Lab 2 - Interrupts and Timing

Matt Douglas, Priyanka Makin, Zach Passarelli, and Savio Tran

September 26, 2016

ECEN 2020

This lab will teach us how to properly configure and trigger interrupts on our MSP432. An interrupt is designed to 'trigger' under set conditions which calls the processor to perform an operation specific to the interrupt. In this lab we will configure our MSP432 to trigger interrupts using the buttons and timers, and we will configure LEDs and pin outputs to observe the interrupts being handled.

Button Interrupts

We start with copying the code given to us in the lab instructions into Code Composer. We then write our own code to configure the pins we will be using, specifically, two buttons set for input and triggering interrupts. In order for interrupts to work properly, we made changes to our interrupt function vector table to declare and externalize our interrupt service routines (ISR).

Before delving further into button interrupts, we need to address the issue of button debouncing. Buttons have a mechanical limitation that creates an unstable input signal that our processor may interpret wrong, so we must write code to accommodate for this. Using an oscilloscope on a button contact pin, we were able to capture the button debounce as seen in Figure 1. The button took 60 microseconds to completely change states from high to low so we determined our delay count to be 32 *for* loop iterations, calculated with data from Lab 1. This delay will be used later in our interrupt handler function.

Figure 1 - Button Debounce

The first procedure is to configure an LED to toggle by a button press. In detail, we must configure a button to trigger an interrupt which will be recognized by an interrupt handler. The interrupt handler will then change the state of the LED. The interrupt handler will also clear an interrupt flag to signify the interrupt operation is completed. As mentioned before, we need to accommodate for button debounce so that a single button press will not trigger multiple interrupts, so we implement a *for* loop delay before clearing the flag to suppress erroneous interrupts.

The LED toggle procedure allows us to see how interrupts can be handled, but we can also investigate how quickly they can be handled. We configure our main function to manually set an interrupt flag, while the interrupt handler clears the flag, causing an infinite loop of interrupts. We also toggle the output of pin 1.6, and capture its output as seen in Figure 2.

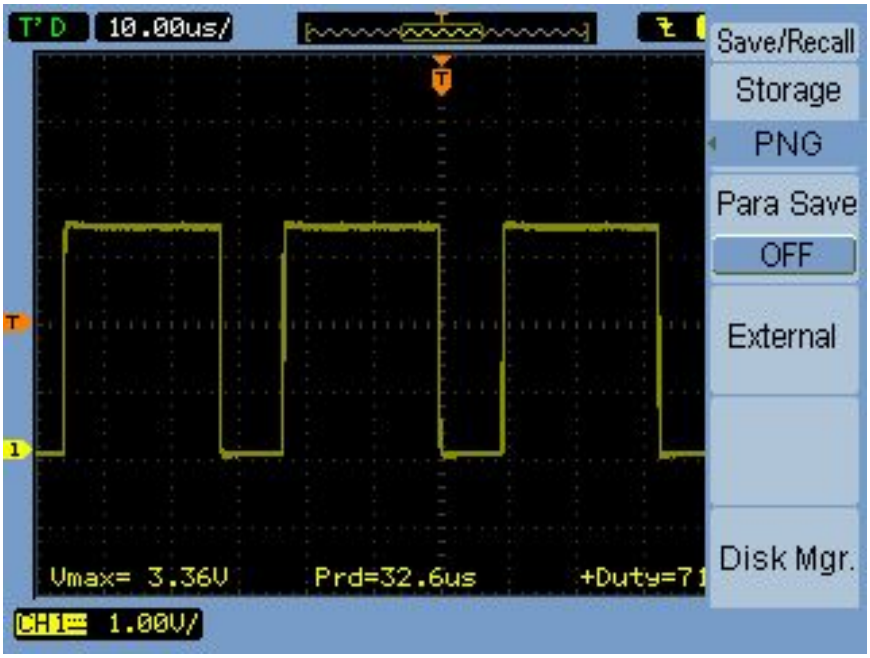


Figure 2 - Interrupt Latency

The main function sets the output of pin 1.6 high, and the handler sets the pin low. Figure 2 reveals that the pin remains high for about 23 microseconds before being changed by the handler, meaning the latency at which interrupts are handled is roughly this time period.

