Facultad de Informática
Universidad Complutense de Madrid

Prática 3 EC - Interrupciones del MSP430

15 de febrero de 2024

Cuando esperamos un evento en software, tenemos dos principales formas de hacerlo:

- La primera es la espera activa, o *polling*, donde el programa está continuamente comprobando si la condición de fin de espera se cumple, antes de continuar la ejecución.
- Por otro lado, podemos utilizar *Interrupciones*, que es una forma más eficaz de gestión. En este caso, se delega la comprobación al gestor de interrupciones, quien cuando detecte la condición de fin de espera, interrumpirá al procesador para avisar de que los datos están listos.

Podemos encontrar una analogía con un viaje:

Polling	Interrupciones
Ya hemos llegado? NO	Avísame cuando lleguemos!
Ya hemos llegado? NO	
Ya hemos llegado? NO	
Ya hemos llegado? NO	
Ya hemos llegado? NO	
Ya hemos llegado? NO	
...	
Ya hemos llegado? NO	
Ya hemos llegado? NO	
Ya hemos llegado? SÍ	Llegamos!

A la vista del ejemplo, es claro cuál preferiríamos si lleváramos pasajeros en el coche, por nuestra salud mental.
A nivel de código, la diferencia no es tan evidente, un pseudocódigo sería de este estilo:

```
//Poll GPIO button
while(1) {
    while(GPIO_getInputPinValue() == 1)
        GPIO_toggleOutputOnPin();
}
```

```
//GPIO button interrupt
#pragma vector=PORT1_VECTOR
__interrupt void rx (void) {}
    GPIO_toggleOutputOnPin();
}
```

1. E/S con espera de respuesta (*polling*)

En la entrada/salida con espera de respuesta la CPU comprueba periódicamente el estado del dispositivo, leyendo de una dirección de entrada/salida correspondiente a uno o varios registros de estado del dispositivo. Los bits de los registros de estado indican la situación concreta del dispositivo (por ejemplo, si se trata de un teclado, podemos saber si el usuario ha presionado una tecla).

Esta forma de entrada/salida es sencilla, pero claramente ineficiente. Por ejemplo, si un usuario tarda 10 segundos en mover el ratón, se habrán realizado miles de encuestas al dispositivo sin detectar un nuevo evento, con la consecuente pérdida de tiempo para realizar otras tareas en la CPU. Por otra parte, el ritmo de transferencia de datos está limitado por la velocidad de la CPU, ya que no podremos encuestar al dispositivo con una frecuencia arbitrariamente alta. Por tanto, esta forma de entrada/salida debe evitarse en lo posible. Sin embargo, en algunas ocasiones no quedará otra opción, ya que el dispositivo en cuestión no genera interrupciones.

2. Interrupciones

Son eventos que afectan a la rutina principal al forzar un cambio en el flujo normal del programa. Existen tres tipos de interrupciones:

- Reinicio o RESET. Es un evento especial que lleva asociado una interrupción no enmascarable. Puesto que la interrupción no se puede deshabilitar, se usa para "sacar" a la CPU de cualquier situación en la que se encuentre. Nos permite definir un estado inicial para el sistema. El MSP430presenta dos señales RESET:
  - POR (Power On Reset): la señal se genera al encender el dispositivo o cuando hay problemas de alimentación.
  - PUC (Power Up Clear): la señal se genera cuando se produce un PUR o cuando expira el tiempo de Watchdog.
- Interrupción no enmascarable (NMI). Son interrupciones de alta prioridad, para las que no tiene efecto el bit de habilitación global de interrupciones (GIE). Se pueden enmascarar mediante bits individuales (NMIE, ACCVIE, OFIE) del registro deshabilitación de interrupciones (IE1).
- Interrupción enmascarable. Se controlan con el registro GIE. Se usan principalmente en periféricos con capacidad de Interrupción (habilitables o deshabilitables individualmente por software). Con el bit GIE se deshabilitan globalmente todas las interrupciones enmascarables.

El MSP430 trabaja con un mecanismo de interrupciones con prioridad. El sistema atiende primero aquellas interrupciones que presenten mayor prioridad, quedando pendientes el resto (el flag de interrupción permanece activo). La prioridad de las interrupciones no la puede determinar el usuario, se establece de fábrica.

