Práctica 6

El controlador de interrupciones

Introducción

Las interrupciones son un tipo especial de excepción que se diferencia del resto en que su origen está en un dispositivo, externo a la CPU, que demanda su atención para requerir algún servicio relacionado en la gestión del dispositivo. Otra diferencia importante es que normalmente existen bastantes dispositivos en un sistema, y dado que todos requieren en algún momento la atención de la CPU, y que normalmente suele haber sólo un pin para solicitar interrupciones, todos los dispositivos tienen que realizar la petición a través de dicho pin, lo que exige que la CPU implemente un mecanismo de gestión y priorización de interrupciones para minimizar la latencia de las interrupciones prioritarias. Este caso se ve favorecido en la arquitectura *ARM,* al incorporar dos tipos de interrupciones, IRQ y FIQ, de forma que se puede asociar la fuente de interrupción más prioritaria del sistema a la interrupción FIQ y el resto a la interrupción IRQ. En cualquier caso, seguirá existiendo la necesidad de priorizar la gestión de las interrupciones IRQ.

La priorización de interrupciones se puede realizar mediante software, con ayuda de una tabla de prioridades gestionada por el manejador de la excepción IRQ, o bien con ayuda de un controlador de interrupciones, que descargará de estas tareas al manejador IRQ, lo que minimizará la latencia de las interrupciones. Dado que hoy en día es muy habitual que la mayoría de sistemas incorporen un controlador de interrupciones, en esta práctica se mostrará cómo se puede sacar partido de este circuito para gestionar la atención de interrupciones eficazmente.

Por otro lado, la atención de interrupciones involucra también tanto la gestión de excepciones, dado que son tipos especiales de excepción, como el diseño de Rutinas de Servicio de Interrupción (ISR) adecuadas para cada dispositivo.

Objetivos

- Entender el funcionamiento de un controlador de interrupciones.
- Saber diseñar un *driver* para el controlador de interrupciones de la *Econotag*.
- Saber diseñar un manejador de interrupciones que haga uso de dicho *driver* para priorizar la gestión de interrupciones en nuestro BSP.
- Saber diseñar una ISR.
- Saber usar el controlador de interrupciones para definir regiones críticas en modo USER.

Niveles de gestión de las interrupciones

Las interrupciones se pueden gestionar en diferentes niveles dentro de un sistema. El más alto de ellos es mediante la habilitación/inhibición general de interrupciones mediante los bits *I* y *F* de la CPU, como vimos en la práctica anterior. En este nivel se habilitan o deshabilitan de una sola vez todas las interrupciones asignadas a una patilla de petición de interrupción de la CPU (o a las dos), permitiéndonos definir regiones críticas dentro de nuestro código.

En el caso de que nuestro sistema cuente con un controlador de interrupciones, el siguiente nivel de gestión de interrupciones se encuentra en dicho controlador, en el que se podrán habilitar/inhibir las fuentes de interrupción selectivamente. Para ello, y dado que cada fuente de interrupción está conectada a una entrada diferente del controlador, se usará el número de entrada al que esté conectado físicamente el dispositivo para las diferentes acciones dentro del controlador (inhibir sus interrupciones, detectar las diferentes fuentes que tienen pendiente una petición, etc.).

Por último, el nivel más bajo de gestión de interrupciones se encuentra en el propio dispositivo. Todos los dispositivos cuentan con registros de control en los que se puede habilitar o deshabilitar la generación de interrupciones y configurar aquellos casos en los que el dispositivo podrá interrumpir a la CPU (falta de datos para transmitir, demasiados datos recibidos, diferentes condiciones de error, etc.).

Por tanto, para poder gestionar un dispositivo mediante interrupciones será necesario actuar en estos tres niveles:

- Primero, habrá que habilitar la recepción de interrupciones (bits *I* y *F*) del registro CPRS de la CPU.
- Segundo, habrá que configurar el controlador de interrupciones para habilitar las peticiones de la fuente a la que esté conectado el dispositivo, además de configurar si dichas peticiones se enviarán a la CPU como peticiones IRQ o FIQ,
- Por último, habrá que configurar el dispositivo, habilitándole la generación de interrupciones e indicándole aquellos casos en los que debe generar peticiones de interrupción,

El primer caso ya lo vimos en la práctica 4, y el último lo cubriremos más adelante, cuando diseñemos *drivers* de nivel 1. Esta práctica se centrará en la gestión del controlador de interrupciones.