Multiple Button Interrupts

After becoming familiar with the concept of interrupts, we took it a step further by allowing the buttons to toggle states of pins 2.0, 2.1, and 2.2, otherwise known as the MSP432's RGB LED. We configure the buttons to trigger an interrupt and configure the interrupt handler to cycle

through the 8 color states of the RGB LED by changing the last three bits of port 2 output. The left button, pin 1.1, incremented the LEDs' states, and the right button, pin 1.4, decremented the states. Within the interrupt handler the last three bits of port 2 output were stored and ran through an *if* statement. With a left button interrupt, the port 2 output register is incremented by one which will activate specific LEDs and change the color. If bits 0, 1, and 2 in P2OUT all hold the value 1, also known as the LED's final state, then the bits are set to 0b000 to reset the LED to its first state. This *if* statement causes the LED to cycle through all the possible colors every time the button is pressed. The right button interrupt essentially does the opposite of the above operations, cycling through all color states in reverse order.

Timer Interrupts

Next, we explore interrupts that can be triggered by a timer. In this lab our timer increments by the system's clock, known as SMCLK. We also configure the TA0CTL timer register to count up. Then, we set up the Timer Capture Compare register, known as TACCR0, that will trigger an interrupt when the timer counter, TA0R, matches the value set in the TACCR0 register. To start testing, we configure a timer interrupt handler function to toggle an LED. We try out different values for the capture compare value: 1000, 20000, 40000, and 65000. With values of 1000 and 20000 the LED appears always on. We conclude that the LED state is flipping so fast that it is unnoticeable to the human eye. With a TACCR0 value of 40000, the LED seems to be barely flickering, and with a value of 65000 it is apparent that the LED is blinking very fast.

Using readings gathered from the previous procedure, we compare the period of the blinking LED with the known TACCR0 value and determine the SMCLK frequency to be approximately 1.5 kHz. Next, we experiment with the Clock Divider feature of the TA0CTL register, configuring and testing the four different prescaler states. The data is reported in Table 1.

TA0CTL Register: Clock Divider Bits	Clock Division	Frequency (Hz)	Period (ms)
0b00	/1	75.5	13.24
0b01	/2	37.75	26.48
0b10	/4	18.87	52.97
0b11	/8	9.439	105.94

Table 1 - Clock Divider

The divisions are working as intended, as each frequency is cut in half as the divider increases. We can use this data to generate a formula to relate all these variables,

$$f(prescaler, TA0CCR0) = \frac{(prescaler)(SMCLK)(1000)}{TA0CCR0+1} = frequency\ (Hz)$$

We can use the formula to calculate a frequency at which we want timer interrupts to occur. In this procedure, if we wish to blink an LED at a frequency of 5 Hz using timer interrupts, we will set TA0CCR0 to 37750 with a prescaler of 1/8. We capture the output of the LED, seen in Figure 3, and the oscilloscope's frequency measurement, 5 Hz, confirms our calculations.

