Memoria Técnica

Proyecto Hardware

Jaime Yoldi Vigueras 779057 José Marín Díez 778148

Enero 2021

Índice

1 Introducción

En este documento se presenta la memoria técnica correspondiente a las prácticas 2 y 3 de la asignatura Proyecto Hardware. A lo largo del documento se detalla el proceso de implementación del conocido juego Reversi. Partiendo del proyecto en lenguaje c previamente optimizado en la práctica 1 se realizan una serie de modificaciones y adaptaciones, añadiendo nuevas funcionalidades y permitiendo que este pueda ser ejecutado en un emulador de máquina real con sus periféricos, como pueden ser botones, leds, teclado, pantalla, etc.

Algunas partes del código serán desarrolladas en lenguaje `ARM` y otras en `c` para finalmente ser todo ello ejecutado con la ayuda de un emulador del procesador *ARM LPC2105*. El entorno de desarrollo empleado es *uVision IDE*.

2 Librerias

2.1 Botones (EINT0, EINT1)

Para simular los botones del juego se emplean dos de las líneas de interrupción externas con las que cuenta el chip: `EINT0` y `EINT1`. Estas permiten interaccionar al procesador con dispositivos de E/S. Para usarlas hay que conectarlas a los pines del sistema (GPIO).

Para establecer la función de los pines, el procesador incluye los registros `PINSEL0` y `PINSEL1`, los cuales son configurados al comienzo de la partida.

Al botón `EINT0` le corresponde el pin 16 (P0.16), para conectarlo se le da un valor de 1 a los bits 1:0 de `PINSEL1`.

```
PINSEL1 = PINSEL1 & 0xfffffffC; // Se limpian los bits 1:0
PINSEL1 = PINSEL1 | 1;          // 01 en los bits 1:0 para EINT0
```

Por otro lado, para conectar el botón `EINT1` con el P0.14 se da un valor de 2 a los bits 29:28 de `PINSEL0`.

```
PINSEL0 = PINSEL0 & 0xcfffffff; // Se Limpian los bits 29:28
PINSEL0 = PINSEL0 | 0x20000000; // 10 en los bits 29:28 para EINT1
```

Además, para que estas interrupciones puedan llegar es necesario habilitarlas en el vector de IRQs, para ello se cuenta con `VicVectEnable`.

La `EINT0` corresponde con la interrupción número 14

```
VICVectCntl2 = 0x20 | 14;
VICIntEnable = VICIntEnable | 0x00004000; // Enable EXTINT0 Interrupt
```

y la `EINT1` con la interrupción número 15

```
VICVectCntl3 = 0x20 | 15;
VICIntEnable = VICIntEnable | 0x00008000; // Enable EXTINT1 Interrupt
```

Las interrupciones se activan por nivel y son activas a baja: es decir que si hay un cero se activa la solicitud de interrupción a la que se ha conectada ese pin.

Cuando una de las dos interrupciones llega –un botón se ha pulsado–, el programa entra en la rutina de servicio que se le ha especificado en el `VicVectAddr`. En ese momento, indica al gestor el evento que ha sucedido mediante el envío de un `EV_BOTON`, no sin antes deshabilitar la interrupción externa correspondiente en el VIC, para evitar que interrumpa de nuevo mientras no haya terminado la gestión de la pulsación.

Como las interrupciones son muy rápidas y un botón puede estar bastante tiempo presionado, si no se hace algo una misma pulsación puede generar multitud de interrupciones. Por evitarlo la cola de eventos deberá encargarse de comprobar si una interrupción se trata de una nueva pulsación o de un botón que se mantiene pulsado. Para ello se facilitan las funciones `eint_esta_pulsado`, `eint1_read_nueva_pulsacion` y `eint0_clear_nueva_pulsacion`.

El botón EINT0 (conectado al pin 16) indicará que el usuario ha introducido un nuevo movimiento.

El botón EINT1 (conectado al pin 14) indicará que el usuario pasa o cancela el movimiento realizado, si se pulsa antes de que pasen 3 segundos.

2.2 Temporizadores

2.2.1 TIMER0

Tratado como fast interrupt, explicar FIQ_handler del startup ### TIMER1 ## GPIO

El *General Purpose Input/Output* (GPIO) es un puerto que permite conectar al chip elementos de E/S. Esta formado por un puerto de 32 bits donde cada uno de ellos representa un pin totalmente independiente, que con la ayuda de una mascara puede ser utilizado para emular diferentes periféricos como botones y leds.

