Lab 4 Report

EEL4742C - 00446

Yousef Awad

September 2025

Contents

Introduction **2**

4.1 Timer's Continuous Mode with Interrupt **2**

4.2 Timer's Up Mode with Interrupt **3**

4.3 Push Button with Interrupt **5**

4.4 Low-Power Modes **6**

4.5 Application: Crawler Guidance System **6**

Student Q&A **9**

1 9
2 9
3 9
4 9
5 9
6 10

Introduction

I don't have time for this, sorry :(

4.1 Timer's Continuous Mode with Interrupt

The period of interrupts that I will see will be when the timer overflows or:

$$\frac{65536\ cycles}{32768\ \frac{cycles}{second}} = 2\ seconds$$

After checking this time with my phone, it lined up almost perfectly with slight error due to my hand not reacting as fast as it should have (I have a bad reaction time). If we do not clear the flag each time an interrupt occurs, then the ISR conditions will always occur forever, leading to the led constantly turning on and off, and never leaving the ISR function therefore breaking the purpose of the interrupt to begin with. Now, when the CPU is between interrupts, the CPU simply just idles and does nothing due to the forever loop in main. Now, as told before, the ISR is only called when the hardware signals that the timer overflows, therefore setting the TAIFG flag.

```
#include <inttypes.h>
#include <msp430fr6989.h>

#define red BIT0

// 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;
}

// This function writes to the Interrupt Signal Response via the
    __interrupt return type
#pragma vector = TIMER0_A1_VECTOR
__interrupt void signal_response()
{
    // Interrupt response goes here
    // On any given interrupt toggle led
    P1OUT ^= red;
    // clearing the time since the interrupt occured
```

```
32      TA0CTL &= ~TAIFG; // use TAIFG by default when timer is in
        continuous
33  }
34
35  int main(void)
36  {
37    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
38    PM5CTL0 &= ~LOCKLPM5;     // opening gpio
39
40    // setting direction to inputs and outputs of red, green, and
        buttons
41    P1DIR |= red;
42
43    // Setting green led as outputs
44    P1OUT &= ~red;
45
46    // configuring clock to 32khz
47    config_ACLK_to_32KHz_crystal();
48
49    // Configuring Timer_A
50    TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR | TAIE;
51    // TAIE enables interrupt for rollback to 0 by default
52
53    // clearing flag at start just in case TACLR fails, lmao
54    TA0CTL &= ~TAIFG;
55
56    _enable_interrupts(); // allows for interrupts globally hardware
        wise
57
58    // P1IFG is port 1's 8-bit register for interrupts for 8
        individual interrupt use-cases
59
60    for (;;)
61    {
62        // Infinite loop
63    }
64  }
```

4.2 Timer's Up Mode with Interrupt

Thankfully, the timing of the stopwatch lines up almost perfectly with the expected timing of the LEDs toggling, just as it did before. Alongside this, the TAIE doesn't need to be set to 1 as we are simply using the Capture and Compare registers meaning we use CCIE as 1 instead. For our ISR flag, we should clear channel 0, and do, due to the fact that this is the channel that the timer uses. If that channel is not cleared, then the ISR will therefore never be exited (uh oh!!). To change the code to 0.5 seconds or even 0.1, I would need to change the TA0CCR0 to 16384 or 3276 respectively.

```
1  #include <inttypes.h>
2  #include <msp430fr6989.h>
3
4  #define red BIT0
5  #define green BIT7
```

```
6
7  // Configures ACLK to 32 KHz crystal
8  void config_ACLK_to_32KHz_crystal()
9  {
10   // By default, ACLK runs on LFMODCLK at 5MHz/128 = 39 KHz
11   // Reroute pins to LFXIN/LFXOUT functionality
12   PJSEL1 &= ~BIT4;
13   PJSEL0 |= BIT4;
14   // Wait until the oscillator fault flags remain cleared
15   CSCTL0 = CSKEY; // Unlock CS registers
16   do
17   {
18     CSCTL5 &= ~LFXTOFFG; // Local fault flag
19     SFRIFG1 &= ~OFIFG;   // Global fault flag
20   } while ((CSCTL5 & LFXTOFFG) != 0);
21   CSCTL0_H = 0; // Lock CS registers
22   return;
23 }
24
25 // This function writes to the Interrupt Signal Response via the
       __interrupt return type
26 #pragma vector = TIMER0_A1_VECTOR
27 __interrupt void signal_response()
28 {
29     // Interrupt response goes here
30     // On any given interrupt toggle led
31     P1OUT ^= red;
32     P9OUT ^= green;
33     // clearing the time since the interrupt occured
34     TA0CCTL0 &= ~CCIFG; // use TAIFG by default when timer is in
       continuous
35 }
36
37 int main(void)
38 {
39   WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
40   PM5CTL0 &= ~LOCKLPM5;     // opening gpio
41
42   // setting direction to inputs and outputs of red, green, and
       buttons
43   P1DIR |= red;
44   P9DIR |= green;
45
46   // Setting green led as outputs
47   P1OUT &= ~red;
48   P9OUT &= green;
49
50
51   // configuring clock to 32khz
52   config_ACLK_to_32KHz_crystal();
53
54   // Configuring Timer_A
55   TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
56
57   TA0CCR0 = 32768; // 1 Second
58   TA0CCTL0 |= CCIE; // enabling the channel 0
59   TA0CCTL0 &= ~CCIFG; // clearing flag for channel 0
```

```
60
61    _enable_interrupts(); // allows for interrupts globally hardware
        wise
62
63    // P1IFG is port 1's 8-bit register for interrupts for 8
        individual interrupt use-cases
64    for (;;)
65    {
66        // Infinite loop
67    }
68  }
```

