EEL 4742C: Embedded Systems

Christopher Badolato
10/07/2019
EEL4742-0011
LAB 4 Interrupt and Low-power Modes

Introduction

In this lab we will learn to program with interrupts. First, we will use timer interrupts to toggle the LEDs with continuous mode and up mode. We will set an interrupt flag which means we do not need to poll the timer's flag constantly. Instead, when a time period elapses an interrupt flag raises, and the hardware then invokes a special intrinsic function to raise this flag. These functions are called an Interrupt Service Routine (ISR). The hardware performs a lookup on the Vector Table to find these ISRs. A vector is a start address of all the ISRs in memory. We can then program within this ISR interrupt to ensure our interrupts occur when intended.

Next, we then program a similar interrupt but with a push button interrupt. Instead of constantly polling the button press, we invoke an ISR that will raise a hardware flag when a button is pressed. We can program within the ISR on button push.

Finally, we will recreate all of these experiments with their respective intrinsic low power mode functions which will turn off the MCU when there is no interrupt signal. If we see the interrupt flag, the This will save power

Part 4.1: ...

Below we are writing code that runs Timer_A in continuous mode using only the ACLCK running at 32KHz. Each time our TAR rolls back to 0, the ISR will toggle the LEDs using a timer interrupt. We will link the ISR to TIMER0_A1_VECTOR using the pragma line.
To ensure that interrupts are enabled we must include the intrinsic function `_enable_interrupts();` which will enable the GIE bit. This bit will ensure that interrupts are enabled and will run the code within our ISR.

What happens if we don't clear the flag in the ISR? Explain.

If we don't clear the flag in the ISR the loop will continue running with the timer forever since our incrementing bit will never know when it rolls back.

What is the CPU doing between interrupts?

Waiting for the next interrupt, once we encounter an interrupt, the MCU engages active mode automatically and launches corresponding ISR. Then returns to low power mode and waits for next interrupt.

Who is calling the ISR? Is it the software? Explain.

The intrinsic function is call (enable_interrupts();) the ISR from the vector table.

Submit the code in your report.

```
//Christopher Badolato
//9/23/2019
//LAB 4.1
//EEL 4742L-0011

// Timer_A continuous mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>


#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7

void config_ACLK_to_32KHz_crystal();

void main(void) {
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT &= ~greenLED; // Turn LED Off
        // Configure ACLK to the 32 KHz crystal
    config_ACLK_to_32KHz_crystal();
        // Timer_A configuration (fill the line below)
        // Use ACLK, divide by 1, continuous mode, TAR cleared, enable
        //interrupt for rollback-to-zero event

    TA0CTL = (TASSEL_1|ID_0|MC_2|TACLR|TAIE);
            // Ensure the flag is cleared at the start
    TA0CTL &= ~TAIFG;
        // Enable the global interrupt bit (call an intrinsic function)
    _enable_interrupts();
        // Infinite loop... the code waits here between interrupts
    for(;;) {}
}
    //******* Writing the ISR *******
    #pragma vector = TIMER0_A1_VECTOR // Link the ISR to the vector

__interrupt void T0A1_ISR() {
            // Toggle both LEDs
        P9OUT ^= greenLED;
        P1OUT ^= redLED;

        TA0CTL &= ~TAIFG;
}


//*********************************
// 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;
}
```

Part 4.2: ...

In this part of the lab we use the clock in up-mode. The ISR in this case automatically sets the CCIFG flag, so we can just simply confirm that it is set to 0 at the start then ignore it, the ISR will take care of the rest. We place our light toggles within the ISR interrupt function without a flag reset bceause the hardware will take care of it. The lights will alternate flashing with a 1 second delay between.

```
//Christopher Badolato
//9/23/2019
//LAB 4.2
//EEL 4742L-0011

// Timer_A up mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7

void config_ACLK_to_32KHz_crystal();

void main(void) {
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT |= greenLED; // Turn LED On (alternate flashing)

    // Configure ACLK to the 32 KHz crystal
    config_ACLK_to_32KHz_crystal();

    // Configure Channel 0 for up mode with interrupt
    TA0CCR0 = (32768-1); // Fill to get 1 second @ 32 KHz
    TA0CCTL0 |= CCIE; // Enable Channel 0 CCIE bit
    TA0CCTL0 &= ~CCIFG; // Clear Channel 0 CCIFG bit

    // Timer_A: ACLK, div by 1, up mode, clear TAR (leaves TAIE=0)
    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
    // Enable the global interrupt bit (call an intrinsic function)
    _enable_interrupts();
    for(;;) {}
}
//*******************************
```

```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR() {
    // Toggle the LEDs
    P9OUT ^= greenLED;
    P1OUT ^= redLED;
    // Hardware clears the flag (CCIFG in TA0CCTL0)
}

//*********************************
// 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;
}
```

Part 4.3.1

For this part of the experiment we no longer need the clock. We will now use interrupts to signal when a button has been pressed. So, we need to find a new ISR vector function. Using the msp430fr6989.h we can find the ISR that corresponds to the button presses, in this case the vector is called "**#pragma** vector = PORT1_VECTOR" which will signal an interrupt for all port 1 pins. This means that this function can be used for both button s1 and button s2. Once we have the vector name, we can simply fill in our ISR function with our intended button press interrupt functions. If s1 is pressed, turn on red LED, if s2 is pressed, turn on green led. Both will also shut the LED off on second press toggling the light on press.

