**10/11/2020**

Joshua Frazer

Lab 4

Interrupts & Low-Power Modes

**Introduction:**

For this lab, we learned the basics of interrupts. For part 1, we get introduced to incorporating an interrupt with continuous mode. Part 2 was very similar to part 1, except for using the up mode for the timer. Finally for part 3, we learned how to use an interrupt upon an action, such as a button.

**Part 1:**

Using previous knowledge from Lab 4, an interrupt became a simple addition to our previous code. Once we gathered the right intrinsic function and used the knowledge of toggling LEDs and clearing the TAIFG flag, this part was very straightforward.

// 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** **T0A1\_ISR**();

**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, enableinterrupt 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**(;;) {}

// // Low Power Mode

//

// \_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

P1OUT ^= redLED;

P9OUT ^= greenLED;

// 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**;

}

**Questions:**

* What happens if we don’t clear the flag in the ISR? Explain.
  + **If we do not clear the flag in the ISR, we would be constantly toggling the LEDs. This is because we would be constantly within the ISR, due to the flag never being cleared.**
* What is the CPU doing between interrupts?
  + **The CPU is on and operating between interrupts.**
* Who is calling the ISR? Is it the software? Explain.
  + **The software is calling the ISR. Every time the flag is activated the ISR gets called upon.**

**Part 2:**

This part carries on a lot of what I just did in part 1. The only changes were the linked vector of the ISR, and that we are to change our mode to up mode. We are also tasked to achieve one second for the timer’s period.

// 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** **T0A0\_ISR**();

**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();

// Configure Channel 0 for up mode with interrupt

TA0CCR0 = 32767; // 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();

// Infinite loop... the code waits here between interrupts

**for**(;;) {}

// // Low Power Mode

//

// \_low\_power\_mode\_3();

}

//\*\*\*\*\*\*\*Writing the ISR\*\*\*\*\*\*\*

**#pragma** vector = TIMER0\_A0\_VECTOR // Link the ISR to the vector

**\_\_interrupt** **void** **T0A0\_ISR**() {

// Toggle both LEDs

P1OUT ^= redLED;

P9OUT ^= greenLED;

// 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**;

}

**Questions:**

* Modify the code so that the delay is 0.5 seconds (then try 0.1 seconds). Ensure the code works for both cases.
  + **0.5 Seconds: Change TA0CCR0 to 16383**
  + **0.1 Seconds: Change TA0CCR0 to 3275**

**Part 3:**

For Part 3, the goal is to get the LEDs to toggle upon the push of each button. S1 would toggle the red LED and S2 would toggle the green LED. In order to do this, we had to set up the right configuration for the button to activate the interrupt.

// Timer\_A button, 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** **Port1\_ISR**();

**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

// 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**(;;) {}

// // Low Power Mode

//

// \_low\_power\_mode\_4();

}

//\*\*\*\*\*\*\*Writing the ISR\*\*\*\*\*\*\*

**#pragma** vector = PORT1\_VECTOR // Write the vector name

**\_\_interrupt** **void** **Port1\_ISR**() {

// Detect button 1 (BUT1 in P1IFG is 1)

**if** ((BUT1 & P1IFG) == BUT1) {

// Toggle the red LED

P1OUT ^= redLED;

// Clear BUT1 in P1IFG

P1IFG &= ~BUT1;

}

// Detect button 2 (BUT2 in P1IFG is 1)

**if** ((BUT2 & P1IFG) == BUT2) {

// Toggle the green LED

P9OUT ^= greenLED;

// Clear BUT2 in P1IFG

P1IFG &= ~BUT2;

}

}

**Questions:**

* 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).
  + **The code is not working flawlessly.**
* Roughly, what is the success rate of this code?
  + **Red LED: 26 Successes/30 Pushes = 93.33% Success Rate**
  + **Green LED: 23 Successes/30 Pushes = 76.67% Success Rate**

**Part 4:**

Within part 4, we are tasked to select the proper low power saving mode for each of the parts within this lab. I have already done this in each part. Below each infinite for loop is the low power mode that I selected.

Part 1 modification:

// // Low Power Mode

//

// \_low\_power\_mode\_3();

Part 2 modification:

// // Low Power Mode

//

// \_low\_power\_mode\_3();

Part 3 modification:

// // Low Power Mode

//

// \_low\_power\_mode\_4();

**Questions:**

* Which low-power mode did you choose for each of the three codes? Explain your choices.
  + **Part 1: Low Power Mode 3 since ACLK is still being used consistently.**
  + **Part 2: Low Power Mode 3 since ACLK is still being used consistently.**
  + **Part 3: Low Power Mode 4 since clocks are not being used.**

**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?
   1. **The CPU would be sleeping between interrupts on low power mode. When the CPU is not in low power mode, then the CPU would stay on and continue operating.**
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 ADCVECTOR. Where do we find the exact vector name?
   1. **You can find the exact vector name of the ADC converter in the header file, msp430fr6989.h**
3. A vector, therefore the ISR, is shared between multiple interrupt events. Who is responsible for clearing the interrupt flags?
   1. **If a vector is shared, then it is the programmer’s responsibility to clear the flag.**
4. A vector, and its corresponding ISR, is used by one interrupt event exclusively. Who is responsible for clearing the interrupt flag?
   1. **If the vector is used by one interrupt event exclusively then the hardware is responsible for clearing the flag.**
5. In the first code, the ISR’s name is T0A1ISR. Is it allowed we rename the function to any other name?
   1. **We are not allowed to rename the function to any other name.**
6. What happens if the ISR is supposed to clear the interrupt flag and it didn’t?
   1. **Your vector may be shared.**

**Conclusion:**

The most significant part of this lab was all parts, in my opinion. In part 1, we were introduced to ISRs which, alone, was a handful. Then part 2, we were given a different vector which took me awhile to understand and find within the header file. As for Part 3, buttons were introduced, and aside from using the timer, this helped teach me how to configure a button to work with interrupts. As for the last part, I feel this was extremely essential to understand as I assume this is what we will be accomplishing in future labs.