Prioridades de las diferentes fuentes de interrupción

La prioridad de cada dispositivo a la hora de pedir una interrupción dependerá de la entrada del controlador de interrupciones a la que se haya conectado su salida de petición de interrupción. En la Tabla 8.1 del capítulo 8 del manual de referencia del MC1322X de *Freescale* se detalla esta información. Se puede comprobar que se ha asignado un valor 0 (mínima prioridad) a la fuente ASM (*Advanced Security Module*) y un valor 10 (máxima prioridad) al controlador SPI, estando las entradas 11-15 del controlador de interrupciones sin usar.

Por otro lado, además de asignar prioridades a cada fuente, el número asociado a cada fuente de interrupción se usará en los registros de control/estado del controlador de interrupciones para gestionar las diferentes fuentes de interrupción (identificación de la fuente de la interrupción, inhibir interrupciones, etc.). Por tanto, para que nuestro *driver* se adapte a esta restricción, se ha definido un tipo de datos en `include/itc.h` llamado `itc_src_t`, en el que se definen todas las posibles fuentes de interrupción y se les asigna el número adecuado, de forma que se puedan usar en el resto de funciones de nuestro *driver.*

El mapa de memoria del controlador de interrupciones

Como cualquier otro dispositivo, el controlador de interrupciones se gestiona mediante registros mapeados a memoria, que en este caso se utilizarán para habilitar/inhibir fuentes de interrupción,

asignar fuentes de interrupción a la entrada IRQ o FIQ de la CPU, o comprobar qué fuentes de interrupción tienen pendiente el servicio. En la sección 8.5 del manual de referencia de MV1322X podemos obtener la información de qué registros se emplean para estas acciones y de cómo usarlos.

Habilitar/deshabilitar fuentes de interrupción

Suponiendo que se han configurado los dispositivos adecuadamente para que generen interrupciones y que los bits *I* y *F* de la CPU están también con el valor adecuado, el controlador de interrupciones nos permitirá habilitar y deshabilitar interrupciones selectivamente. Para ello, se pueden usar el registro `INTENABLE`, que permitirá habilitar/deshabilitar todas las fuentes de interrupción en paralelo mediante una máscara de bits, o bien la pareja de registros `INTENNUM`/`INTDISNUM`, que habilitan/deshabilitan una fuente de interrupción cada vez en función de su número.

Regiones críticas en modo USER

En la práctica anterior descubrimos que los bits *I* y *F* del registro `CPSR` de la CPU no se pueden alterar en modo USER, por lo que en principio, parece que no se pueden definir regiones críticas en modo USER. Sin embargo, y dado que mediante el controlador de interrupciones podemos habilitar y deshabilitar las peticiones de interrupción, se pueden diseñar funciones para poder definir regiones críticas en modo USER. Concretamente, podemos definir una función `itc_disable_ints` que guarde el valor del registro `INTENABLE` en una variable global y que luego fije `INTENABLE` a 0, para deshabilitar todas las fuentes de interrupción, y otra función, `itc_restore_ints`, que vuelva a colocar el valor original de la variable global en el registro `INTENABLE`, para dejar la habilitación de las interrupciones como estaban. Las variable global que guardará el estado de habilitación de las interrupciones debería declararse en el fichero `drivers/itc.c` como `static`, para garantizar que sea privada al driver del controlador de interrupciones.

Mapear las fuentes de interrupción a las entradas I o F de la CPU

El registro `INTTYPE` se usa para indicar qué fuentes de interrupción se asignarán a la entrada *I* o *F* de la CPU. Para ello se emplean sus bits 0:15, uno para cada una de las 16 fuentes de interrupción. Si el bit se fija a cero (valor por defecto), la entrada se habrá asignado a la entrada IRQ. Por el contrario, si uno de estos bits se pone a 1, se asignará a la entrada FIQ.

Gestión de los manejadores de interrupción

Una vez que un dispositivo pide una interrupción, que dicha interrupción pasa por el controlador de interrupciones, y que acaba llegando la CPU, el manejador de excepciones del BSP debe invocar a la ISR correspondiente. Para ello, el *driver* del controlador de interrupciones debe almacenar en algún sitio una tabla de manejadores de interrupción o ISRs.

En los controladores vectorizados esta tabla se almacena en el propio controlador, lo que agiliza el tratamiento de las interrupciones. Sin embargo, en nuestro caso la *Econotag* no dispone de un controlador vectorizado, por lo que la gestión de las ISR se tendrá que hacer por software en nuestro *driver,* mediante una tabla de punteros a función. Concretamente, se trata de la tabla `itc_handlers`, declarada como `static` en `drivers/itc.c`.

Por otro lado, y dado que esta tabla se ha declarado como `static`, y no será visible fuera del fichero `drivers/itc.c`, nuestro *driver* deberá incorporar funciones para asignar y consultar las

ISRs a cada fuente de interrupción.