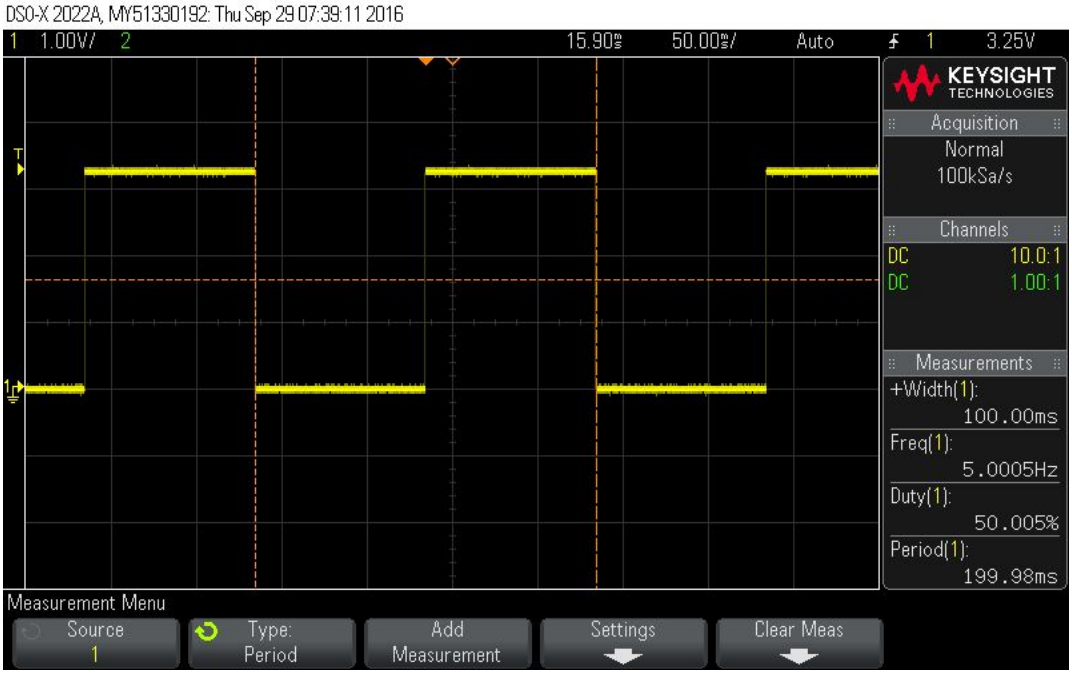


Figure 3 - 5 Hz frequency LED output

Button and Timer Interrupts

Now that we can choose the frequency at which timer interrupts are triggered, we try a more complex procedure using both types of interrupts. We wish to have a button interrupt activate timer interrupts, which in turn will change the state of the RGB LED every half second. We first configure the direction pins for the RGB LED and disable timer interrupts. Then, the button interrupt handler enables timer interrupts, meaning whenever the button is pressed, timer interrupts become active. In addition, the output pins of the LEDs are set to 0b001 so that the cycle begins in the first color state. When the timer interrupt handler runs the button interrupts are disabled so that extra button presses will not interrupt the color cycle operation which is now in the hands of timer interrupts. Unfortunately, the A0 timer cannot actually achieve the 500 ms

period due to register size limitations, so instead we have the interrupts trigger at a 250 ms period. Using our formula, we use a prescaler of ⅛ and a TA0CCR0 value of 47188. Then, each time the interrupt triggers, it increments a static variable. Whenever this static variable is even, the LED color cycle code runs, which operates in a manner similar to the first RGB LED procedure. This static variable method allows us the bypass the size limitations of our timer and achieve the color cycle period of 500 ms. Once the cycle is over and the light returns to the off state, the button interrupts are enabled again while timer interrupts are disabled.

To expand on the register size limitations issue, TA0CCR0 is an unsigned 16-bit register, which means it can hold a value of up to 65,535. If we want our timer interrupts to trigger at a 500 ms period, we would need to set TA0CCR0 to roughly 97,000, a value far beyond the limit. Possible solutions to this are using the 32-bit timer register or, in our case, cutting the period in half and using a static variable to do operations after two interrupts.

Simulate an Encoder

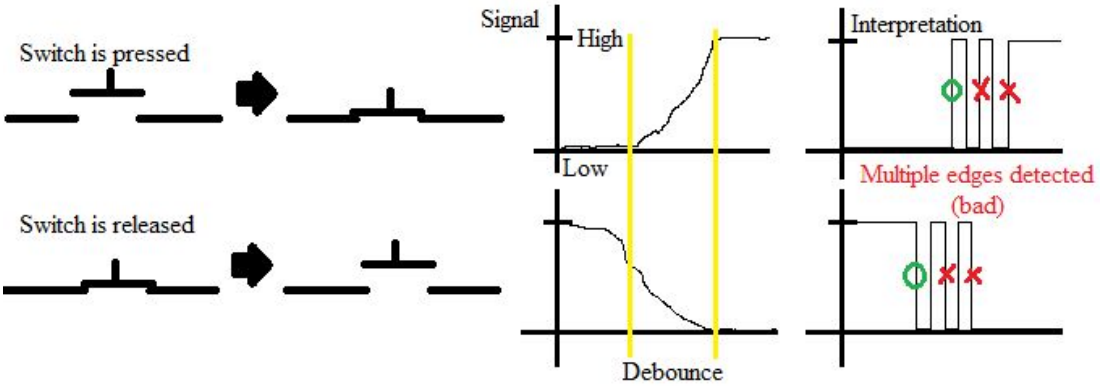
While timers and buttons are clear ways to trigger interrupts, there are also many other possible methods. In the last procedure, we configure a normal pin to trigger an interrupt when it detects a falling edge signal. We supply the signal using a function generator that generates a square wave at a frequency of 100 Hz. The interrupt handler then increments a counter whenever the pin triggers an interrupt, and we can track the number of interrupts that happen in Code Composer's console. We run the program for about two minutes and find that 851 interrupts occur from the 100 Hz signal.

