de eventos (**ev_tick0, ev_pulsacion y ev_mantener**), cada uno de ellos es generado por diferentes causas y son tratados de diferente manera. Pero todos ellos son almacenados en la cola de depuración y son tratados por el bucle principal del programa en orden de entrada a la cola de depuración.

```
while (1) {
    while (hay_eventos_encolados()) {
        //procesar eventos
    }
    dormir_procesador();
}
```

En la siguiente imagen se puede observar el comportamiento al detectar los diferentes eventos.

```
if (evento==ev_tick0){
    Latido_ev_new_tick();        //latido funcionamiento
    antirrebotes_ev_tick();      //automata antirrebotes
}else if (evento==ev_pulsacion){
    antirrebotes_ev_boton();     //automata antirrebotes
    jugada_x_botones(info);      //automata juego
}else if (evento==ev_mantener){
    jugada_x_botones(info);      //automata juego
}
```

De esta manera, los autómatas definidos para el funcionamiento del programa (latido, antirrebotes y juego) son alimentados con los eventos definidos como entrada del autómata.

En el caso del autómata *juego*, tiene como primera entrada del autómata ev_pulsacion/ev_mantener, y como segunda entrada del autómata el valor *Aux Data* almacenado junto al evento en la cola, representando el botón pulsado (izdo o dcho).

El *latido* es alimentado por ev_tick0 y el software *antirrebotes* tiene como entrada ev_tick0 o ev_pulsacion.

4. Lógica del juego e interacción con usuario

Interacción mediante E/S

La interacción del usuario con el juego se va a realizar a través de los dispositivos de entrada/salida. Los botones como entrada y el 8led como salida.

Al comenzar la partida y cada vez que se introduzca un movimiento, se mostrará una 'F' en el 8led, lo que significa que el usuario debe introducir la fila. Pulsando el botón izquierdo aumentará en 1 el valor de la fila (entre el 1 y el 8), mostrándose en el 8led, cuando alcance el valor deseado, deberá pulsar el botón derecho para confirmar fila. El 8led mostrará una 'C', y el usuario debe seguir el mismo patrón para introducir la columna.

Si el usuario mantiene pulsado el botón se incrementará el valor repetidamente hasta que deje de pulsar el botón. En el apartado 1.5.3 se detalla cómo se consigue esta funcionalidad.

Reglas del Reversi

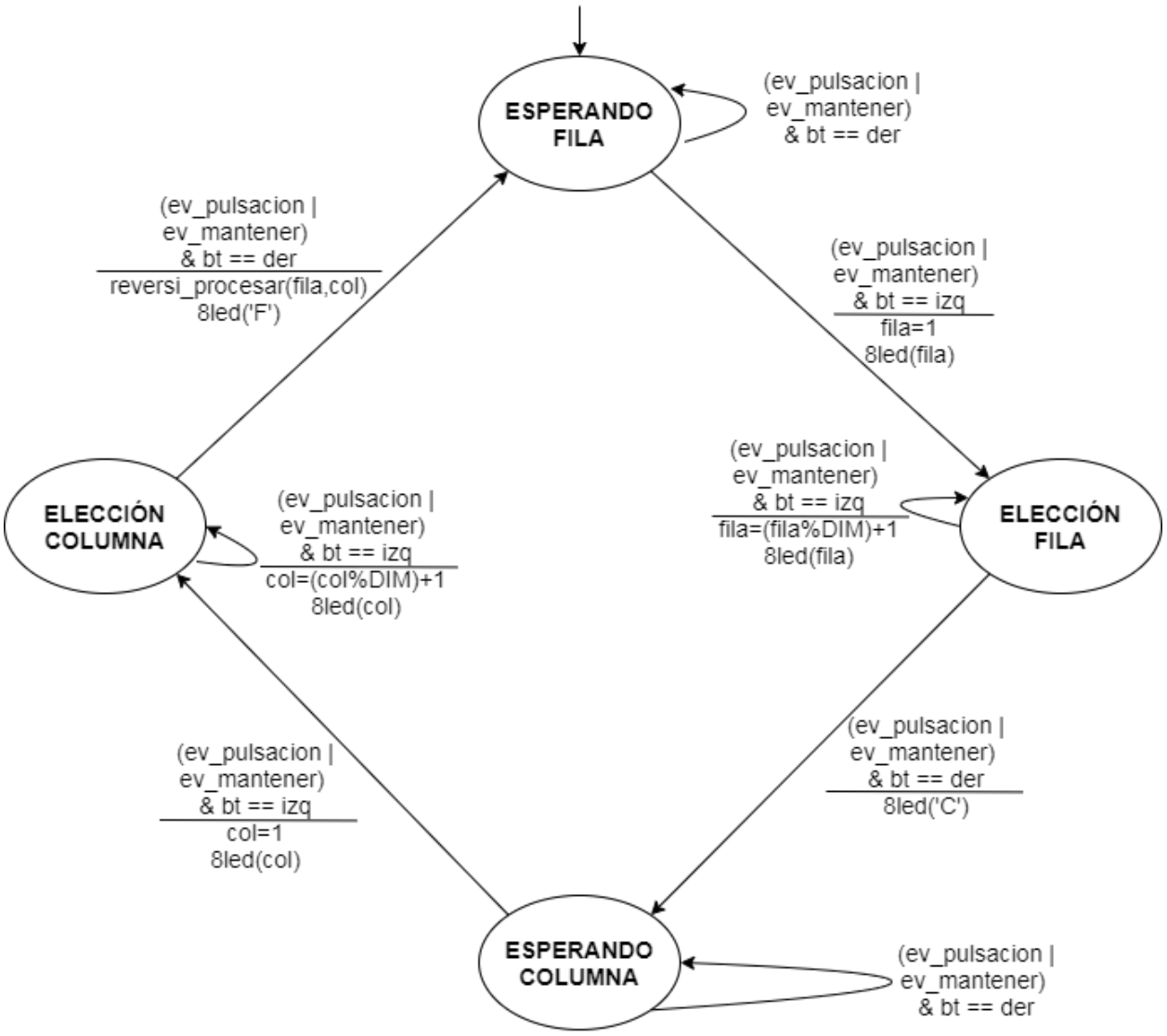
Ya hemos visto en la práctica anterior este juego y desarrollamos una versión propia de la función patron_volteo(). Aunque la versión generada por el compilador -O3 es ligeramente más eficiente que nuestra versión, decidimos utilizar nuestra versión en esta práctica.