4.3 Push Button with Interrupt

The code *does not* work flawlessly sadly. It sadly has issues when the button presses are too close together leading to a staggered and/or awkward input and output delays/occurances. For me, the success rate is about 85% or $\frac{34}{40}$.

```
1  #include <msp430fr6989.h>
2  #include "aclk_config.h"
3
4  #define red BIT0      // Port 1.0
5  #define green BIT7    // Port 9.7
6
7  #define but1 BIT1     // Port 1.1
8  #define but2 BIT2     // Port 1.2
9
10 int main(void)
11 {
12   WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
13   PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins
14
15   P1DIR |= red;     // Set output for red led
16   P1OUT &= ~red;    // Turn off red LED
17
18   P9DIR |= green;   // Set output for green led
19   P9OUT &= ~green;  // Turn off green LED
20
21   // Configure buttons
22   P1DIR &= ~(but1 | but2);    // input
23   P1REN |= (but1 | but2);     // enable resistors
24   P1OUT |= (but1 | but2);     // pull-up
25   P1IES |= (but1 | but2);     // interrupt on falling edge
26   P1IE |= (but1 | but2);      // enable interrupts
27   P1IFG &= ~(but1 | but2);    // set interrupt flag off
28
29   // Enable global interrupt support
30   _enable_interrupts();
31
32   for (;;)
33   {
34     // Infinite Loop
35   }
36 }
37
```

```
#pragma vector = PORT1_VECTOR
__interrupt void PORT1_ISR()
{
  // S1 interrupt raised
  if (!(P1IFG & but1))
  {
    P1OUT ^= red;   // toggle red led
    P1IFG &= ~but1; // turn off flag for button1
  }
  // S2 interrupt raised
  if (!(P1IFG & but2))
  {
    P9OUT ^= green; // toggle green led
    P1IFG &= ~but2; // turn off flag for button 2
  }
}
```

4.4 Low-Power Modes

I chose to use low power mode 3 for all three revisions due to the fact that it is the lowest mode that still enables the auxillary clock that we need to ensure that flags are enabled using the timer. The only pieces of code that I changed was...

```
_enable_interrupts();
```