Is the code working flawlessly? Some buttons bounce when pushed (oscillate multiple times between low and high) and end up raising multiple interrupts in one push. Test each button by pushing it 20 or 30 times until you observe some cases of failure (i.e: you push the button and the LED doesn't toggle).
Roughly, what is the success rate of this code?

No, it is not running flawlessly. I tested each LED 40 times.
11/40 and 13/40 success rates are
72.5% and 67.5%
They sometimes double toggle or, don't turn on at all.

```
//Christopher Badolato
//9/23/2019
//LAB 4.3
//EEL 4742L-0011


// Timer_A continuous mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>


#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7
#define BUT1 BIT1 // Button S1 at Port 1.1
#define BUT2 BIT2 // Button S2 at Port 1.2


void main(void) {
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT &= ~greenLED; // Turn LED Off
    // Configuring buttons with interrupt
    P1DIR &= ~(BUT1|BUT2); // 0: input
    P1REN |= (BUT1|BUT2); // 1: enable built-in resistors
    P1OUT |= (BUT1|BUT2); // 1: built-in resistor is pulled up to Vcc
    P1IE |= (BUT1|BUT2); // 1: enable interrupts
    P1IES |= (BUT1|BUT2);  // 1: interrupt on falling edge
    P1IFG &= ~(BUT1|BUT2);  // 0: clear the interrupt flags
        // Enable the global interrupt bit (call an intrinsic function)
    _enable_interrupts();
        // Infinite loop... the code waits here between interrupts
    for(;;) {}
}

#pragma vector = PORT1_VECTOR // Write the vector name
__interrupt void Port1_ISR() {
        // Detect button 1 (BUT1 in P1IFG is 1)
    if ((P1IFG & BUT1) == BUT1) {
            // Toggle the red LED
        P1OUT ^= redLED;
            // Clear BUT1 in P1IFG
        P1IFG &= ~(BUT1);
    }
        // Detect button 2 (BUT2 in P1IFG is 1)
    if ((P1IFG & BUT2) == BUT2) {
            // Toggle the green LED
        P9OUT ^= greenLED;
            // Clear BUT2 in P1IFG
        P1IFG &= ~(BUT2);
    }
```

```
}
```

Part 4.4.1

Each of the examples below are the same as the ones above with configuration for Low power mode(_low_power_mode_x();). The low power modes give us options to turn off select CPU clocks while we wait for interrupts. This function enables the interrupts and acts as the continuous for loop that's running the board clocks.

Which low-power mode did you choose for each of the three codes? Explain your choices.
4.4.1: LPM3
4.4.2: LPM3
4.4.3: LPM3

Each of our cases in the lab we are only using the ACLK so we can see from the table below that we need to use _low_power_mode_3(); for each of our programs.

Mode	MCLK	SMCLK	ACLK
Active mode	on	on	on
LPM0	x	on	on
LPM3	x	x	on
LPM4	x	x	x

```
//Christopher Badolato
//9/23/2019
//LAB 4.4.1
//EEL 4742L-0011


// Timer_A continuous mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>


#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7

void config_ACLK_to_32KHz_crystal();

void main(void) {
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT &= ~greenLED; // Turn LED Off
```

```
    // Configure ACLK to the 32 KHz crystal
    config_ACLK_to_32KHz_crystal();
    // Timer_A configuration (fill the line below)
    // Use ACLK, divide by 1, continuous mode, TAR cleared, enable
    //interrupt for rollback-to-zero event

    TA0CTL = (TASSEL_1|ID_0|MC_2|TACLR|TAIE);

            // Ensure the flag is cleared at the start
    TA0CTL &= ~TAIFG;
        //Infinite loop... the code waits here between interrupts
        //while in LPM4;
    _low_power_mode_3();
}

    //******* Writing the ISR *******
    #pragma vector = TIMER0_A1_VECTOR // Link the ISR to the vector

__interrupt void T0A1_ISR() {
    // Toggle both LEDs
        P9OUT ^= greenLED;
        P1OUT ^= redLED;
    // Clear the TAIFG flag
        TA0CTL &= ~TAIFG;
}


//*********************************
// 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;
}
```