Servicio de interrupciones

Como se ha explicado en las lecciones de teoría, cuando un dispositivo solicita una interrupción, la petición pasa por el controlador de interrupciones, y si es la más prioritaria y no está enmascarada, acaba llegando a la CPU, que la atenderá si tiene habilitado el servicio de interrupciones (bits I y F del registro de estado). Una vez que la CPU recibe la petición, cambiará de modo (al modo IRQ o FIQ, según el caso), deshabilitará las interrupciones de menor o igual prioridad en el registro `CPRS`, y saltará al vector de excepción correspondiente, que a su vez saltará al manejador de nivel 0 de la interrupción.

Para poder servir la interrupción se debe identificar su fuente, localizar su ISR, llamar a su ISR, y retornar al modo de ejecución anterior. Además, en el caso de las interrupciones IRQ, dado que es habitual que haya más de una fuente de interrupción asociada a la petición IRQ, el manejador puede diseñarse para que rehabilite las interrupciones IRQ cuanto antes, para permitir anidamiento de interrupciones. Sin embargo, en el caso de las interrupciones FIQ, dado que hay solamente una fuente de interrupción asociada a la entrada FIQ, y que son las interrupciones de máxima prioridad, el manejador no tiene por qué rehabilitar las interrupciones anticipadamente ni localizar la ISR de la interrupción, por lo que su implementación será mucho más sencilla.

Teniendo todo esto en cuenta, lo habitual es que el servicio de una interrupción esté estructurado en tres niveles de abstracción. El nivel más alto es el manejador de nivel 0, que está implementado en el mecanismo de gestión de excepciones del BSP, y que depende de la arquitectura de la CPU del sistema. En el caso de los procesadores ARM, como disponen de dos tipos de interrupción, este mecanismo consiste en definir un manejador de nivel 0 para cada tipo de excepción (IRQ y FIQ) que salve en contexto, que ajuste el contenido del registro de enlace y que llame al manejador de nivel 1, encargado de identificar la fuente y dar servicio a la interrupción. Puesto que los manejadores de nivel 0 se llamarán automáticamente tras el cambio a un modo privilegiado (IRQ o FIQ), deben retornar al modo de ejecución anterior a la ejecución (copiando el registro `SPSR` a `CPSR`), por lo que no se pueden implementar como funciones *C* normales. Si queremos gestionar interrupciones anidadas, su implementación deberá ser necesariamente en ensamblador. Sin embargo, si no tenemos necesidad de gestionar interrupciones anidadas, se pueden implementar en *C*, simplemente asignando el atributo `__attribute__ ((interrupt ("IRQ")))` o `__attribute__ ((interrupt ("FIQ")))`, según el caso, a la implementación del manejador.

El siguiente nivel de abstracción es el manejador de nivel 1, que depende del controlador de interrupciones del sistema. En nuestro BSP está implementado dentro del *driver* del controlador de interrupciones, y se encarga de identificar la fuente de interrupción de más prioridad, localizar su ISR y llamarla. En nuestro BSP estas tareas se implementan en las funciones `itc_service_normal_interrupt` e `itc_service_fast_interrupt`, que hacen uso de la tabla de manejadores `itc_handlers` y de los registros `NIVECTOR` y `FIVECTOR`, que indican el número de fuente de interrupción activa más prioritaria en cada momento.

Una vez localizada la fuente de la interrupción, el manejador de nivel 1 llamará a la ISR del dispositivo (con ayuda de la tabla `itc_handlers`, que almacena la lista de ISRs definidas en el sistema) , que es el tercer nivel de abstracción y cuya implementación depende de cada dispositivo.

Simulación de interrupciones

Dado que las interrupciones son eventos provocados por dispositivos externos a la CPU, es difícil

saber con exactitud cuándo se van a provocar, lo que dificulta bastante la depuración de programas basados en interrupciones. Para facilitar el desarrollo de dichos programas, algunos controladores de interrupciones, como el de *Freescale*, incorporan la posibilidad de simular que ha ocurrido una petición de interrupción en cualquiera de sus fuentes. De esta forma, podremos forzar una interrupción en cualquier momento, para comprobar paso a paso con el depurador que el procesador cambia inmediatamente de modo, que copia el registro `CPSR` del modo de ejecución anterior al registro `SPSR` del nuevo modo, que salta automáticamente al vector de excepción correspondiente, que se ejecuta su manejador de nivel 0 (gestión de excepciones del BSP), que el manejador de nivel 0 llama al de nivel 1 (en el *driver* del controlador de interrupciones), y que éste, a su vez, acaba llamando a la ISR de la interrupción forzada, y que por último, una vez que la ISR y el manejador de nivel 1 terminan, el manejador de nivel 0 de la excepción retorna al programa principal cambiando al modo de ejecución original y restaurando el registro `SPSR` en `CPSR`.