With this setup in practice, we then construct a circuit consisting of an IR LED directed at a phototransistor, the output of this circuit connected to the interrupt enabled pin. By physically breaking the beam of the IR LED to the phototransistor the circuit becomes low and the pin triggers an interrupt.

Lab Report Questions

1. The MSP432 supports 64 interrupts, as stated in the documentation.
2. The interrupt function vector table needs an extern declaration of the interrupt because the interrupt service routine we are registering in this file is in main.c.
3. The ISER function for setting interrupts has a ">>5" in it to shift the bits within the register and enable interrupts for the port and timer we use in this lab.
4. The Vector Interrupt IRQ# for Port 1 is 35. Because the two buttons share this port, the code must check the P1IFG register for both bits 1 and 4 whenever a port 1 interrupt occurs.
5. The Vector Interrupt IRQ# for Timer A0 is 8.

6. Switch debounce occurs when a button is pressed and the electrical contacts take time to stabilize. This causes a non-digital signal that can be interpreted as multiple rising and falling edges that could be mistaken as several interrupts for one press. To fix this, a delay must be added to the interrupt port handlers so that after the first edge triggers the interrupt the remaining edges are ignored. Alternatively, a capacitor can be added to the hardware to help smooth the debounce.



Real World Applications

Interrupts allow embedded systems to interact with the outside world. Embedded systems can receive an interrupt trigger from a button, and then perform an action accordingly. This is a vital function in a lot of electronics, such as mp3 players. They need to be able to take commands from a user at any time, and interrupts allow the device to perform the user's desired action. Timer interrupts can also serve their own niche functions, such as when the system only needs to perform an action at specific intervals. For example, a timer interrupt can be used for an embedded system that only needs to take data from peripherals every so often. The timer interrupt can make the device take the necessary data, then shut down to a lower power usage until it needs to take data again. Thus, the embedded system will become significantly more efficient. In addition, timer interrupts can be used as delays for actions in a way similar to how we used the timer interrupt to change the LED color after pressing the button. It almost goes without saying that interrupts have near countless applications in everyday life.

Conclusion

This lab was another very fundamental step in learning embedded systems. Interrupts are an important feature to any embedded system, and are shown to be incredibly versatile in their uses. While the initialization process of interrupts seemed very complicated, once everything started working as intended it was very easy to write code in the interrupt handler functions to perform desired operations. It was also very satisfying to see a button press perform the output we wanted, meaning the interrupt was successfully triggered and handled properly. With the knowledge of configuring registers and interrupts in hand, there already seems to be a wide range of possibilities with the MSP432.