Part 4.4.2

This version is similar to 4.2 with low power mode enabled.

```
//Christopher Badolato
//9/23/2019
//LAB 4.4.2
//EEL 4742L-0011
```

```
// Timer_A up mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7

void config_ACLK_to_32KHz_crystal();

void main(void) {
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT |= greenLED; // Turn LED On (alternate flashing)

    // Configure ACLK to the 32 KHz crystal
    config_ACLK_to_32KHz_crystal();

    // Configure Channel 0 for up mode with interrupt
    TA0CCR0 = (32768-1); // Fill to get 1 second @ 32 KHz
    TA0CCTL0 |= CCIE; // Enable Channel 0 CCIE bit
    TA0CCTL0 &= ~CCIFG; // Clear Channel 0 CCIFG bit

    // Timer_A: ACLK, div by 1, up mode, clear TAR (leaves TAIE=0)
    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
    //LPM4;
    _low_power_mode_3();

}
//******************************
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR() {
    // Toggle the LEDs
    P9OUT ^= greenLED;
    P1OUT ^= redLED;
    // Hardware clears the flag (CCIFG in TA0CCTL0)
}



//*********************************
// 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;
}
```

Part 4.4.3

This version is similar to 4.3 with low power mode enabled.

```
//Christopher Badolato
//9/23/2019
//LAB 4.4.3
//EEL 4742L-0011


// Timer_A continuous mode, with interrupt, flashes LEDs
#include <msp430fr6989.h>


#define redLED BIT0 // Red LED at P1.0
#define greenLED BIT7 // Green LED at P9.7
#define BUT1 BIT1 // Button S1 at Port 1.1
#define BUT2 BIT2 // Button S2 at Port 1.2


void main(void) {

    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5; // Enable the GPIO pins
    P1DIR |= redLED; // Direct pin as output
    P9DIR |= greenLED; // Direct pin as output
    P1OUT &= ~redLED; // Turn LED Off
    P9OUT &= ~greenLED; // Turn LED Off

    // Configuring buttons with interrupt
    P1DIR &= ~(BUT1|BUT2); // 0: input
    P1REN |= (BUT1|BUT2); // 1: enable built-in resistors
    P1OUT |= (BUT1|BUT2); // 1: built-in resistor is pulled up to Vcc
    P1IE |= (BUT1|BUT2); // 1: enable interrupts
    P1IES |= (BUT1|BUT2);  // 1: interrupt on falling edge
    P1IFG &= ~(BUT1|BUT2);  // 0: clear the interrupt flags

        // Enable the global interrupt bit (call an intrinsic function)
        // Infinite loop... the code waits here between interrupts
    _low_power_mode_3();
}

#pragma vector = PORT1_VECTOR // Write the vector name
__interrupt void Port1_ISR() {
        // Detect button 1 (BUT1 in P1IFG is 1)
    if ((P1IFG & BUT1) == BUT1) {
            // Toggle the red LED
        P1OUT ^= redLED;
            // Clear BUT1 in P1IFG
```

```
        P1IFG &= ~(BUT1);
    }
        // Detect button 2 (BUT2 in P1IFG is 1)
    if ((P1IFG & BUT2) == BUT2) {
            // Toggle the green LED
        P9OUT ^= greenLED;
            // Clear BUT2 in P1IFG
        P1IFG &= ~(BUT2);
    }
}
```

Student Q&A

1. Explain the difference between using a low-power mode and not. What would be the CPU doing between interrupts for each case?

When low power mode is engaged the CPU shuts down and possibly other clock signals as well. The only way to reactivate the CPU is via an interrupt. Anytime we enter the low power mode we enable some interrupts and configure interrupt events.

2. We're using a module, e.g. the ADC converter, and we're not sure about the vector name. We expect it should be something like ADC VECTOR. Where do we find the exact vector name?

We can locate the Vector name needed within the header file included in the MSP430 code.

C:\...\ccsv7\ccs_base\msp430\include_gcc\msp430fr6989.h

3. A vector, therefore the ISR, is shared between multiple interrupt events. Who is responsible for clearing the interrupt flags?

The programmer will need to clear the interrupt flags in to let the code know when we have rolled back to 0, and when to reset the clock or if a button is pressed. in

4. A vector, and its corresponding ISR, is used by one interrupt event exclusively. Who is responsible for clearing the interrupt flag?

Once again, the programmer is responsible for clearing the interrupt flag of each vector when the interrupt occurs.

5. In the first code, the ISR's name is T0A1 ISR. Is it allowed we rename the function to any other name?

No, unless we change the channel of the timer to Timer_A then the code should remain the same.
T0A0 = channel 0
T0A1 = channel 1

6. What happens if the ISR is supposed to clear the interrupt flag and it didn't?

The interrupt will occur one time, but after that the flag will stay 1 and the ISR will not be run again.

Conclusion

In conclusion, in this lab experiment we have learned how to control our microcontroller with timer and button interrupts. First, we configured the clock in continuous and up modes, when a timer interrupt is raised, we toggled the lights and reset the flag. Next, the interrupts are configured to buttons, when the button press interrupt is raised, the corresponding light will turn on. Finally, we recreated each version of the experiment with the corresponding low power modes to save power in between interrupts.