Concretamente, el controlador de interrupciones de *Freescale* dispone del registro `INTFRC`, que fuerza la llegada de una petición de interrupción desde las fuentes que se activen en dicho registro en los bits correspondientes. Para dejar de forzar la llegada de interrupciones simplemente hay que volver a poner a cero los bits del registro.

Inicialización del controlador de interrupciones

Una vez que tenemos toda la funcionalidad del controlador de interrupciones implementada, solamente nos queda inicializarlo para que comience a funcionar. Para ello, implementaremos la función `itc_init`, que será invocada en la inicialización del BSP cuando se inicie la ejecución de nuestro *firmware.*

Básicamente, la función `itc_init` debe:

- Inicializar a `NULL` todas las entradas de la tabla `itc_handlers`,
- inicializar el registro `INTFRC` a cero, para que no se provoque ninguna interrupción simulada, al activar el controlador
- Poner a 0 todos los bits del registro `INTENABLE,` ya que inicialmente todas las fuentes de interrupción estarán deshabilitadas
- Poner a 0 los bits 19 y 20 del registro `INTCNTL` para activar el arbitraje de las interrupciones IRQ y FIQ.

Diseño de una ISR

Como ya se ha explicado a lo largo de la práctica, cuando el procesador detecta una petición de interrupción, entre el manejador de excepciones y el *driver* del controlador de interrupciones se encargan de localizar a la ISR de la fuente de la interrupción y llamarla para que atienda la petición del dispositivo que haya realizado la petición.

Si observamos la definición del tipo de datos usado para las ISR en nuestro BSP, podemos comprobar cómo en `include/itc.h` está la definición del tipo `ict_handler_t`, que se define como una función sin argumentos que no retorna nada. Esto es así porque las ISR no reciben argumentos, sino que una vez llamadas inspeccionan los registros de estado del dispositivo para identificar la causa de la interrupción y darle el servicio adecuado. Las ISR tampoco retornan ningún valor. Una vez prestado el servicio, retornan al controlador de interrupciones que a su vez retornará al manejador de excepción, que a su vez cambiará de modo y retornara al programa.

El código de la ISR dependerá de qué tipo de interrupción tenga que atender y del dispositivo concreto, por lo que no tiene mucho sentido hablar de este tema genéricamente. Sin embargo, hay un aspecto muy importante que se debe tener en cuenta a la hora de diseñar una ISR. Como se ha indicado más arriba, las interrupciones se gestionan en tres niveles. El dispositivo genera la petición, el controlador la filtra y prioriza, y la CPU la atiende. Pues bien, en el momento en el que se empieza a ejecutar una ISR, lo primero que tiene que hacer reconocer la petición de interrupción para que el dispositivo deje de seguir solicitando dicha interrupción. Esto se realiza normalmente accediendo a los registros de control/estado del dispositivo, y una vez hecho, el dispositivo deja de activar la línea de petición de interrupción. El proceso depende del dispositivo en concreto, por lo que habrá que mirar en su hoja técnica para ver cómo se hace en cada caso.

Si no se reconociera la petición de interrupción, el dispositivo seguiría pidiendo la interrupción indefinidamente, por lo que en cuanto la ISR retornara, la CPU volvería a ser interrumpida de nuevo para volver a servir la interrupción, volviendo a ejecutar de nuevo la ISR. Este proceso desencadenaría un bucle infinito y nuestro programa dejaría de funcionar.

Ejercicios

El objetivo final de esta serie de ejercicios es añadir al BSP que estamos diseñando el *driver* del controlador de interrupciones, de forma que más adelante podamos implementar *drivers* de nivel 1 en nuestro BSP.

Acceso estructurado a los registros de control/estado del controlador

Implementar en el fichero `drivers/itc.c` la estructura `itc_regs_t`, que permita acceder a los registros del ITC de forma cómoda mediante las funciones del *driver*. Prestar especial cuidado en la generación del espacio reservado desde la dirección `ITC_BASE + 0x18` a `ITC_BASE + 0x24`.

Comprobar que los campos de la estructura están en la dirección correcta. Este paso es fundamental para que el resto de la práctica funcione correctamente. Para ello se puede hacer uso de la función de inspeccionar variables del depurador, indicándole que nos muestre las direcciones en memoria de cada uno de los campos de la estructura.