Podemos ver la documentación relacionada con interrupciones en la guía `slau367p.pdf`, sección 1.3

2.1. Proceso de una interrupción

1. Se produce un evento (cambio en una de las entradas, salidas o en los periféricos internos (watchdog, temporizadores, etc.) del microcontrolador.
2. Se activa el flag de interrupción asociado a ese evento. Una interrupción puede estar asociada a varias fuentes (registro IE1).
3. Se pone en marcha el proceso de tratamiento de interrupción. En función de cómo se encuentre la CPU se tienen dos casos:
   - La CPU se encuentra ejecutando una instrucción del programa. Para poner en marcha el proceso de interrupción se espera a que se termine de ejecutar la instrucción en curso.
   - La CPU se encuentra parada, es decir, no está ejecutando instrucciones. Se finaliza "temporalmente" el estado de bajo consumo hasta que se haya tratado completamente la interrupción.
4. Se almacena en la pila la dirección de memoria de la siguiente instrucción del código del programa que se estaba ejecutando.
5. Se almacena en la pila el valor del registro de estado.
6. Si existen varias interrupciones pendientes se selecciona aquella que presente mayor prioridad (la selecciona el microcontrolador automáticamente).
7. Si el flag de interrupción está asociado a una sola fuente, se borra (se pone a 0) el flag de interrupción asociado al evento que provocó la interrupción. Si la CPU no hiciera esto, entendería que el evento se está produciendo continuamente. Si la interrupción está asociada a múltiples fuentes (varios eventos) el programador es el que debe desactivarlo.
8. El registro de estado toma el valor 0x0000. En consecuencia se deshabilitan todas las interrupciones enmascarables (GIE = 0).
9. A partir del contenido del vector de interrupción se obtiene la dirección de memoria de la RTI asociada a la interrupción, y ésta se carga en el contador de programa.
10. Se ejecuta la secuencia de sentencias de la RTI.
11. La RTI termina con la instrucción RETI (RETurn from Interruption).
12. Se extrae de la pila el valor del registro de estado y contador de programa anteriores a la ejecución de la RTI.

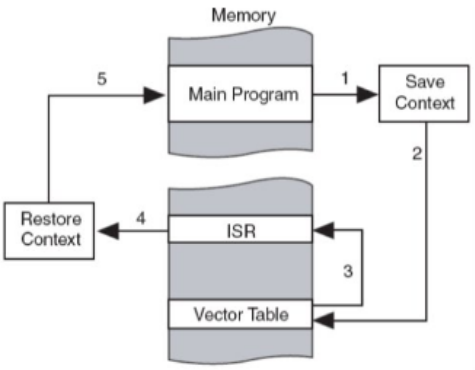

Figura 1: Diagrama del proceso general de gestión de una interrupción

2.2. Vector de interrupción

Se trata de un array ubicado al final de la memoria (a partir de la dirección 0xFF80 en la placa MSP430FR6989) cuya finalidad es ubicar las direcciones de comienzo de las RTI. Cuando ocurre una interrupción, la CPU recurre a este vector para saber dónde se halla la RTI asociada a la interrupción que se produjo. Cada periférico tiene asociado intrínsecamente una posición en el array. En el archivo de cabecera del microcontrolador (msp430fr6989.h) están declaradas las constantes asociadas a cada interrupción, que sumadas a la dirección 0xFF80 permiten obtener la posición de memoria en la que se almacena la primera instrucción de la RTI correspondiente.

2.3. Registros de interrupción

Los pines de los puertos P1 y P2 pueden configurarse para producir interrupciones usando los registros PxIFG, PxIE, y PxIES. Los flags de interrupción de estos puertos tienen asociadas prioridades, de forma que el pin 0 (PxIFG.0) tiene la prioridad más alta, y todos los pines utilizan el mismo vector de interrupción.

Podemos ver la documentación relacionada en la guía `slau367p.pdf`, sección 12.4.9 hasta 12.4.11

1. **Registro de estado (status register)**

   Para activar las interrupciones globalmente hay que poner a 1 el bit GIE (generate interrupt enable) del registro de estado. Puede hacerse de dos formas mediante funciones intrínsecas:

   ```
   __bis_SR_register(GIE);
   __enable_interrupt();
   ```

   El registro de estado *status register*, en caso del MSP430, no está mapeado en memoria, sino que es un registro propio de la arquitectura (dentro del banco de registros). Por tanto, se opera de esta manera especial mediante las funciones intrínsecas, que generan directamente el ensamblador necesario.

   Podemos ver la documentación relacionada en la guía `slau367p.pdf`, sección 4.3.3

2. **Registros de activación de interrupción (PxIE)**

   Cada bit de los registros PxIE habilita las interrupciones asociadas al puerto y pin correspondiente.