Al realizar un movimiento, el usuario puede elegir cualquier posición existente del tablero, pero únicamente se realizará el movimiento si es válido, de lo contrario se pasará su turno. Un movimiento es válido cuando la casilla está libre y hace voltear al menos una ficha del contrincante.

El final de la partida ocurrirá cuando el usuario pase turno y la máquina no tenga movimientos disponibles.

Lógica del juego

El software que gestiona la interacción con el usuario tiene como entradas el evento ev_pulsacion/ev_mantener y el botón pulsado. Se encarga de mostrar por el 8led la salida correta y ejecutar los movimientos. El movimiento de la máquina se ejecuta inmediatamente después del movimiento del usuario.



5. Script de enlazamiento

El script se encarga de combinar objetos y archivos, copia sus datos y vincula referencias de símbolos. Normalmente el último paso en la compilación de un programa es ejecutar este script. Los diferentes segmentos que maneja el script son los siguientes.

.text – la cual guarda las instrucciones en las que consiste el programa. Es marcada como ejecutable y read-only (r-x).

.data – guarda las variables estáticas y globales (las no estáticas se guardan en la pila). Es marcada como read-write y no ejecutable (rw-).

.rodata – guarda las constantes. Es marcada como read-only y no ejecutable (r--).

.bss – guarda las variables no inicializadas e inicializadas a cero. Es marcada como read-write y no ejecutable (rw-).

Problema en inicialización

En el script se declaran variables que delimitan las zonas Read-Only, Read-Write y Zero-Init. En el código de inicialización se utilizan estas variables para copiar la zona Read-Write y para escribir a cero toda la zona Zero-Init. Posteriormente también se utilizarán para copiar de la memoria ROM a la RAM todos los datos.

Para que el código de inicialización funcione correctamente las variables tienen que guardar direcciones alineadas a 4. Esto se consigue utilizando la directiva ". = ALIGN (4)". Para evitar que se escriban ceros donde no debería, incluimos esta directiva para alinear ZI_Limit.

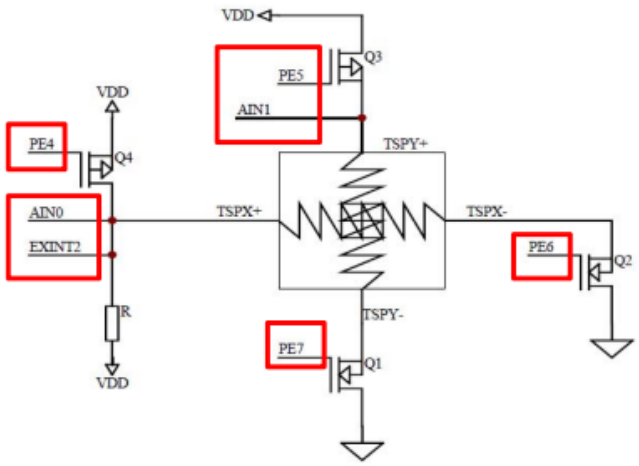
```
. = 0x0C000000;
Image_RO_Base = .;
.text : { *(.text) }
. = ALIGN (4);
Image_RO_Limit = .;

Image_RW_Base = .;
.data : { *(.data) }
.rodata : { *(.rodata) }
Image_RW_Limit = .;

. = ALIGN (4);
Image_ZI_Base = .;
.bss : { *(.bss) }
. = ALIGN (4);
Image_ZI_Limit = .;
```

6. Dotar de interfaz gráfica a nuestro juego

Para implementar una interfaz gráfica en nuestro juego, necesitamos añadir y configurar nuevos dispositivos, además de adaptar a ellos la nueva lógica del juego.

TouchPad y A/D Conversor

El touchpad está conectado a dos puertos diferentes del GPIO. Está conectado a 4 pines del puerto E (**PE[7:4]**) para el control de voltajes. El retorno analógico se hace a través del **conversor A/D**, Ain0 (TSPX+) y Ain1 (TSPY+) son las entradas del conversor, la salida será leída en cada pulsación y transformada en coordenadas de pulsación. También está conectado al pin **PG2** del GPIO (EINT2) para generar una interrupción en cada pulsación.



Configuración del touchpad

Los 4 pines del puerto E los configuramos en PCONE con funcionalidad output. Inicializamos las tensiones en el registro PDATE[7:4] = (1,0,1,1), cada pin está conectado a un transistor MOSFET, lo que permiten dar selectivamente tensión a cada capa.

