Lab 4 – Interrupts & Low-Power Modes

# Youssef Samwel

# [yo800238@ucf.edu](mailto:yo800238@ucf.edu)

# EEL4742C Embedded Systems

Prof. Dr. Zakhia Abichar - Section 00419

# 2/8/2024



# **Project Description**

In this lab, we will learn programming interrupts and using the low-power modes. We will apply these mechanisms to the timer and to the push buttons.

# **2.0 Experiment Code**

## 2.2 Code that Flashes the LEDs

void ContinousModeInterrupt()

{

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_CONTINUOUS | TACLR | TAIE;

    \_enable\_interrupts();

}

#pragma vector = TIMER0\_A0\_VECTOR

\_\_interrupt void ISR\_T0A1()

{

    ToggleRedLED();

    ToggleGreenLED();

}

int main(void)

{

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    config\_ACLK\_to\_32KHz\_crystal();

    P1DIR |= redLED;   // Direct pin as output

    P9DIR |= greenLED; // Direct pin as output

    ContinousModeInterrupt();

    \_low\_power\_mode\_3();

    return 0;

}

The code configures buttons S1 and S2 for input and configures the timer A to continuous mode using ACLK clock source. When timer is interrupted, the ISR toggles both red and green LEDs.

What happens if we don’t clear the flag each time an interrupt occurs? Explain.

In this case, Timer0A0, the interrupt flag is cleared by the hardware because this ISR has only one source.

What is the CPU doing between interrupts?

The CPU is in sleep mode doing nothing to conserve energy.

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

The hardware is calling the ISR once the interrupt flag is triggered by the hardware. The ISR is defined by the software and the developer point’s to the ISR function by using vector table.

## 2.4 Timer’s Up Mode with Interrupt

void UpModeInterrupt()

{

    TA0CCR0 = 6535;

    TA0CCTL0 |= CCIE;

    TA0CCTL0 &= ~CCIFG;

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_UP | TACLR;

    \_enable\_interrupts();

}

#pragma vector = TIMER0\_A0\_VECTOR

\_\_interrupt void ISR\_T0A1()

{

    ToggleRedLED();

    ToggleGreenLED();

}

int main(void)

{

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    config\_ACLK\_to\_32KHz\_crystal();

    P1DIR |= redLED;   // Direct pin as output

    P9DIR |= greenLED; // Direct pin as output

    UpModeInterrupt();

    \_low\_power\_mode\_3();

    return 0;

}

Should we set TAIE to 1 in this code? Explain.

Yes, because TAIE enables ISR interrupt, otherwise the hardware will not invoke the ISR when the interrupt occurs.

Should the ISR clear the flag of Channel 0? Explain.

Yes, because there is only one source and therefore the hardware will clear the interrupt flag once the ISR exits.

Modify the code so that the delay is 0.5 seconds (then try 0.1 seconds).

For 0.5 seconds, set TA0CCR0 to 16,384 and for 0.1 seconds: 3276

## 2.6 Push Button with Interrupt

#pragma vector = PORT1\_VECTOR

\_\_interrupt void ISR\_PORT1() {

    if (P1IFG & BUTTON1) {

        ToggleRedLED();

    }

    if (P1IFG & BUTTON2) {

        ToggleGreenLED();

    }

    // delay 20ms to combat button debouncing

    \_\_delay\_cycles(5e5);

    // clear interrupt flags

    P1IFG &= ~(BUTTON1 | BUTTON2);

}

void PushButtonInterrupt()

{

    P1DIR &= ~(BUTTON1 | BUTTON2);

    P1REN |= BUTTON1 | BUTTON2;

    P1OUT |= BUTTON1 | BUTTON2;

    P1IE |= BUTTON1 | BUTTON2;

    P1IES = BUTTON1 | BUTTON2;

    P1IFG &= (BUTTON1 | BUTTON2);

    \_enable\_interrupts();

}

int main(void)

{

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    config\_ACLK\_to\_32KHz\_crystal();

    P1DIR |= redLED;   // Direct pin as output

    P9DIR |= greenLED; // Direct pin as output

    PushButtonInterrupt();

    \_low\_power\_mode\_3();

    return 0;

}

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.

The code worked flawlessly after I increased the delay to 25 milliseconds. After that change I did not experience multiple interrupts for “one” user click. I observed failure when I had a lower delay value.

Roughly, what is the success rate of this code?

With 25 milliseconds, 100% with the lower delay value it was 6/10.

