Práctica 5

Gestión de excepciones e interrupciones

Introducción

Esta práctica añade el soporte de excepciones a nuestro BSP. Para ello se completará un API de gestión de excepciones que permitirá fijar manejadores para cada tipo de excepción definido en la arquitectura *ARM*. Por otro lado, y dado que las interrupciones son un tipo particular de excepción, también se añadirá al BSP el soporte para habilitar y deshabilitar interrupciones en la CPU, de forma que se puedan implementar secciones críticas en las aplicaciones para garantizar el acceso secuencial a los recursos compartidos.

Objetivos

- Saber gestionar la tabla de manejadores de excepción del BSP para poder definir manejadores para cualquier excepción que pueda ocurrir en tiempo de ejecución.
- Saber cómo escribir un manejador de excepciones en *C*.
- Implementar funciones para habilitar/deshabilitar interrupciones, de forma que se puedan usar para crear secciones críticas en nuestra aplicación o BSP cada vez que sea necesario acceder a un recurso compartido.

Gestión de los manejadores de excepción

Cualquier BSP debe permitir a la aplicación la posibilidad de instalar manejadores de excepción a la medida de sus necesidades. Para ello, la tabla de vectores del BSP se debe haber diseñado de forma que cada vez que ocurra una excepción se salte a la dirección indicada en la entrada correspondiente de una tabla de punteros a manejadores de excepción. Por otro lado, y para permitir la gestión de dichos manejadores, el BSP debe proporcionar funciones para fijar y consultar los manejadores dentro de esta tabla.

Diseño de manejadores de excepción

Un manejador de una excepción no deja de ser una función más de nuestra aplicación, con la salvedad de que no recibe argumentos de entrada, no retorna nada, y se llama automáticamente cuando ocurre una excepción. Sin embargo, como cuando ocurre una excepción el procesador cambia de modo antes de llamar al manejador, el código del manejador debe salvar todos los registros que use (y que no estén replicados en el modo de ejecución privilegiado), ejecutar el código, y para finalizar, restaurar los registros y retornar cambiando al modo de ejecución original. Además, en algunos casos también deberá ajustar el valor del registro de enlace antes de retornar.

Este comportamiento se puede conseguir escribiendo el código del manejador en ensamblador o bien mediante el uso del atributo `interrupt` al declarar la función que se usará como manejador. Concretamente, si miramos el manual de GCC[1], podemos ver que para la arquitectura *ARM* se

1 GCC online documentation: http://gcc.gnu.org/onlinedocs/gcc-4.4.7/gcc/Function-Attributes.html

puede indicar para qué tipo de excepción se está generando el manejador, ya que el ajuste de la dirección de retorno del manejador va a depender del tipo de excepción que se produzca, tal y como se ha explicado en las clases de teoría. Por tanto, a la hora de declarar un manejador de interrupciones debe hacerse de la siguiente forma:

```
__attribute__ ((interrupt ("<Tipo de excepción>")))
void my_handler (void)
{
    /* Código del manejador */
}
```

En el compilador de GNU están definidos los siguientes tipos de manejador: `IRQ`, `FIQ`, `SWI`, `ABORT` y `UNDEF`.

Habilitación y deshabilitación de interrupciones en el procesador

Una vez habilitadas las interrupciones en nuestro entorno de ejecución (fichero `crt0.s` del BSP de la práctica anterior) hay que prestar especial cuidado en el diseño del BSP y de nuestra aplicación, ya que en cualquier momento se puede interrumpir la normal ejecución de nuestra aplicación (o de alguna función del BSP) para dar servicio a una interrupción. Si esta interrupción ocurriera cuando se está accediendo a un recurso compartido (un dispositivo, un búfer, una variable compartida, etc.), podría ocurrir que el estado del recurso se modifique sin que la aplicación tenga constancia del cambio, lo que daría paso a un funcionamiento incorrecto. Para evitar esta fuente de problemas es necesario hacer uso de regiones críticas en nuestro código cada vez que se vaya a acceder a un recurso compartido.