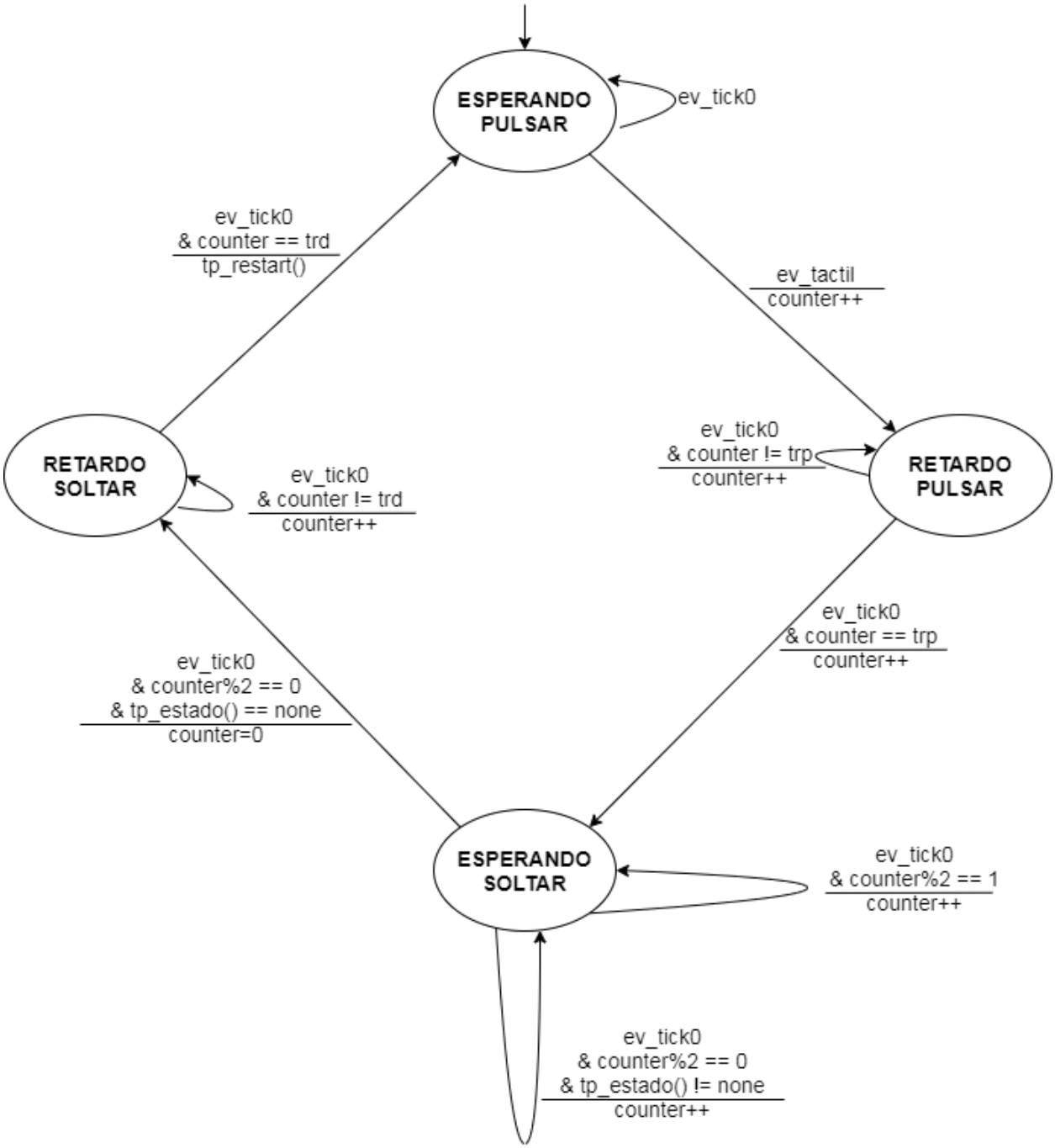
El pin PG2 debe ser configurado para generar interrupciones, para ello escribimos en el registro de configuración PCONG[5:4] la funcionalidad EINT2. Además, configuramos EXTINT[2] como flanco de bajada y PUPG[2] como desconectado.

Para que el A/D conversor funcione correctamente, debemos activar su alimentación CLKCON[12] y configurar su preescalado (19) para conseguir la tasa de conversión máxima (100KHz).

$$100\text{KHz} = 64\ \text{MHz} / 2(\text{n}+1) \times 16 \implies \text{n} = 19$$

Rebotes del touchpad

Al pulsar el touchpad se pueden producir rebotes de la misma forma que ocurría con los botones. La solución es similar, lo primero que se ejecuta en la ISR es la desactivación de las interrupciones y hemos desarrollado el software que reactiva las interrupciones tras un retardo inicial, comprobar que no sigue pulsado y un retardo final.



Lectura de coordenadas

La lectura se realiza dentro de la ISR cuando salta la interrupción.

Primero leemos la posición X, damos tensión a las capas X- e Y+, escribiendo PDATE[7:4] = (0,1,1,0), y seleccionando como entrada del A/D conversor Ain1 escribiendo

ADCCON[4:2] = 001. Para obtener más precisión, calculamos la media de 8 lecturas de la salida del A/D conversor (ADCDAT).

Para obtener la lectura de la posición Y, realizamos los mismos pasos dando tensión a las capas X+ e Y- (1,0,0,1) y seleccionando la entrada Ain0 (000).

Antes de cambiar de canal es necesario esperar un retardo para que las lecturas sean correctas, para evitar que el compilador -O3 optimice el retardo, hemos utilizado un atributo en C para que el compilador optimice la función DelayTime con opción -O0.

```
void DelayTime(int num) __attribute__((optimize("-O0")));
```

Para transformar los valores de salida del A/D conversor en coordenadas, utilizamos unos máximos y mínimos configurados previamente (calibración) y utilizamos la siguiente formula. Previamente nos aseguramos de que el valor leído está contenido entre el mínimo y el máximo, de no ser así se le asigna el mínimo/máximo.

$$X = 320 * (Vx - Xmin)/(Xmax - Xmin)$$

$$Y = 240 * (Vy - Ymin)/(Ymax - Ymin)$$

Calibrado del touchpad

Hemos desarrollado una función *tp_calibrar(int n)* que al ser invocada configura que las próximas n pulsaciones van a ser utilizadas para calibrar la pantalla, asignando nuevos mínimos y máximos.

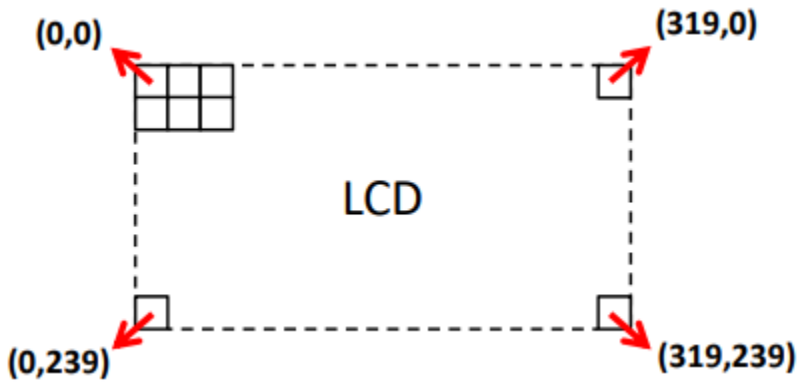
Función del touchpad

El touchpad va a permitir al usuario navegar entre las diferentes pantallas de la interfaz pulsando los botones dibujados en la pantalla. Además, en la pantalla de juego se utilizará para confirmar la posición del movimiento del usuario.