Appendix - main.c

```
1
2 #include "msp.h"
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdint.h>
6
7 #define DELAY_COUNT (32) //Delay used for button debounce
8 #define Proc2 //Button interrupt to LED
9 #define Proc3 //Interrupt latency test
10 #define Proc4 //Button interrupts to 8 stage LED
11 #define Proc5 //Timer Counter Compare interrupt test
12 #define Proc9 //5Hz LED test
13 #define Proc10 //Button interrupt to timer interrupt to LED
14 #define Proc11 //Encoder test
15
16 void configure_pins();
17 void Port1Handler();
18 void configure_timer_interrupts();
19 void TimerA0_Handler();
20
21 void main(void)
22 {
23     WDTCTL = WDTPW | WDTHOLD; //Stop watchdog timer
24
25     configure_pins(); //Call configure pins function
26     configure_timer_interrupts(); //Call configure timer interrupts
27
28     __enable_interrupt(); //Global interrupt enable
29     NVIC_EnableIRQ(PORT1_IRQn); //NVIC Port 1 interrupt enable
30
31     while(1){ //Infinite loop for interrupt testing
32 #ifdef Proc3
33         P1IFG |= BIT1; //Manually trigger pin1.1 interrupt flag
34 #endif
35     }
36 }
37
38 void configure_pins(){
39     //Configures buttons for interrupts
40     //Note: pin1.1 is left button, pin 1.4 is right button
41     //Note: pin1.7 is activated for interrupts during Encoder test
42     P1DIR &= ~(BIT1 | BIT4 | BIT7); //Input direction [pin1.1, 1.4, 1.7]
43     P1OUT |= BIT1 | BIT4 | BIT7; //Pullup resistor input [pin1.1, 1.4, 1.7]
44     P1REN |= BIT1 | BIT4 | BIT7; //Enable resistor input [pin1.1, 1.4, 1.7]
45     P1IFG &= ~(BIT1 | BIT4 | BIT7); //Clear interrupt flags [pin1.1, 1.4, 1.7]
46     P1IES |= BIT1 | BIT4 | BIT7; //Falling edge interrupt trigger [pin1.1, 1.4, 1.7]
47     P1IE |= BIT1 | BIT4 | BIT7; //Enable interrupts [pin1.1, 1.4, 1.7]
48
49 #ifdef Proc2
50     //Configures LED which will be toggled by button interrupt
51     //Note: pin1.0 is a red LED
52     P1DIR |= BIT0; //Output direction [pin1.0]
53     P1OUT &= ~BIT0; //Output low [pin1.0]
54 #endif
55 #ifdef Proc3
56     //Configures pin which will be toggled by button interrupt
57     P1DIR |= BIT6; //Output direction [pin1.6]
```

```
58      P1OUT |= BIT6; //Output high [pin1.6]
59  #endif
60  #ifdef Proc4
61      //Configures RGB LED which will be controlled by button interrupts
62      //Note: pin2.0, 2.1, 2.2 are the RGB LED
63      P2DIR |= BIT0 | BIT1 | BIT2; //Output direction [pin2.0, 2.1, 2.2]
64  #endif
65  #ifdef Proc5
66      //Configures LED which will be toggled by timer interrupt
67      P1DIR |= BIT0; //Output direction [pin1.0]
68  #endif
69  #ifdef Proc9
70      //Configures LED and output pin which will be toggled by timer interrupt
71      P1DIR |= BIT0 | BIT6; //Output direction [pin1.0, 1.6]
72  #endif
73  #ifdef Proc10
74      //Configures RGB LED which will be controlled by timer interrupts
75      P2DIR |= BIT0 | BIT1 | BIT2; //Output direction [pin2.0, 2.1, 2.2]
76  #endif
77  }
78
79  void Port1Handler(void){
80      volatile uint32_t i;
81      if(P1IFG & BIT1){ //Interrupt handler for left button [pin1.1]
82
83  #ifdef Proc2
84          //Turns LED on/off on button interrupt
85          P1OUT ^= BIT0; //Toggle output [pin1.0]
86  #endif
87  #ifdef Proc3
88          //Turns pin1.6 output high/low on button interrupt
89          //Note: For this procedure the interrupt is triggered in main
90          P1OUT ^= BIT6; //Toggle output [pin1.6]
91  #endif
92  #ifdef Proc4
93          //Changes state of RGB LED on button interrupt (left button advance)
94          uint8_t temp = P2OUT; //Store value of P2OUT in temporary variable
95          temp &= 0b00000111; //Clear all bits unrelated to RGB LED
96
97          if (temp < 7){
98              P2OUT++; //Advance RGB LED to next state