Dado que la fuente del problema es la posible aparición de una interrupción, las regiones críticas se implementan deshabilitando las fuentes de interrupción de aquellos dispositivos que puedan hacer uso del recurso compartido que se está usando. De esta forma se impedirá que las ISR de dichos dispositivos puedan interrumpir el programa hasta que se haya terminado de usar el recurso compartido. Llegados a este punto, se puede restaurar el estado de las fuentes de interrupción para volver a un comportamiento normal del sistema. Como se ha visto en la lección 11, la deshabilitación de interrupciones se puede realizar en tres niveles, el dispositivo, el controlador de interrupciones y el procesador. Dado que esta práctica trata sobre la gestión de excepciones en el BSP, nos ocuparemos de este último caso, dejando los otros para más adelante.

Por tanto, una región crítica a nivel de procesador se implementará deshabilitando las interrupciones en el registro de control y estado antes de acceder al recurso compartido y volviendo a dejar las interrupciones como estaban después del acceso al recurso. Para ello, y dado que en el registro de control y estado hay dos bits para habilitar/deshabilitar interrupciones (*I* y *F*), la función que deshabilita interrupciones debe devolver los valores de los bits *I* y *F* para que la función que rehabilita las interrupciones los pueda dejar como estaban antes de la sección crítica.

En cuanto a la implementación de estas funciones, como nos interesa que el API de estas funciones esté en *C*, pero la única forma de acceder al registro de control y estado es mediante las instrucciones máquina `msr` y `mrs`, será necesario hacer uso de ensamblador en línea.

Ejercicios

El objetivo final de esta serie de ejercicios es añadir al BSP que empezamos a diseñar en la práctica anterior el soporte para la gestión de excepciones e interrupciones.

Gestión de manejadores de excepción

En nuestro caso, la tabla de punteros a manejadores de excepción se definió en el fichero `hal/crt0.s` en la práctica anterior. Si el proceso se hizo bien, debería existir un símbolo llamado `_excep_handlers` definido en la dirección `0x00400020` (consultarlo mirando la tabla de símbolos de la imagen de la aplicación).

Para poder acceder a este símbolo desde lenguaje *C* es necesario declararlo como `extern` (ya que se definió en el fichero `hal/crt0.s`), y asignarle un tipo, que en nuestro caso nos interesa que sea un *array* de 8 punteros a manejadores de excepción. Una vez declarado el *array* `_excep_handlers` en el fichero `hal/excep.c`, simplemente hay que rellenar el código de las funciones `excep_set_handler` y `excep_get_handler` y ya podremos definir manejadores de excepción en nuestras aplicaciones.

Forzar una instrucción no definida

Para comprobar que hemos realizado correctamente la implementación del soporte de excepciones en nuestro BSP, realizaremos un sencillo experimento, que consistirá en forzar una instrucción no definida en nuestro código para comprobar que se llame al manejador de dicho tipo de excepción. Pero antes, es necesario que comprendamos bien qué es lo que el procesador entiende por una instrucción.

Para ello, partiremos del código del programa *Hello world* en *C* de la práctica anterior[2], que hacía parpadear el led rojo de la *Econotag*. Una vez construida la aplicación, desensamblaremos el fichero *Elf* resultante mediante la utilidad *objdump* para ver el código ejecutable tal y como lo ve el procesador. Una vez tenemos el código desensamblado, podemos observar como, para cada sentencia de nuestro programa en *C*, aparecen debajo tres columnas con la siguiente información:

- las direcciones de las instrucciones máquina que implementan dicha sentencia,
- el código máquina de dichas instrucciones tal y como lo recibe el procesador (enteros de 32 bits), y
- la traducción de dichas instrucciones máquina a ensamblador para facilitar el entendimiento de dichas instrucciones por un humano.

Una vez visto el código desensamblado, es fácil entender que, desde el punto de vista del procesador, un programa es simplemente una secuencia de enteros de 32 bits que se irán captando, decodificando como instrucciones y ejecutando de forma secuencial. Por tanto, para forzar una instrucción no definida simplemente tenemos que insertar un entero de 32 bits en la secuencia de instrucciones que no se pueda decodificar por ninguna instrucción. Lamentablemente, esto no se puede hacer en *C*, ya que *C* sólo genera instrucciones válidas, por lo que tendremos que hacerlo mediante ensamblador en línea.