Hemos definido un nuevo evento (**ev_tactil**), tras calcular las coordenadas de la pulsación, el evento es guardado en la cola de depuración, utilizando el campo Aux Data para guardar las coordenadas de la pulsación (12 bits para X y 12 bits para Y).

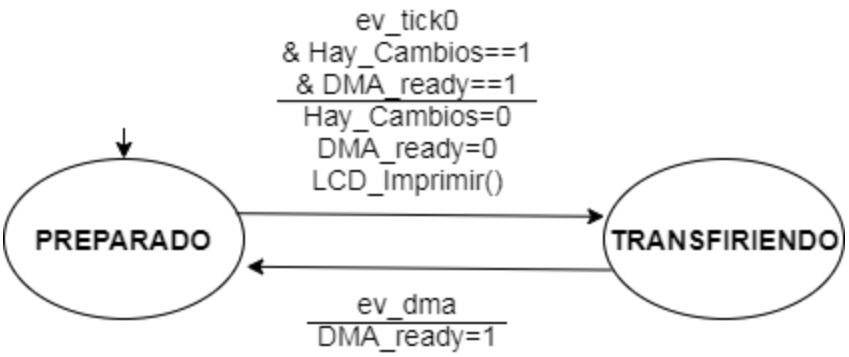
LCD y DMA

La placa S3CEV40 dispone de un LCD conectado al controlador de LCD del microcontrolador a través del puerto D del GPIO. Tiene una resolución de 320 × 240 píxeles. Cada pixel lo lee por DMA de una región de memoria (buffer de vídeo) de ubicación programable. Cada byte almacena 2 pixeles, el tamaño del buffer es 38400B.



Eliminación espera activa DMA

El DMA puede ser configurado para producir una interrupción al finalizar una transferencia, de esta forma podemos seguir procesando otros eventos mientras se realiza la transferencia (pero nunca escribir en el buffer). La siguiente imagen representa el funcionamiento del DMA.



La variable *Hay_Cambios* toma valor 1 por la lógica del juego únicamente cuando es necesario, es decir, cuando hay un cambio de pantalla, cuando hay un cambio en el tablero o cuando hay un cambio en las métricas. El fin de la transferencia se comunica generando un nuevo evento, **ev_dma**.

Pantallas de la interfaz

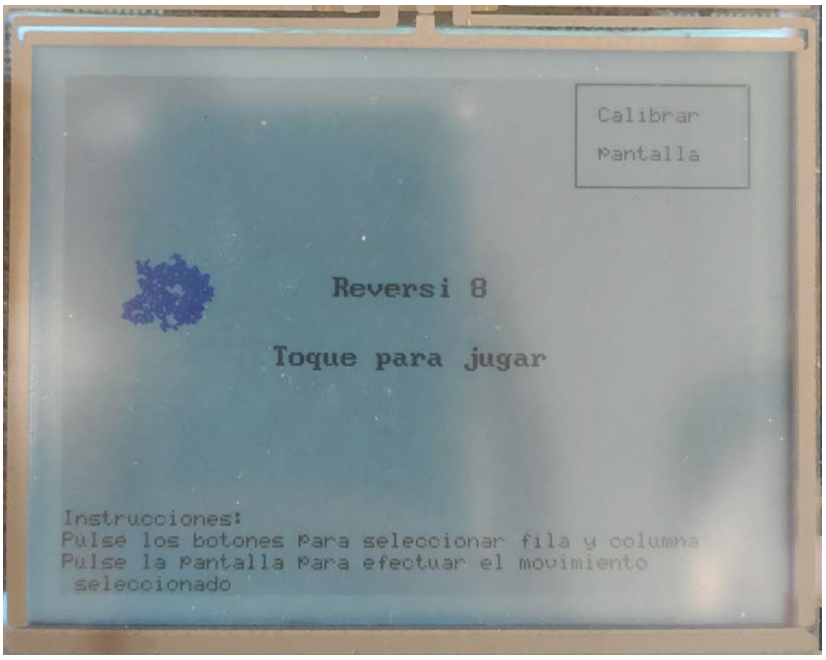
La librería LCD implementa una serie de funciones de dibujado sobre el buffer. Algunas de ellas dibujan líneas, rectángulos o cadenas de caracteres en varios formatos.

Existe la posibilidad de escribir en unas coordenadas determinadas del buffer un dibujo creado previamente (BitMap). Hemos implementado los dibujos de las fichas blanca, negra y gris.

Para dibujar números, utilizamos las funciones de dibujado de cadenas de caracteres y una función implementada en el proyecto para convertir int a char (itoa).