PxEN	Significado
0	Interrupción deshabilitada
1	Interrupción habilitada

3. **Interrupt edge select registers (PxIES)**

   Cada bit de los registros PxIES selecciona el flanco en que se dis paran las interrupciones asociadas al puerto y pin correspondiente.

PxIES	Significado
0	Interrupción en flanco de subida
1	Interrupción en flanco de bajada

4. **Port Interrupt Flag Register (PxIFG)**

   Cada bit de los registros PxIFG corresponde al flag de interrupción del puerto y pin asociado. El flag se activa cuando en la señal de entrada se produce el cambio de valor seleccionado. La interrupción debe ser atendida si el bit correspondiente en el registro PxIE y el bit GIE están activos, es decir, cuando la interrupción está habilitada local y globalmente.

PxIFG	Significado
0	No hay interrupción pendiente
1	Sí hay interrupción pendiente

2.4. Rutina de tratamiento de interrupción (RTI)

Las RTI deben indicarse usando las siguientes directivas del preprocesador de C:

```
#pragma vector=[fuente de interrupción]
__interrupt
```

Las fuentes de interrupción están disponibles en el archivo `msp430fr6989.h` (generado automáticamente en los proyectos de code composer studio) a partir de la sección `Interrupt Vectors`.

Si la interrupción está asociada a múltiples fuentes el programador puede resolver mediante encuesta en la RTI cuál de ellas provocó la interrupción. Antes de salir de la RTI es importante borrar el flag de interrupción atendida. Si no se hace, se volvería a disparar la interrupción nada más salir de la RTI y el programa quedaría ejecutando esta rutina indefinidamente.

A continuación se muestra un programa que cambia el estado de P1.0 según lo recibido por P1.4 basado en *polling*.

```
#include <msp430.h>

void main(void) {
    //disable watchdog
    WDTCTL = WDTPW | WDTHOLD;
    //disable high impeadance on I/O
    PM5CTL0 &= ~LOCKLPM5;

    P1DIR = 0x01;   //set P1.0 as output, rest as inputs
    P1REN = 0x10;   //enable pull on P1.4, rest disabled
    P1OUT = 0x10;   //set pull-up on P1.4

    while(1) {
        if (P1IN & 0x10) { //if P1.4 is not pushed (1 is default since we use pullup)
            P1OUT |= 0x01;  //write 1 on P1.0
        } else {
            P1OUT &= ~0x01; //write 0 on P1.0
        }
    }
}
```

A continuación se muestra un programa que cambia el estado de P1.0 con cada pulsación de P1.4 mediante *RTIs*:. En el tiempo en que está esperando, entra en modo de baja energía.

```
#include <msp430.h>

void main(void) {
    //disable watchdog
    WDTCTL = WDTPW | WDTHOLD;
    //disable high impeadance on I/O
    PM5CTL0 &= ~LOCKLPM5;

    P1DIR = 0x01;   //set P1.0 as output, rest as inputs
    P1REN = 0x10;   //enable pull on P1.4, rest disabled
    P1OUT = 0x10;   //set pull-up on P1.4

    P1IE    |= 0x10;    //P1.4 IRQ enabled
```

Figure 25-17. TAxR Register

15	14	13	12	11	10	9	8
TAxR							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
TAxR							
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Table 25-5. TAxR Register Description

Bit	Field	Type	Reset	Description
15-0	TAxR	RW	0h	Timer_A register. The TAxR register is the count of Timer_A.

Figura 2: Registro TAxR

```
    P1IES   |= 0x10;    //P1.4 Falling edge
    P1IFG   &= ~0x10;    //P1.4 clear pending interrupts

    __bis_SR_register(LPM4_bits + GIE);  //Enter LMP4 mode (sleep)
}

//Port 1 ISR
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void) {
    P1OUT ^= 0x01;  //Toggle P1.0
    P1IFG &= ~0x10;  //P1.4 clear pending interrupts
}
```

3. Temporizadores

El MSP430 está provisto de varios temporizadores que permiten medir intervalos de tiempo y generar eventos basados en el tiempo. Hay dos tipos de timers: los `Timers_A` y los `Timers_B`. Ambos son muy similares, aunque los de tipo B poseen alguna característica más que los A. Algunas características de estos temporizadores son:

- Utilizan un contador de 16 bits con cuatro modos de operación.
- Pueden seleccionar y usar diferentes fuentes de reloj.
- Utilizan registros de captura/comparación para su configuración.
- Permiten salidas PWM (Power Width Modulation).
- Pueden usar interrupciones.