Como hemos visto en prácticas anteriores, la directiva de ensamblador `.word` permite escribir un entero de 32 bits en la imagen ejecutable. Hasta ahora habíamos usado esta directiva para inicializar

2 El código se encuentra en `/fenix/depar/atc/se/pracs/app`.

las variables globales en la sección de datos, pero si la usamos en la sección de código, lo que haremos será insertar un entero en mitad del programa. Si el entero se puede decodificar como una instrucción válida, el procesador la ejecutará, y si no, generará una excepción del tipo `UNDEF`.

La mayoría de enteros de 32 bits se van a poder decodificar por alguna instrucción válida. De hecho, es complicado encontrar un entero que no sea decodificable, aunque existen algunos, como por ejemplo `0x26889912`. Podemos probar a insertar dicho entero justo antes del bucle `while` que hace parpadear el led rojo. Para ello escribiremos la siguiente sentencia en nuestro programa *C*:

```
asm(".word 0x26889912\n");
```

Una vez modificado el código, podemos construir de nuevo la aplicación. Al desensamblar el ejecutable, si nos fijamos en la secuencia de instrucciones máquina que implementan la función `main` podremos observar cómo la instrucción que hemos insertado no se ha podido traducir a ninguna instrucción ensamblador.

Si ejecutamos el programa modificado, al llegar a la sentencia que acabamos de insertar se provocará una excepción, se cambiará al modo *undefined,* y se saltará al manejador por defecto de nuestro BSP para las instrucciones no válidas, `_undef_handler`, definido en `crt0.s`, que entra en un bucle infinito (comprobarlo con el depurador).

Uso de manejadores

El siguiente paso es definir dentro de nuestra aplicación un manejador para las excepciones causadas por la aparición de una instrucción no válida en nuestro código. Como el objetivo de este ejercicio es comprobar que el código de gestión de las excepciones de nuestro BSP es correcto, nuestro manejador simplemente indicará que se ha producido una excepción encendiendo el led verde, ignorará el problema y retornará. Una vez escrito el manejador, sólo hay que indicarle al BSP que lo instale en la tabla de manejadores mediante una llamada a la función `excep_set_handler`. Esta llamada debe hacerse justo al principio del código de nuestra aplicación, en su inicialización.

Una vez que tengamos el ejercicio resuelto, el programa debería ejecutarse normalmente, haciendo parpadear el led rojo como siempre, con la salvedad de que el led verde estará encendido, indicando que se ha ejecutado el manejador de la excepción *undefined* cuando se intentó ejecutar la instrucción no válida.

Un aspecto muy importante a tener en cuenta es que para que nuestra aplicación pueda usar los símbolos, constantes y funciones que hay definidas en el BSP será necesario insertar al principio del fichero la siguiente línea de código:

```
#include "system.h"
```

que incluye todo el API de nuestro BSP en el fichero de la aplicación.

Soporte para regiones críticas

Los ficheros `hal/excep.c` y `include/excep.h` se encargan de cubrir el soporte de excepciones de nuestro BSP. Este ejercicio se centrará en la implementación de las funciones

`excep_disable_ints`, `excep_disable_irq`, `excep_disable_fiq`, `excep_restore_ints`, `excep_restore_irq` y `excep_restore_fiq`, que permitirán habilitar y deshabilitar las interrupciones de forma global o de forma selectiva.

Uso de regiones críticas

Modificar la práctica anterior para que la inicialización de los pines del GPIO se haga dentro de una región crítica.

Una vez modificada, si la ejecutamos podremos comprobar con el depurador que el hecho de definir una región crítica mediante las funciones anteriores en la aplicación no sirve para nada, ya que desde el modo USER no se permite que la instrucción máquina `msr` altere los bits 0:23 de los registros de estado de la CPU, y dado que los bits *I* y *F* están dentro de los bits protegidos (son los bits 6 y 7), y que la aplicación se ejecuta en modo USER, tendremos que buscar otro mecanismo para poder definir regiones críticas en modo USER. En la práctica siguiente se indicará como hacerlo mediante el controlador de interrupciones.

Sin embargo, estas funciones son de utilidad en la implementación de las funciones de inicialización del BSP, que se ejecutan en modo SVC, cuando es necesario acceder de forma atómica a un recurso.