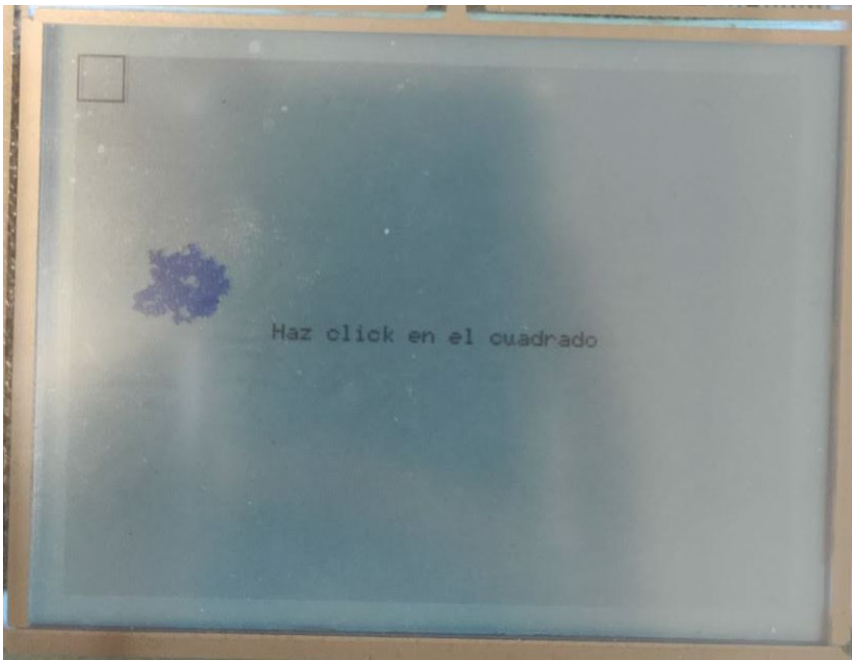
Pantalla inicial

La pantalla principal del juego nos muestra el título “Reversi 8”, la leyenda “Toque para jugar”, las instrucciones de juego y un botón para iniciar el calibrado de pantalla. Si pulsas el botón inicias el calibrado, si pulsas en cualquier otro punto, empieza una nueva partida.



Pantalla de calibrado

En esta pantalla se guía al usuario a pulsar en cada una de las 4 esquinas de la pantalla, mostrando un cuadrado en la esquina que se desea que pulse el usuario (la esquina deseada es gestionada desde la lógica del juego con la función *LCD_esquina(int e)*). Al realizar la cuarta pulsación se muestra la pantalla inicial y la pantalla ha sido calibrada.



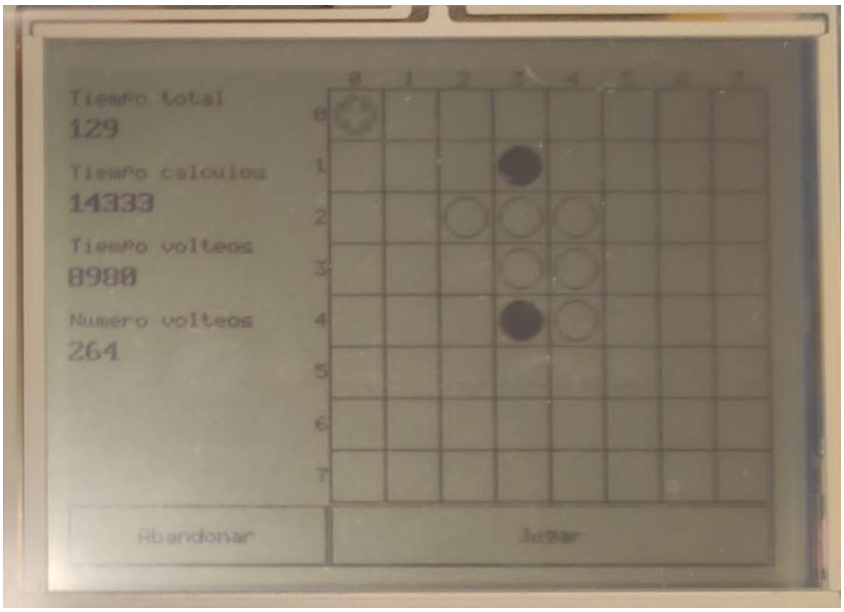
Pantalla de juego

La pantalla de juego muestra el desarrollo de la partida desde su inicio hasta su fin.

A la izquierda mostramos diferentes métricas. El tiempo de volteo y el número de volteos es calculado utilizando la función *LCD_volteo(unsigned int t) tras cada invocación de patron_volteo()*. Para el tiempo de cálculo hemos desarrollado la función LCD_tcalculo(unsigned int t). El tiempo total es calculado utilizando funciones del timer2.

A la derecha se muestra el tablero con índices de fila y columna, las fichas colocadas en el tablero y la ficha gris que marca la posición seleccionada por el usuario para su próximo movimiento. Para mostrar las fichas colocadas hemos desarrollado la función *LCD_tablero(char t[DIM][DIM])* la cual genera una matriz de los cambios a realizar en el siguiente refresco de pantalla, es invocada tras realizar un movimiento desde la lógica del juego. La ficha gris es gestionada también desde la lógica del juego tras la pulsación de un botón, hemos desarrollado la función *LCD_movimiento(int f,int c)* que actualiza la posición de la ficha gris en el siguiente refresco de pantalla.

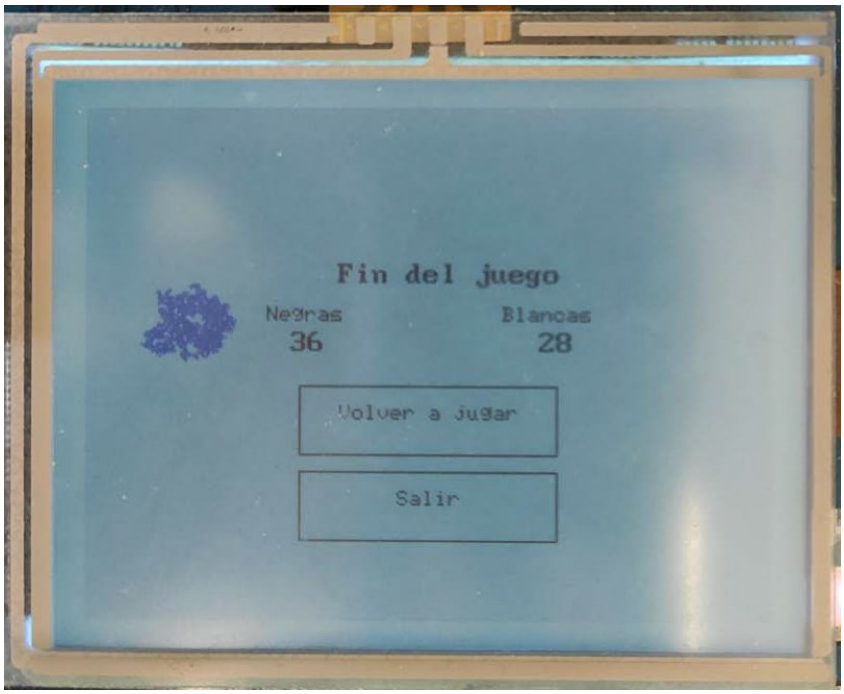
En la parte inferior hemos diseñado 2 botones, para ejecutar el movimiento seleccionado y para abandonar partida (se mostrará la pantalla inicial).