3.1. Registros del temporizador A

El temporizador dispone de un registro contador de 16 bits llamado TAxR (donde x será 0 para el timer TA0 y así sucesivamente). Para controlar el temporizador se utilizan los registros TAxCTL y TAxCCTLn. Cada temporizador tiene un registro TAxCTL y hasta siete registros TAxCCTLn que se utilizan para captura/comparación.

Podemos ver la documentación relacionada en la guía `slau367p.pdf`, sección 25

1. **Registro TAxR**

   Este registro es simplemente un contador que se incrementa/decrementa (Según el modo de operación) con cada flanco de subida del reloj. Se puede leer o escribir desde software, y el temporizador puede generar una interrupción al desbordar. Para más información mirar la **??**

2. **Registro TAxCTL**

   Este registro contiene la configuración general del temporizador. Se pueden configurar las fuentes de reloj, si se quiere dividir la frecuencia a la hora de contar, el modo de operación, bits de control de interrupciones, y un bit de clear para resetear el contador. Para más información mirar la **??**.

Figure 25-16. TAxCTL Register

15	14	13	12	11	10	9	8
Reserved						TASSEL	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)
7	6	5	4	3	2	1	0
ID		MC		Reserved	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	w-(0)	rw-(0)	rw-(0)

Table 25-4. TAxCTL Register Description

Bit	Field	Type	Reset	Description
15-10	Reserved	RW	0h	Reserved
9-8	TASSEL	RW	0h	Timer_A clock source select 00b = TAxCLK 01b = ACLK 10b = SMCLK 11b = INCLK
7-6	ID	RW	0h	Input divider. These bits along with the TAIDEX bits select the divider for the input clock. 00b = /1 01b = /2 10b = /4 11b = /8
5-4	MC	RW	0h	Mode control. Setting MC = 00h when Timer_A is not in use conserves power. 00b = Stop mode: Timer is halted 01b = Up mode: Timer counts up to TAxCCR0 10b = Continuous mode: Timer counts up to 0FFFFh 11b = Up/down mode: Timer counts up to TAxCCR0 then down to 0000h
3	Reserved	RW	0h	Reserved
2	TACLR	RW	0h	Timer_A clear. Setting this bit clears TAR, the clock divider logic (the divider setting remains unchanged), and the count direction. The TACLR bit is automatically reset and is always read as zero.
1	TAIE	RW	0h	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0b = Interrupt disabled 1b = Interrupt enabled
0	TAIFG	RW	0h	Timer_A interrupt flag 0b = No interrupt pending 1b = Interrupt pending

Figura 3: Registro TAxCTL

Figure 25-18. TAxCCTLn Register

15	14	13	12	11	10	9	8
CM		CCIS		SCS	SCCI	Reserved	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r-(0)	r-(0)	rw-(0)
7	6	5	4	3	2	1	0
OUTMOD			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

Figura 4: Registro TAxCTL

3. **Registro TAxCCTLn**

   El campo CAP del registro TAxCCTLn nos permite seleccionar el modo captura o el modo comparación. El campo CCIFG se activa cuando el valor de la cuenta del registro TAxR alcanza el valor del registro TAxCCRn.

   El timer puede generar dos tipos de interrupciones. Por un lado, mediante el bit TAIE del registro TAxCTL podemos generar una interrupción cada vez que el contador alcance el valor 0. El segundo tipo de interrupciones se desencadenan cuando el valor de la cuenta del registro TAxR ha alcanzado el valor almacenado en el registro TAxCCRn, se habilitan mediante el bit CCIE del registro TAxCCTLn. En este caso, se activa el campo CCIFG del registro TAxCCTLn. Se pueden ver los campos en la **??**

3.2. Ejemplo de configuración y RTI del TimerA

```
// Configuración TIMER_A:
// TimerA1, ACLK/1, modo up, reinicia TACLR
TA1CTL = TASSEL__ACLK | ID__1 | MC__UP | TACLR;
// ACLK tiene una frecuencia de 32768 Hz
// Carga cuenta en TA1CCR0 0.1seg TA1CCR=(0,1*32768)-1
// se disparará aprox. cada 0.1s
TA1CCR0 = 3276;
TA1CCTL0 = CCIE; // Habilita interrupción (bit CCTIMER1_A0IE) en TIMER1_A0



// Rutina de interrupción de TIMER1_A0
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR(void){
    P1OUT ^= 0x01;        // conmuta LED en P1.0
}
```

4. Ejercicios para la placa de desarrollo MSP430FR6989

1. Generar un programa en C embebido que utilizando polling encienda o apague el led rojo en función del evento del switch P1.1 de la placa.