to the following...

```
_low_power_3();
```

4.5 Application: Crawler Guidance System

For what I've written below, I've set up a timer that uses the auxillary clock in up mode, with the TAR cleared by default, and the Capture and Command Interrupt Enable set to true. I specifically set up TA0CCR0 to be 32767 so that the flag raises every 1 second. Alongside this, I cleared TAIFG flag as well as enabled the low power mode 3. To track states I used an enum to track the state, so as to make my life easier and label things with words. This therefore made it so that when the button is pressed, it first turns off the LEDs, then checks the state and switches it up or down accordingly, then sets the correct TA0CCR0 for next flashing LED (either green or red or magical purple, right??????). After this, I reset the timer, and flags so that it can continue on its journey cycling infinitely through the states.

```
#include <msp430fr6989.h>
#include "aclk_config.h"

#define red BIT0     // Port 1.0
#define green BIT7   // Port 9.7

#define but1 BIT1    // Port 1.1
```

```
#define but2 BIT2    // Port 1.2

typedef enum {
    RedFast, RedNormal, RedSlow, RedGreen, GreenSlow, GreenNormal,
    GreenFast
} led_state_t;

// Volatile and global for access by both ISRs
volatile led_state_t state = RedGreen;

void crawler(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop the Watchdog timer
    PM5CTL0 &= ~LOCKLPM5;     // Enable the GPIO pins

    // Configure LEDs
    // Output Directions of LEDs
    P1DIR |= red;
    P9DIR |= green;
    // Turning off the LEDs
    P1OUT &= ~red;
    P9OUT &= ~green;

    // Configure buttons
    P1DIR &= ~(but1 | but2);    // Setting buttons as inputs
    P1REN |= (but1 | but2);     // Enable resistors for buttons
    P1OUT |= (but1 | but2);     // Set resistors as pull-up
    P1IES |= (but1 | but2);     // Set interrupt on falling edge
    P1IE  |= (but1 | but2);     // Enable interrupts proper
    P1IFG &= ~(but1 | but2);    // Set the interrupt flag off as
    sanity check

    // Configure 32Khz clock
    config_ACLK_to_32KHz_crystal();

    // ACLK | Divide by 1 | Up Mode | Clear Timer
    TA0CTL = TASSEL_1 | ID_0 | MC_1 | TACLR;
    // Enable Channel 0 interrupt / disable flag
    TA0CCTL0 |= CCIE;
    TA0CCTL0 &= ~CCIFG;
    // Set time to default 1 second for RedGReen
    TA0CCR0 = 32768 - 1;

    // Lowest power mode which enables ACLK
    _low_power_mode_3();

    for (;;)
    {
      // Infinite Loop
    }
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void TimerISR()
{
    switch(state)
    {
```

```
63        case RedFast:
64        case RedNormal:
65        case RedSlow:
66            // Toggle the red LED
67            P1OUT ^= red;
68            break;
69        case RedGreen:
70            // Toggle both LEDs
71            P1OUT ^= red;
72            P9OUT ^= green;
73            break;
74        case GreenSlow:
75        case GreenNormal:
76        case GreenFast:
77            // Toggle the green LED
78            P9OUT ^= green;
79            break;
80    }
81}
82
83#pragma vector = PORT1_VECTOR
84__interrupt void ButtonISR()
85{
86    // 5 ms debounce delay
87    _delay_cycles(5000);
88    // Check for both the interrupt flag and button
89    // press to prevent bouncing
90    int but1press = (P1IFG & but1) != 0 && (P1IN & but1) == 0;
91    int but2press = (P1IFG & but2) != 0 && (P1IN & but2) == 0;
92
93    if (but1press || but2press)
94    {
95        // Turn off LEDs for state change
96        P1OUT &= ~red;
97        P9OUT &= ~green;
98        // Move state to the next left state
99        if (but1press && state != RedFast)
100       {
101           state--;
102       }
103       else if (but2press && state != GreenFast)
104       {
105           state++;
106       }
107       // Set time based on updated state
108       switch(state)
109       {
110         case RedFast:
111         case GreenFast:
112             TA0CCR0 = (32768 / 8) - 1; // 1/8 s
113             break;
114         case RedNormal:
115         case GreenNormal:
116             TA0CCR0 = (32768 / 4) - 1; // 1/4 s
117             break;
118         case RedSlow:
119         case GreenSlow:
```

```
120                 TA0CCR0 = (32768 / 2) - 1; // 1/2 s
121                 break;
122             default:
123                 TA0CCR0 = (32768) - 1; // 1s, default for RedGreen
124                 break;
125         }
126         // Reset timer
127         TA0R = 0;
128         TA0CTL &= ~TAIFG;
129     }
130     // clear interrupt flags
131     P1IFG &= ~(but1 | but2);
132 }
```

Student Q&A

1

Using low power mode makes it so that the CPU can suspend its operations and stop drawing power when compared to never using low power mode, of which means the CPU is always on and polling devices/pins. This, therefore, makes the entire board and application more power efficient whenever power draw is a conceern. In between each interrupt the CPU will simply just be waiting for an interrupt flag to occur, do the task the interrupt flag wished to happen, then clear the flag.

2

It will be found in the *¡MSP430FR6989.h¿* header file. In our case, we'd want the vector name to be ADC12_VECTOR.

3

The programmer is responsible for clearing the interrupt flags. The CPU only does what the programmer tells it, it does not assume any instructions.

4

Again, the programmer is the one responsible for clearing the interrupt flag. It's always the programmer's fault.

5

We can remame the ISR function to anything we want. The only thing that must be constant is the *#pragma* vector declaration followed by the __interrupt return type function below it.

6

If the ISR was supposed to cleanup the flag and did not, then the ISR will not leave from its state, making the CPU most likely, never returning back to the Low Power Mode/State, assuming it was in one to begin with. As well as that quite glaring problem, any other expected functionality of the ISR most likely wil not trigger as any further interrupts will not be checked/raised, thereby causing an indefinate stutter.