Pantalla final

La pantalla final es mostrada al terminar una partida (el usuario debe pasar turno y la maquina no tener movimientos disponibles). Muestra el resultado de la partida, para ello hemos desarrollado la función *LCD_fichas(int b, int n)*.

Muestra dos botones, el primero te muestra la pantalla de juego con una nueva partida y el segundo te muestra la pantalla inicial.



Lógica final del juego

En la primera versión del proyecto, alimentábamos con eventos el software encargado del latido, de botones antirrebotes y de la jugada. En la versión con interfaz añadimos más bloques software que alimentamos con eventos.

Además, añadimos la interacción entre pantallas a la lógica del juego, y cambiamos la forma de seleccionar el movimiento del usuario.

Bucle principal

Como ya hemos visto, añadimos a los eventos existentes, los eventos **ev_tactil** y **ev_dma**.

El bloque *jugada*, añade como nueva entrada del autómata **ev_tactil**, para gestionar la interacción mediante la pantalla.

El bloque *tp_antirrebotes*, se alimenta de la misma manera que *botones_antirrebotes* de ev_tick0 o ev_tactil como entrada del autómata.

El bloque *DMA_ready*, se encarga de comunicar que el DMA está disponible al procesar un ev_dma.

El bloque *LCD_Refresca_ttotal*, comprueba el estado del juego. Si la pantalla mostrada es la pantalla de juego, activa la variable *Hay_cambios* una vez por segundo, mostrando correctamente actualizada la métrica Tiempo Total de partida. El resto de casos en los que hay que activar la variable *Hay_cambios* están definidos en la lógica del juego (cambio de pantalla y cambio en tablero).

El bloque *LCD_Imprimir*, comprueba si Hay_*cambios* está activada y *DMA_ready* está activada para pintar en el buffer e iniciar una nueva transferencia.

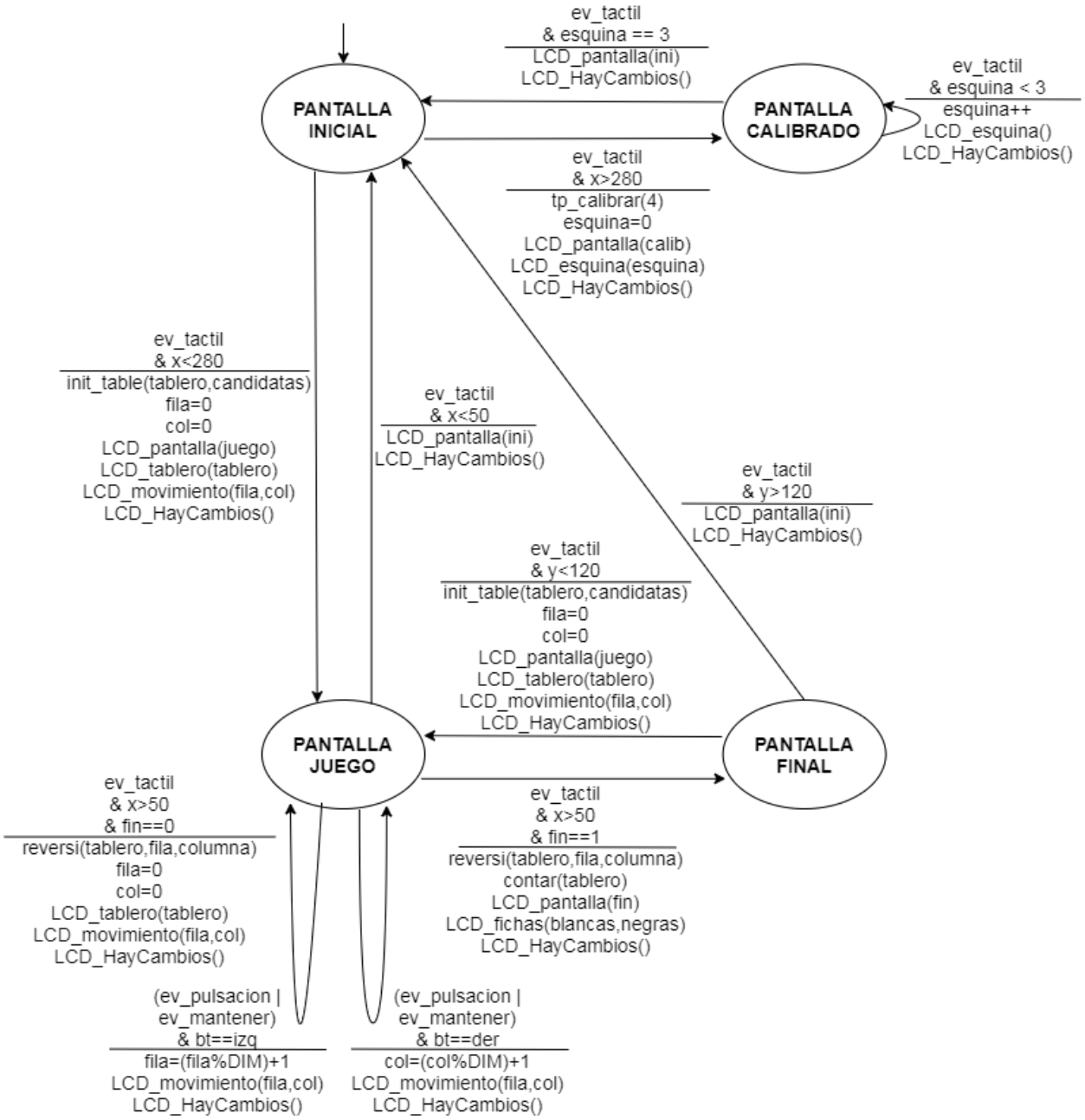
```
if (evento==ev_tick0){//-------------------------------------------------------
    LCD_Refresca_ttotal();          // comprueba si hay que refrescar el ttotal
    LCD_imprimir();                 // IMPRIME pantalla si hay cambios
    Latido_ev_new_tick();           // latido led
    antirrebotes_ev_tick();         // antirrebotes botones
    tp_antirrebotes_ev_tick();      // antirrebotes tactil
}else if (evento==ev_pulsacion){//---------------------------------------------
    antirrebotes_ev_boton();        // antirrebotes botones
    jugada_x_pantalla(evento,info); // AUTÓMATA juego
}else if (evento==ev_mantener){//----------------------------------------------
    jugada_x_pantalla(evento,info); // AUTÓMATA juego
}else if (evento==ev_tactil){//------------------------------------------------
    tp_antirrebotes_ev_tactil();    // antirrebotes tactil
    jugada_x_pantalla(evento,info); // AUTÓMATA juego
}else if (evento==ev_dma){//---------------------------------------------------
    LCD_DMA_ready();                // comunica que dma ha terminado
}
```