2. Generar un programa en C embebido que utilizando interrupciones encienda o apague el led rojo en función de la captura del evento producido por el switch P1.1 de la placa.

3. Generar un programa en C embebido que utilizando el contador TIMER0 del microprocesador (inicializado a 40.000) y a través del método de interrupciones, haga parpadear el led rojo en función del delay introducido por el timer0.

4. Generar un programa en C embebido que utilizando el contador TIMER0 del microprocesador (inicializado a 40.000) y a través del método de interrupciones, muestre el conteo en el LCD de la placa añadiendo el control de los dos switches (botones) para Parar/Continuar y Reiniciar el contador:

   - Botón1: La parada del contador cuando se pulsa el switch. Si se vuelve a pulsar continua el conteo desde el valor actual.
   - Botón2: Reiniciar el contador a 0.

Se proporciona el siguiente código para trabajar con el LCD:

```
//*********************************
// Configures ACLK to 32 KHz crystal
void config_ACLK_to_32KHz_crystal() {
    // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz
    // Reroute pins to LFXIN/LFXOUT functionality
    PJSEL1 &= ~BIT4;
    PJSEL0 |= BIT4;
    // Wait until the oscillator fault flags remain cleared
    CSCTL0 = CSKEY; // Unlock CS registers
    do {
        CSCTL5 &= ~LFXTOFFG; // Local fault flag
        SFRIFG1 &= ~OFIFG; // Global fault flag
    } while((CSCTL5 & LFXTOFFG) != 0);
    CSCTL0_H = 0; // Lock CS registers
    return;
}

const unsigned char LCD_Num[10] = {0xFC, 0x60, 0xDB, 0xF3, 0x67, 0xB7, 0xBF, 0xE0, 0xFF,
0xE7};
//***************function that displays any 16-bit unsigned integer************
inline void display_num_lcd(unsigned int n){

    int i = 0;
    do {
        unsigned int digit = n % 10;
        switch(i){
        case 0: LCDM8    = LCD_Num[digit]; break;    //first digit
        case 1: LCDM15   = LCD_Num[digit]; break;    //second digit
        case 2: LCDM19   = LCD_Num[digit]; break;    //third digit
        case 3: LCDM4    = LCD_Num[digit]; break;    //fourth digit
        case 4: LCDM6    = LCD_Num[digit]; break;    //fifth digit
        }
        n /= 10;
        i++;
    } while(i < 5);
}

//*********************************************************
// Initializes the LCD_C module
// *** Source: Function obtained from MSP430FR6989 Sample Code ***
void Initialize_LCD() {
    PJSEL0 = BIT4 | BIT5; // For LFXT
    // Initialize LCD segments 0 - 21; 26 - 43
    LCDCPCTL0 = 0xFFFF;
    LCDCPCTL1 = 0xFC3F;
    LCDCPCTL2 = 0x0FFF;
    // Configure LFXT 32kHz crystal
    CSCTL0_H = CSKEY >> 8; // Unlock CS registers
    CSCTL4 &= ~LFXTOFF; // Enable LFXT
    do {
        CSCTL5 &= ~LFXTOFFG; // Clear LFXT fault flag
        SFRIFG1 &= ~OFIFG;
    }while (SFRIFG1 & OFIFG); // Test oscillator fault flag

    CSCTL0_H = 0; // Lock CS registers
    // Initialize LCD_C
    // ACLK, Divider = 1, Pre-divider = 16; 4-pin MUX
    LCDCCTL0 = LCDDIV__1 | LCDPRE__16 | LCD4MUX | LCDLP;
    // VLCD generated internally,
    // V2-V4 generated internally, v5 to ground
```

```
    // Set VLCD voltage to 2.60v
    // Enable charge pump and select internal reference for it
    LCDCVCTL = VLCD_1 | VLCDREF_0 | LCDCPEN;
    LCDCCPCTL = LCDCPCLKSYNC; // Clock synchronization enabled
    LCDCMEMCTL = LCDCLRM; // Clear LCD memory
    //Turn LCD on
    LCDCCTL0 |= LCDON;
    return;
}
```