## 2.8 Application: Crawler Guidance System

void ApplicationCrawlerGuidenceSystem\_4\_5()

{

    // Set input direction for buttons

    P1DIR &= ~(BUTTON1 | BUTTON2);

    // enable pull-up resistors to avoid false triggers

    P1REN |= BUTTON1 | BUTTON2;

    // pull high buttons

    P1OUT |= BUTTON1 | BUTTON2;

    // interrupt on falling edge

    P1IES = BUTTON1 | BUTTON2;

    // reset interrupt flags

    P1IFG &= (BUTTON1 | BUTTON2);

    // enable PORT1 interrupts

    P1IE |= BUTTON1 | BUTTON2;

    // Up mode counter, 2x division, 32kHz clock module, interrupt enable

    // CCIE interrupts when target value in TA0CCR0 is reached, no flags need

    // to be cleared in TIMER0 since the ISR is not shared with other timers

    // TACLR resets the timer to start at 0

    TA0CTL = MC\_\_UP | ID\_\_1 | TASSEL\_\_ACLK | CCIE | TACLR;

    // Interrupt when timer reaches the following value

    TA0CCR0 = 2e3;

    TA0CCTL0 |= CCIE;

    TA0CCTL0 &= ~CCIFG;

    LightRedLED(false);

    LightGreenLED(false);

    \_enable\_interrupts();

}

#pragma vector = TIMER0\_A0\_VECTOR

\_\_interrupt void ISR\_T0A1() {

    static int8\_t g\_CycleCounter = CYCLE\_COUNT;

    int8\_t direction = g\_DirectionCounter;

    if (direction == 0 && g\_CycleCounter % 10 == 0)

    {

        ToggleRedLED();

        ToggleGreenLED();

    }

    else if (direction > 0 || direction < 0)

    {

        uint8\_t delay = (direction > 0) ? 1 : 1;

        switch (direction)

        {

            case 3: delay = 2; break;

            case 2: delay = 3; break;

            case 1: delay = 5; break;

            case -3: delay = 2; break;

            case -2: delay = 3; break;

            case -1: delay = 5; break;

        }

        if (g\_CycleCounter % delay == 0)

        {

            if (direction > 0)

            {

                ToggleRedLED();

                LightGreenLED(false);

            }

            else

            {

                ToggleGreenLED();

                LightRedLED(false);

            }

        }

    }

    g\_CycleCounter--;

    if (g\_CycleCounter < 0)

        g\_CycleCounter = CYCLE\_COUNT;

}

#pragma vector = PORT1\_VECTOR

\_\_interrupt void ISR\_PORT1() {

    if (IsButton1Pressed())

        g\_DirectionCounter++;

    if (IsButton2Pressed())

        g\_DirectionCounter--;

    // clamp value

    g\_DirectionCounter = g\_DirectionCounter > 3 ? 3 : g\_DirectionCounter;   // max

    g\_DirectionCounter = g\_DirectionCounter < -3 ? -3 : g\_DirectionCounter; // min

    if (g\_DirectionCounter == 0)

    {

        LightRedLED(false);

        LightGreenLED(false);

    }

    // delay 20ms for button debouncing

    \_\_delay\_cycles(5e5);

    // clear interrupt flags

    P1IFG &= ~(BUTTON1 | BUTTON2);

}

int main(void) {

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    config\_ACLK\_to\_32KHz\_crystal();

    P1DIR |= redLED;   // Direct pin as output

    P9DIR |= greenLED; // Direct pin as output

    ApplicationCrawlerGuidenceSystem\_4\_5();

    \_low\_power\_mode\_3();

    return 0;

}

## 3.0 Complete Code

#include <msp430fr6989.h>

#include <stdint.h>

#include <stdbool.h>

#define BUTTON1 BIT1

#define BUTTON2 BIT2

#define redLED BIT0   // Red LED at P1.0

#define greenLED BIT7 // Green LED at P9.7

#define SHOW\_APPLICATION 1

#define CYCLE\_COUNT 100

static int8\_t g\_DirectionCounter = 0;

void LightRedLED(bool state) {

    if (state)

        P1OUT |= redLED;

    else

        P1OUT &= ~redLED;

}

void LightGreenLED(bool state) {

    if (state)

        P9OUT |= greenLED;

    else

        P9OUT &= ~greenLED;

}

void ToggleRedLED() { P1OUT ^= redLED; }

void ToggleGreenLED() { P9OUT ^= greenLED; }

bool IsButton1Pressed() {

    return (~P1IN & BUTTON1) ? true : false;

}

bool IsButton2Pressed() {

    return (~P1IN & BUTTON2) ? true : false;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 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;

}

#pragma vector = TIMER0\_A1\_VECTOR

\_\_interrupt void ISR\_T1A1() {

    ToggleRedLED();

    ToggleGreenLED();

    TA0CTL &= ~TAIFG;

}

#pragma vector = TIMER0\_A0\_VECTOR

\_\_interrupt void ISR\_T0A1() {

#if !SHOW\_APPLICATION

    ToggleRedLED();

    ToggleGreenLED();

#else

    static int8\_t g\_CycleCounter = CYCLE\_COUNT;

    int8\_t direction = g\_DirectionCounter;

    if (direction == 0 && g\_CycleCounter % 10 == 0) {

        ToggleRedLED();

        ToggleGreenLED();

    }

    else if (direction > 0 || direction < 0) {

        uint8\_t delay = (direction > 0) ? 1 : 1;

        switch (direction) {

            case 3: delay = 2; break;

            case 2: delay = 3; break;

            case 1: delay = 5; break;

            case -3: delay = 2; break;

            case -2: delay = 3; break;

            case -1: delay = 5; break;