Lógica del juego

La versión anterior de la lógica del juego se encargaba de gestionar la interacción del usuario, mediante los botones, con la partida de Reversi. Al añadir interfaz al juego, tiene que gestionar también la interacción del usuario, mediante la pulsación táctil sobre botones dibujados en la pantalla LCD, con las diferentes pantallas a mostrar.

Además, ha cambiado la manera de elegir movimiento, pulsando el botón izquierdo seleccionas la fila y pulsando el botón derecho seleccionas la columna. Sigue estando la opción de mantener el botón para sumar repetidamente el valor y, en todo el proceso

de elección de posición, es mostrado en la pantalla LCD una ficha gris marcando la posición seleccionada.



7. Fast Interrupt request (FIQ)

Las interrupciones, externas e internas, pueden ser gestionadas tanto por IRQ como por FIQ. La principal ventaja de FIQ es que es un tipo de excepción más prioritaria que IRQ. La principal desventaja es que no tiene interrupciones vectorizadas, por lo tanto, habría que implementarlo para identificar la interrupción y saltar a la ISR correspondiente.

En este proyecto únicamente vamos a utilizar interrupciones FIQ para el timer2, asignando la ISR del timer2 como ISR de FIQ, no necesitamos implementar una solución vectorizada.

Para configurar el timer2 como interrupción FIQ, configuramos el bit 11 del registro INTMOD a 1, asignamos la ISR (ISR_FIQ = timer2_ISR) y el bit pending hay que bajarlo en el registro F_ISPC.

8. Ejecución sobre modo Usuario

Al inicio de la ejecución del programa el procesador opera en modo Supervisor y queremos que tras realizar la inicialización de dispositivos en este modo, cambie a modo usuario para ejecutar nuestro juego.

Para cambiar de modo Supervisor a modo Usuario, guardamos el SPSR y modificamos los bits de modo del CPSR (5 bits menos significativos del módulo de control).

```
void modo_usuario(){
    __asm__ volatile("mrs r0, cpsr");
    __asm__ volatile("msr spsr, r0");
    __asm__ volatile("msr cpsr_c, 0x10");
}
```

El código de inicialización del sistema inicializa las pilas de los diferentes modos de operación. Hemos añadido a este código la inicialización de la pila de Usuario. Para hacerlo hay que cambiar de modo y asignar al *StackPointer* la dirección correspondiente.

El modo usuario es el único modo no privilegiado, esto significa que no tiene acceso a algunos registros y no puede acceder a los bits de modo del CPSR por lo que para cambiar de modo Usuario a un modo privilegiado hay que producir una excepción SWI y tratarla correctamente. Por limitaciones de la arquitectura, el modo System y el modo Usuario comparten registros, entre ellos el *StackPointer*, por lo que evitamos acceder a modo Usuario para inicializar su pila y accedemos a modo System para asignar al StackPointer la dirección de la pila de Usuario.

```
orr     r1,r0,#MODEMASK
msr     cpsr_cxsf,r1
ldr     sp,=UserStack
```

9. Modo de ahorro de energía

En el enunciado de la práctica se plantea la siguiente estructura para el bucle principal del proyecto, y se menciona la función dormir_procesador().

```
while (1) {
    while (hay_eventos_encolados()) {
        //procesar eventos
    }
    dormir_procesador();
}
```

Después de investigar la documentación de la placa, descubrimos el modo IDLE.

El modo IDLE desactiva el reloj del núcleo del CPU manteniendo el reloj activo para todos los periféricos, el controlador del bus, el controlador de memoria, el controlador de interrupciones y el bloque de consumo de energía. Usando el modo IDLE se puede reducir el consumo de energía.

Para activar el modo IDLE hay que activar el bit 2 del Clock generator control Register (CLKCON). Al escribir a 1 este bit (IDLE BIT), se activa la transición al modo IDLE. Si el usuario espera utilizar EINT[7:0] para desactivar el modo, deberá activar el bit 10 del CLKCON (activa alimentación del bloque GPIO) antes de activar el modo IDLE.

Para desactivar el modo IDLE tiene que saltar una interrupción y el procesador despertará automáticamente. Se puede utilizar EINT[7:0], una alarma RTC o cualquier otra interrupción.