Para su manipulación el GPIO cuenta con 4 registros: `IOPIN`, que muestra el estado de los pines en cada momento, si están activados o no. `IOSET` e `IOCLR` para controlar el estado de los pines y modificar el valor de estos. En el primero, escribiendo un uno en cualquiera de los 32 bits se puede activar el pin correspondiente. Por otro lado, `IOCLR`, tiene el efecto contrario, mediante la escritura en él se pueden desactivar cualquiera de los pines. Finalmente, `IODIR` permite indicar el modo de funcionamiento de ellos, entrada o salida.

Para trabajar con él se implementan una serie de funciones que permiten realizar las operaciones necesarias de manera mas cómoda. Contará con una función `GPIO_leer`

para consultar el valor de unos pines concretos y `GPIO_escribir` para modificar el valor de estos, ambas permitiran su uso en formato decimal.

Además se crearán otras dos funciones `GPIO_marcar_entrada` y `GPIO_marcar_salida` para establecer el modo de funcionamiento de cada pin. En este caso entrada para los botones y salida para los leds.

Finalmente y como se explicará mas adelante, la función que tiene cada pin y el periférico al que corresponde puede ser especificado con ayuda del registro de control `PINSEL0`.

2.3 Cola de eventos

2.4 Gestión de los eventos

2.4.1 Máquina de estados

2.5 Comandos

La librería `comandos` consta de una única función `buscar_comando` la cuál es llamada por el gestor de eventos cuando recibe un evento de tipo `EV_UART0`. Este es generado por la linea de serie (UART0) cuando recibe datos.

Como su nombre indica, su tarea es la de comprobar si se ha introducido algún comando, además de validarlo y procesarlo en caso de que sea correcto. Un comando consta de un delimitador de inicio (#) y un delimitador de fin (!).

Los comando válidos son los siguientes:

- Pasar: #PAS!
- Acabar la partida: #RST!
- Nueva partida:#NEW!
- Jugada: #FCS!

La función va recibiendo los caracteres uno a uno, y cuando uno de ellos se trata del delimitador de inicio la función guardara los siguientes caracteres en un pequeño *buffer* hasta que se encuentre el delimitador de fin. Esto significará que un comando ha llegado, pero es necesario validarlo antes de nada.

Para que un comando sea válido tiene que tener una longitud de tres. Además debe coincidir con alguno de los nombrados previamente: `PAS`, `RST`, `NEW` o `FCS`. En caso de tratarse de alguno de los tres primeros ya se puede considerar válido, por lo que se puede enviar un evento `EV_COMANDO` con la información necesaria.

Sino, si se trata de una jugada, habrá que comprobar también que la fila y la columna respetan las dimensiones del tablero además del *checksum*:

$$(F + V) \mod 8 = S$$

En ese caso se generara de nuevo un `EV_COMANDO` donde se indique la posición de la ficha a colocar.

Si un comando no es válido, se vacía el *buffer* y se manda otro evento con la información necesaria para que la linea de serie muestre un mensaje de aviso en pantalla.

2.6 Power Management

2.7 Modificaciones Reversi8

`* Utilizacion del ARM_ARM`

2.8 Modificaciones Startup

2.9 RTC

2.10 SWI

Durante la ejecución del programa, el procesador –de arquitectura ARM– se encuentra en modo usuario, esto provoca que en momentos concretos surjan algunas limitaciones y sea necesario cambiar a un modo con mas permisos, como el supervisor. Un ejemplo es el de las llamadas al sistema operativo, las cuales solo pueden ser invocadas desde este modo ya que están restringidas al resto.

Para el diseño del juego se han implementado 5 llamadas al sistema. Una para leer marcas temporales y cuatro para gestionar las interrupciones. Estas últimas permiten habilitar y deshabilitar las IRQ y FIQ, pudiendo así realizar operaciones en exclusión, como "alimentar" al Watchdog.

Para cambiar de modo y poder emplear los servicios del sistema operativo se hace uso de la instrucción SWI seguida del número de servicio al que va a corresponder esa llamada.

El acceso al tiempo del `TIMER1` se realiza vía llamada al sistema con número de servicio 0. Para ello se implementa la función `clock_gettime` en lenguaje c declarada en `timer1.h` e incorporada al vector de interrupciones.

```
unsigned long __swi(0) clock_gettime(void);
```

Para la implementación de las otras llamadas, se asume que no hay mas espacio en el vector de interrupciones, por lo tanto son implementadas en la propia rutina de servicio de SWI.