        }

        if (g\_CycleCounter % delay == 0) {

            if (direction > 0) {

                ToggleRedLED();

                LightGreenLED(false);

            }

            else {

                ToggleGreenLED();

                LightRedLED(false);

            }

        }

    }

    g\_CycleCounter--;

    if (g\_CycleCounter < 0)

        g\_CycleCounter = CYCLE\_COUNT;

#endif

}

#pragma vector = PORT1\_VECTOR

\_\_interrupt void ISR\_PORT1()

{

#if !SHOW\_APPLICATION

    if (P1IFG & BUTTON1)

    {

        ToggleRedLED();

    }

    if (P1IFG & BUTTON2)

    {

        ToggleGreenLED();

    }

#else

    if (IsButton1Pressed())

    {

        g\_DirectionCounter++;

    }

    if (IsButton2Pressed())

    {

        g\_DirectionCounter--;

    }

    // clamp value

    g\_DirectionCounter = g\_DirectionCounter > 3 ? 3 : g\_DirectionCounter;   // max

    g\_DirectionCounter = g\_DirectionCounter < -3 ? -3 : g\_DirectionCounter; // min

    if (g\_DirectionCounter == 0)

    {

        LightRedLED(false);

        LightGreenLED(false);

    }

#endif

    // delay 20ms for button debouncing

    \_\_delay\_cycles(5e5);

    // clear interrupt flags

    P1IFG &= ~(BUTTON1 | BUTTON2);

}

void ContinousModeInterrupt() {

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_CONTINUOUS | TACLR | TAIE;

    \_enable\_interrupts();

}

void UpModeInterrupt() {

    TA0CCR0 = 6535;

    TA0CCTL0 |= CCIE;

    TA0CCTL0 &= ~CCIFG;

    TA0CTL = TASSEL\_\_ACLK | ID\_\_1 | MC\_\_UP | TACLR;

    \_enable\_interrupts();

}

void PushButtonInterrupt() {

    P1DIR &= ~(BUTTON1 | BUTTON2);

    P1REN |= BUTTON1 | BUTTON2;

    P1OUT |= BUTTON1 | BUTTON2;

    P1IE |= BUTTON1 | BUTTON2;

    P1IES = BUTTON1 | BUTTON2;

    P1IFG &= (BUTTON1 | BUTTON2);

    \_enable\_interrupts();

}

void ApplicationCrawlerGuidenceSystem\_4\_5() {

    // Set input direction for buttons

    P1DIR &= ~(BUTTON1 | BUTTON2);

    // enable pull-up resistors to avoid false triggers

    P1REN |= BUTTON1 | BUTTON2;

    // pull high buttons

    P1OUT |= BUTTON1 | BUTTON2;

    // interrupt on falling edge

    P1IES = BUTTON1 | BUTTON2;

    // reset interrupt flags

    P1IFG &= (BUTTON1 | BUTTON2);

    // enable PORT1 interrupts

    P1IE |= BUTTON1 | BUTTON2;

    // Up mode counter, 2x division, 32kHz clock module, interrupt enable

    // CCIE interrupts when target value in TA0CCR0 is reached, no flags need

    // to be cleared in TIMER0 since the ISR is not shared with other timers

    // TACLR resets the timer to start at 0

    TA0CTL = MC\_\_UP | ID\_\_1 | TASSEL\_\_ACLK | CCIE | TACLR;

    // Interrupt when timer reaches the following value

    TA0CCR0 = 2e3;

    TA0CCTL0 |= CCIE;

    TA0CCTL0 &= ~CCIFG;

    LightRedLED(false);

    LightGreenLED(false);

    \_enable\_interrupts();

}

int main(void) {

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    config\_ACLK\_to\_32KHz\_crystal();

    P1DIR |= redLED;   // Direct pin as output

    P9DIR |= greenLED; // Direct pin as output

    ApplicationCrawlerGuidenceSystem\_4\_5();

    \_low\_power\_mode\_3();

    return 0;

}

# **4.0 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 using the polling method, the CPU is constantly wasting power by checking the interrupt flag, whereas, with low power mode and interrupts, the CPU is “sleeping” between interrupts and therefore reducing power usage (extending battery life).

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?

In the family guide of the MSP430.

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

The software is responsible for clearing the ISR when it is shared by multiple sources.

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

The hardware is responsible for clearing the interrupt flag.

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

The function name can be anything, but the ISR vector name is constant is defined in the datasheet.

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

The ISR will get continuously invoked.

# **5.0 Conclusion**

In conclusion, this lab focused on implementing interrupts and low-power modes in an embedded system using MSP430FR6989. The experiments involved configuring timers for continuous and up modes, handling push button interrupts, and developing a practical application - the Crawler Guidance System. The importance of clearing interrupt flags to prevent continuous invocation of the ISR was highlighted. Additionally, the advantages of utilizing low-power modes over constant polling were discussed, emphasizing the reduction in power consumption during idle periods. The successful implementation of these concepts was demonstrated through code examples and practical testing, providing valuable insights into efficient embedded systems design.