Habilitación/inhibición de fuentes de interrupción

Implementar en el fichero `drivers/itc.c` las funciones `itc_enable_interrupt` e `itc_disable_interrupt`, que nos permitirán habilitar/inhibir fuentes de interrupción en nuestro sistema.

Regiones críticas en modo USER

Implementar en el fichero `drivers/itc.c` las funciones `itc_disable_ints` e `itc_restore_ints`.

Mapeo las fuentes de interrupción a las entradas *I* o *F* de la CPU

Implementar en el fichero `drivers/itc.c` la función `itc_set_priority`. Para ello será imprescindible el uso de máscaras de bits, ya que la asignación de una prioridad a una fuente no debe modificar el resto de los bits del registro `INTTYPE`. Por otro lado, la función debe asegurar de que sólo hay una fuente de interrupción mapeada a la entrada FIQ en cada momento.

Gestión de los manejadores de interrupción

Implementar en el fichero `drivers/itc.c` la función `itc_set_handler`, que asigna una ISR a cada fuente de interrupción. El contenido de dicha función deberá almacenar el puntero a función indicado en la entrada correspondiente dentro de la tabla `itc_handlers`.

Servicio de interrupciones

Implementar las funciones `itc_service_normal_interrupt` e `itc_service_fast_interrupt` del fichero `drivers/itc.c`. Una vez implementadas, ya tenemos los manejadores de nivel 1 del soporte de interrupciones en nuestro BSP. Para completar el soporte de interrupciones nos faltaría implementar el nivel alto de abstracción, es decir, sería necesario implementar los manejadores de nivel 0 de las excepciones IRQ y FIQ.

Como de momento sólo vamos a usar una fuente de interrupción, implementaremos sólo el manejador de nivel 0 de excepciones IRQ, y lo implementaremos como un manejador de interrupciones no anidadas (función `excep_nonnested_irq_handler`). Dicha implementación es trivial, ya que simplemente debe llamar a la función del *driver* del controlador de interrupciones que sirve la ISR de la IRQ de más prioridad, implementada más arriba. El único detalle a tener en cuenta es que como es un manejador de excepción, no puede ser una función normal, ya que al retornar deberá cambiar de modo de ejecución y copiar el registro `SPSR` al `CPSR` del modo al que se retorne. Por tanto, en la implementación de la función en `hal/excep.c` (y no en su cabecera en `include/excep.h`) hay que indicarle al compilador que es un manejador de excepción mediante el atributo adecuado.

Simulación de interrupciones

Implementar las funciones `itc_force_interrupt` e `itc_unforce_interrupt` para habilitar en nuestro *driver* la posibilidad de simular peticiones de interrupción. Asegurarse de que que las funciones sólo afectan al bit indicado como argumento, dejando el resto del bits del registro inalterados.

Inicialización del controlador de interrupciones

Implementar la función `itc_init`.

Comprobación de la práctica

Dado que todavía no hemos implementado el driver del GPIO, ni ningún otro driver de E/S, lo que haremos para comprobar que la gestión de interrupciones funciona correctamente en nuestro BSP será forzar una interrupción en el controlador de interrupciones. Concretamente, forzaremos una petición de interrupción del módulo ASM del chip, el dispositivo con menor prioridad del sistema.

La ISR para este módulo simplemente encenderá el led verde y retornará, pero si atendemos a los consejos sobre el diseño de ISRs dados más arriba, lo primero que debe hacer dicha ISR, incluso antes de encender el led verde, es reconocer la interrupción para que el dispositivo deje de solicitar la interrupción. Como en este caso la interrupción ha sido forzada con propósitos de depuración, y somos conscientes de que la IRQ ha sido forzada, el proceso de reconocimiento consistirá en indicarle al controlador de interrupciones que deje de forzar la petición de interrupción. Si la ISR retornara sin haber indicado al controlador de interrupciones que deje de forzar la petición de IRQ, la CPU entraría en un bucle infinito en el que una vez ejecutada la ISR se volvería e ejecutar de

nuevo indefinidamente, ya que el controlador de interrupciones seguiría manteniendo la petición activa.

Una vez escrita la ISR, simplemente hay que indicarle al *driver* del controlador de interrupciones que asigne dicha ISR a la fuente ASM, habilitar la interrupción para la fuente ASM en el controlador de interrupciones y forzar la interrupción para dicha fuente. Si hacemos estas tres llamadas al *driver* del controlador de interrupciones justo antes del bucle `while`, 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 atendido la interrupción simulada.

Para que nuestra aplicación pueda usar el API de gestión del controlador de interrupciones del BSP será necesario insertar al principio del fichero la siguiente línea de código:

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