Cuando se invoca una de ellas, SWI causa una interrupción provocando la entrada a la rutina de servicio y el cambio a modo supervisor. Después se guarda la información de estado del usuario en el SPSR y se extrae el número de servicio de la llamada al sistema que ha provocado la interrupción, pudiendo así identificarla y tratar a cada una de manera individual dentro de la ISR.

```
CMP     R12,#0xFF
BEQ     __enable_isr
CMP     R12,#0xFE
BEQ     __disable_isr
CMP     R12,#0xFD
```

Por ejemplo la llamada `__disable_isr` se encarga de deshabilitar las IRQ, para ello modifica el SPSR guardado anteriormente escribiendo en el bit de las IRQ para luego restaurar el modo usuario con el nuevo estado. De igual manera funcionan las FIQ.

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

Para trabajar con interrupciones FIQ que hacen llamadas a funciones, es muy importante tener espacio suficiente reservado a la pila de *Fast interrupt request* ya que si no es así puede surgir un desbordamiento de memoria. Para esto basta con asignar valor a la siguiente posición en el `Startup.s`.

```
FIQ_Stack_Size  EQU     0x00000080
```

2.11 UART0

2.12 RTC

Para medir el tiempo transcurrido durante la partida se hace uso de uno de los dos contadores con funcionalidades especificas del procesador *ARM LPC2105*, el *Real Time Clock* (RTC). Es necesario destacar que este contador, a diferencia de los otros no genera ninguna interrupción, ya que como se ha dicho su único propósito es proporcionar información sobre el tiempo transcurrido y consumiendo poca energía.

Este es iniciado al comienzo del juego y está en funcionamiento en todo momento incluso cuando el procesador está suspendido, en powerdown o idle. Antes de esto es necesario configurarlo para adaptarlo a la frecuencia a la que trabaja el procesador (60MHz), para ello se modifica el *Prescaler Integer register* y el *Prescaler Fraction register* de la siguiente manera.

```
PREINT = 0x726;
PREFRAC = 0x700;
```

Para que el RTC comience a contar es necesario habilitarlo, para ello se activa el bit 0 del *Clock Control Register*, además de poner a 0 la cuenta de minutos y segundos.

```
CCR = 0x01;
```

Para obtener el tiempo transcurrido se hacen funciones `RTC_leer_segundos` y `RTC_leer_minutos` que se encarga de leer del registro `CTIME0` los bits correspondientes y devolver el tiempo para cada caso.

2.13 Watchdog

Otra de las funcionalidades que tiene el juego es que se reinicia tras cierto tiempo de inactividad. Para ello, se utiliza el otro contador específico del procesador, el watchdog (WD).

Cuando el juego comienza, el Watchdog es iniciado especificándole el número de segundos a los que se quiere su reinicio. Para ello, se escribe en el registro *Watchdog Timer Constant* el numero de tics, en función de la frecuencia del procesador. Una vez realizado esto se habilita, se resetea su valor y se alimenta por primera vez para que comience a contar.

Cuando el WD se dispara se activa el segundo bit del *Watchdog Mode register* por lo que previamente se comprueba que no este ya disparado, limpiando el bit en caso afirmativo.

```
if( WDMOD & 0x04 )
    WDMOD &= ~0x04;

// Time out: Pclk*WDTC*4
WDTC  = (60000000 * sec) / 4;
WDMOD = 0x03;
feed_WT();
```

A partir de ese momento el temporizador se decrementa en cada pulsación de reloj, disparándose cuando su valor llegue a 0. La manera de evitar que se dispare es incrementar su tiempo de cuenta (alimentarlo) haciendo que comience de nuevo.

La manera de alimentar al Watchdog es mediante dos escrituras en el registro WDFEED. Para ello se ha creado la función `feed_WT`.

Es de gran importancia destacar que estas escrituras deben ser consecutivas, si no es así el correcto funcionamiento del programa se vera alterado. Por tanto es necesario asegurarse de que no va a llegar ninguna interrupción entre medio. Esto se consigue desactivando todo tipo de interrupción antes de las escrituras y activandolas de nuevo después. Para hacerlo se hará uso de las funciones `disable_isr_fiq` y `enable_isr_fiq` mencionadas anteriormente.

```
disable_isr_fiq();
WDFEED = 0xAA;
WDFEED = 0x55;
enable_isr_fiq();
```

Durante el transcurso del juego el encargado de alimentar el WD es el gestor de eventos,

que lo hará cuando el jugador muestra algún tipo de actividad, ya sea la escritura de un nuevo comando o la pulsación de cualquiera de los dos botones. Si no se realiza ningún movimiento el contador continua decrementandose hasta llegar al final provocando que el procesador se resetee.

También es una manera de evitar que en caso de fallo el procesador se quede bloqueado, ya que si esta colgado el jugador no puede hacer nada, por lo que no será alimentado y también se reseteará.