

# Lab 11 – Concurrent Event Handling

Youssef Samwel

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

EEL4742C Embedded Systems

Prof. Dr. Zakhia Abichar - Section 00419

4/18/2024



## 1.0 Lab Description

In this lab, we will learn programming concurrent events that are processed with interrupts. We will learn how to program multiple interrupt events that interact with one another and how to program interrupts that are enabled/disabled in various phases of the program.

## 2.0 Experiment Code

HVAC

### Port 1 ISR

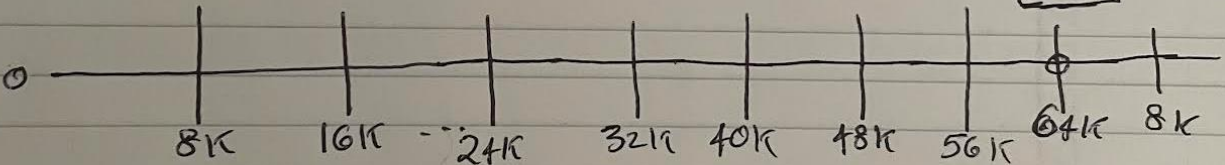
Timer 0 run using 32kHz ACLK, Continuous mode,  
12--2 Clock division

- Enable Channel 1 for Led flashing  
 $TACCTL1 = \frac{32\text{kHz}}{2} \cdot 0.5 = 8192$ 
  - Clear TAR
- Enable Channel 2 for toggling Compressor  
 $TACCTL2 = 49152$
- Disable Button Interrupts (only for 11.1)  
[For Part 11.2]

### Channel 1

Compressor On: Toggle Green led / Turn off Red Led  
Compressor Off: Opposite / Opposite  
 $TACCTL1 += 8192;$

Rest



Clear Interrupt flag

### Channel 2:

- Toggle Compressor
- Turn Off Channel 2 Interrupts
- Enable Port 1 Interrupts
- Clear Interrupt flag

# Button Debouncing

## Port 1 ISR

Start Timer 0, Continuous mode 20 millisecond interrupt  
Turn on Channel 0 with duration of 20 ms.  
Clear IFG Flag and disable button 1 interrupts to  
avoid resetting the timer.

## Channel 0 ISR:

```
if Button 1 is low (still pressed) {  
    toggle green led.  
}
```

Disable Channel 0 interrupts  
Re-enable button 1 interrupts.

```
#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 ENABLE_PART_3 0  
  
//*****  
// 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;
}

static bool isCompressorOn = false;

#pragma vector = PORT1_VECTOR
__interrupt void ISR_PORT1()
{
    if (P1IFG & BUTTON1)
    {
        #if ENABLE_PART_3
            TA0CTL = TASSEL__ACLK | ID__2 | MC__CONTINUOUS | TACLR;
            TA0CCR0 = 656; // about 40 ms delay
            TA0CCTL0 |= CCIE;
            P1IE = ~(BUTTON1);
        #else
            // Configure toggle compressor channel
            TA0CCR2 = TA0R + 49152; // 3 seconds delay
            TA0CCTL2 = CCIE;
            // Disable button interrupts
            // P1IE &= ~(BUTTON1);
            // the following condition is used for Part 2
            // where we renew the delay if the the button is pressed again.
            // we do this by re-enabling the button interrupts
            // this will work because if the interrupt occurs again
            // the ISR_PORT1() will re-configure TA0CTL which will clear
            // the timer 0 TAR register therefore the timer will reset
            __delay_cycles(4e4); // about 40 ms delay
        #endif
        P1IFG &= ~(BUTTON1);
    }
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void ISR_ButtonDebounce()
{
    // if button is still pressed, toggle green LED
    // otherwise do thing
    // disable timer and re-enable Button 1 interrupt
    if (~P1IN & BUTTON1)
    {
        // button is pressed
        P9OUT ^= greenLED;
    }
    TA0CCTL0 &= ~CCIE;
    P1IE |= BUTTON1;
}

```

```

#pragma vector = TIMER0_A1_VECTOR
__interrupt void ISR_Timer1()
{
    if (( (TA0CCTL1 & CCIFG) != 0) && ( (TA0CCTL1 & CCIE) != 0) )
    {
        P9OUT ^= greenLED;
        TA0CCR1 = TA0R + 8192;
        TA0CCTL1 &= ~CCIFG;
    }
    if (( (TA0CCTL2 & CCIFG) != 0) && ( (TA0CCTL2 & CCIE) != 0))
    {
        isCompressorOn = !isCompressorOn;
        // we are no longer needed, disable channel 2
        TA0CCTL2 &= ~CCIE;
        P1OUT ^= redLED;
        P1IE |= (BUTTON1);
        TA0CCTL2 &= ~CCIFG;
    }
}

/**
 * main.c
 */
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;    // Enable the GPIO pins

    // Set input direction for buttons
    P1DIR &= ~(BUTTON1);
    // enable pull-up resistors to avoid false triggers
    P1REN |= BUTTON1;
    // pull high buttons
    P1OUT |= BUTTON1;
    // interrupt on falling edge
    P1IES = BUTTON1;
    // reset interrupt flags
    P1IFG &= ~(BUTTON1);
    // enable PORT1 interrupts
    P1IE |= BUTTON1;

    config_ACLK_to_32KHz_crystal();

    __delay_cycles(2e5);

#if !ENABLE_PART_3
    TA0CTL = TASSEL__ACLK | ID__2 | MC__CONTINUOUS | TACLK;
    // channel configurations
    TA0CCR1 = 8192; // 0.5 seconds delay
    TA0CCTL1 &= ~CCIFG;
#endif
}

```

```

    TA0CCTL1 |= CCIE; // also clears the CCIFG flag
#endif

    P1DIR |= redLED;    // Direct pin as output
    P9DIR |= greenLED;  // Direct pin as output

    P9OUT &= ~greenLED;

    _low_power_mode_3();

    return 0;
}

```

### 3.0 Student Q&A

What is the maximum bounce duration that is set in your code?

About 40 milliseconds.

1. An interrupt uses a shared ISR and is always enabled in the program. What does the if-statement in the ISR check for before servicing this interrupt?

The if statement checks which timer channel caused the interrupt since multiple channels use the same ISR and may have different interrupt durations. This is done by checking the xIFG flag in TAxCTLx.

2. An interrupt uses a shared ISR and is enabled/disabled in various phases of the program. What does the if-statement in the ISR check for before servicing this interrupt?

The if statement checks which timer channel caused the interrupt since multiple channels use the same ISR and may have different interrupt durations. This is done by checking the xIFG flag in TAxCTLx.

3. For the debouncing algorithm that we implemented, is it possible that the LED will be toggled when the button is released? Explain.

No, because the timer will wait about 40 milliseconds and check if the button is still pressed and after this duration the button will be released and therefore the LED will not be toggled.

4. If two random pulses occur on the push button line due to noise and these pulses are separated by the maximum bounce duration, will our debouncing algorithm fail? Explain

It depends on how long the pulse lasts for. If the pulse lasts for more than 40 milliseconds then this will cause our algorithm to fail, however, due to the nature of random pulses being short in time duration it will most likely not fail.

### 4.0 Conclusion

In summary, this lab equipped us with practical knowledge in programming concurrent event handling using interrupts in embedded systems. Through experimentation with debouncing algorithms and dynamic interrupt management, we gained essential skills for building responsive and reliable systems capable of handling multiple events simultaneously.