Para implementarlo en nuestro proyecto, activamos el modo IDLE y el bloque de alimentación del bloque GPIO ya que va a despertarse por interrupciones del timer0, timer2, dma, botones (EINT[7:6]) o touchpad (EINT[2]).

```
void dormir_procesador(){
    //modo de bajo consumo de energía
    rCLKCON |= 0x404; // GPIO + IDLE
}
```

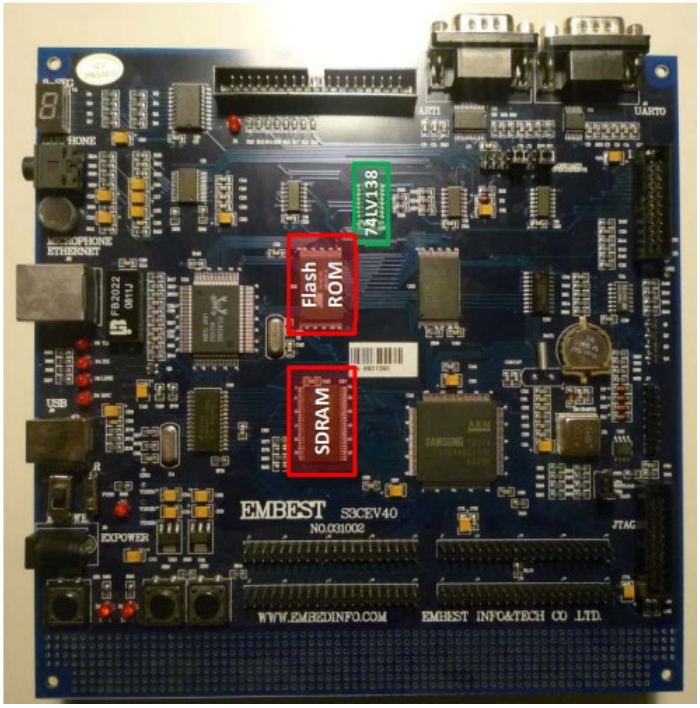
JTAG y modo IDLE

Al depurar desde eclipse utilizando el JTAG, ha funcionado perfectamente nuestro código. Como hemos comentado, se generan interrupciones de timer0, timer2, dma, botones (EINT[7:6]) o touchpad (EINT[2]).

Realizamos una prueba desactivando las interrupciones de timer0, timer2 y dma. Al entrar en dormir_procesador() se pausa el debugger, en este momento si pulsas play, se vuelve a pausa automáticamente. Al pulsar el touchpad o uno de los botones para generar una interrupción, comienza la ejecución, pero se genera una excepción y no continúa la ejecución del programa.

10. Plataforma autónoma

La placa Embest S3CEV40 cuenta con una memoria principal SDRAM y con una memoria ROM-Flash de arranque.



La memoria ROM está conectada al banco 0 y tiene una capacidad de 2MB, su rango de direcciones es 0x00000000 - 0x001FFFFF. La memoria RAM está conectada al banco 6 y tiene una capacidad de 8MB, su rango de direcciones es 0x0C000000 - 0x0C7FFFFF. En los bancos 1 y 3 se encuentran mapeados dispositivos como el teclado matricial, el 8led o el controlador USB.

Hasta ahora, ejecutamos nuestro juego en la placa gracias al dispositivo JTAG conectado al ordenador del laboratorio y queremos que no sea necesario estar conectado a un ordenador para poder ejecutar el juego, para ello tenemos que cargar nuestro juego en la Flash-ROM de la placa, pero antes tenemos que configurar nuestro código de inicialización para ello.

En concreto, debemos modificar el código que se ejecuta al producirse un Reset (ResetHandler). Debemos copiar los valores de configuración del controlador de memoria (SMRDATA) a los registros de configuración del controlador de memoria (BWSCON, BANKCON0…).

Para copiar los valores de la etiqueta SMRDATA, hay que introducir a la instrucción la dirección de SMRDATA – 0x0C000000 ya que el LinkerScript empieza en esta dirección y es el encargado de enlazar las etiquetas con sus valores.

Tras configurar el controlador de memoria, escribimos a cero el segmento ZI, limitado por las etiquetas ZI_Base y ZI_Limit (en memoria RAM).

Además, copiamos el contenido de la Flash-ROM (dirección 0x00000000) a la memoria RAM (dirección 0x0C000000), se copiarán el número de bytes equivalente a la resta de las etiquetas ZI_Base – RO_Base.

Con el código del ResetHandler escrito, ya podemos generar nuestro binario y ejecutarlo en la placa sin necesidad de estar conectado a un ordenador. La placa ejecutará sobre la Flash-ROM el código de ResetHandler que copiará el programa a la RAM y, cuando ejecute el salto a la etiqueta Main, pasará a ejecutar sobre la memoria RAM nuestro juego Reversi.

Conclusión

En este proyecto empezamos estudiando a fondo la arquitectura ARM. Hemos aprendido a programar en bajo nivel (C/ASM) y hemos estudiado técnicas de programación eficiente comparando nuestros resultados con los de un compilador con opciones de optimización (práctica 1). Después estudiamos los modos de procesador y las excepciones, lo que nos introdujo a la gestión de Entrada/Salida de la arquitectura, estudiamos las interrupciones, sus tipos y las posibilidades de capturarlas e incluso anidarlas.

La placa utilizada nos proporciona distintos dispositivos internos y externos, hemos estudiado a fondo el modo en que son accesibles (puertos, mapeados…), para configurarlos y controlarlos directamente. Los dispositivos de entrada como los botones y el touchpad generan oscilaciones al ser pulsados, hemos desarrollado el software para gestionarlos y eliminar dichos rebotes.

El producto final del proyecto es nuestro juego, hemos diseñado la lógica de nuestro programa de tal forma que reaccione a la interacción del usuario, desarrollando una cola de depuración de eventos concurrentes asíncronos. Generamos la salida correcta y la mostramos por la pantalla LCD, para ello hemos diseñado una interfaz gráfica interactiva de múltiples pantallas, buscando la mejor experiencia de usuario.

Hemos implementado, de forma adicional, la entrada a un modo de ahorro de energía del procesador (IDLE), obteniendo un menor consumo mientras no haya carga de trabajo.

Por último, hemos estudiado la memoria de la placa junto con la inicialización del sistema y el LinkerScript, cargando el software final en la memoria ROM-Flash y consiguiendo una plataforma autónoma para jugar a nuestra versión de Reversi 8.