99          }
100         else if (temp == 7){
101             P2OUT ^= 0x07; //Rollover RGB LED to first state
102         }
103 #endif
104 #ifdef Proc10
105         //Enable timer interrupts, procedure continued in timer interrupt handler
106         TA0CCTL0 |= BIT4; //Capture compare interrupt enabled
107         P2OUT |= BIT0; //Activate RGB LED first state
108 #endif
109         for(i=0;i<DELAY_COUNT;i++); //Delay for button debounce
110         P1IFG &= ~BIT1; //Clear left button interrupt flag
111     }
112     else if(P1IFG & BIT4){ //Interrupt handler for right button [pin1.4]
113
114 #ifdef Proc4
```

```
115         //Changes state of RGB LED on button interrupt (right button reverse)
116         uint8_t temp = P2OUT; //Store value of P2OUT in temporary variable
117         temp &= 0b00000111; //Clear all bits unrelated to RGB LED
118
119         if (temp > 0){
120             P2OUT--; //Decrease RGB LED to previous state
121         }
122         else if (temp == 0){
123             P2OUT = 0x07; //Rollover RGB LED to final state
124         }
125 #endif
126
127         for(i=0;i<DELAY_COUNT;i++); //Delay for button debounce
128         P1IFG &= ~BIT4; //Clear right button interrupt flag
129     }
130     else if(P1IFG & BIT7){ //Interrupt handler for pin1.7
131
132 #ifdef Proc11
133         //Square signal applied to pin1.7 by Waveform Generator triggers interrupts
134
135         static uint32_t irtcount = 0; //Create static counter variable
136         irtcount++; //Increment counter when interrupt is triggered
137         printf("%d\n",irtcount); //Print value of interrupt counter
138 #endif
139         P1IFG &= ~BIT7; //Clear pin1.7 interrupt flag
140     }
141 }
142 void configure_timer_interrupts(){
143     NVIC_EnableIRQ(TA0_0_IRQn); //NVIC Timer0 interrupt enable
144     TA0R = 0; //Reset timer counter
145     TA0CTL |= BIT9 | BIT4; //Timer control: SMCLK source, Mode-Control Up mode
146     TA0CCTL0 |= BIT4; //Capture compare interrupt enabled
147
148 #ifdef Proc5
149     //Test timer interrupts for compare values of 1000, 20000, 40000, 65000
150     TA0CCR0 = 20000; //Compare value (triggers interrupt when timer matches value)
151 #endif
152 #ifdef Proc9
153     //Configure capture compare interrupts to trigger at 5 Hz frequency
154     TA0CCR0 = 37750; //Compare value set for 5 Hz
155     TA0CTL |= BIT6 | BIT7; //Timer control: 1/8 Clock Divider
156 #endif
157 #ifdef Proc10
158     //Configure capture compare interrupts to trigger every 250ms
159     TA0CCR0 = 47188; //Compare value set for 250ms
160     TA0CTL |= BIT6 | BIT7; //Timer control: 1/8 Clock Divider
161 #endif
162 }
163
164 void TimerA0_Handler(void) {
165     if(TA0CCTL0 & CCIFG){ //Handler for Capture compare interrupt
166         TA0CCTL0 &= ~CCIFG; //Clear capture compare interrupt flag
167
168 #ifdef Proc5
169         //Turns LED on/off on timer interrupt
170         P1OUT ^= BIT0; //Toggle output [pin1.0]
171 #endif
```

```
172 #ifdef Proc9
173         //Turns LED on/off and pin1.6 high/low on timer interrupt
174         P1OUT ^= BIT0 | BIT6;
175 #endif
176 #ifdef Proc10
177         //Button interrupt activates this timer interrupt
178         //Button interrupts are disabled
179         //Timer interrupts occur every 250ms
180         //Every 500ms the RGB LED will advance to next state
181         //After final RGB LED state, timer interrupts are disabled
182         //Button interrupts are re-enabled
183
184         P1IE &= ~BIT1; //Temporarily disable port 1 interrupts (disable buttons)
185
186         static uint8_t intcount = 0; //Create static interrupt counter variable
187         intcount++; //Increment interrupt counter on timer interrupt (every 250ms)
188
189         if(intcount % 2){ //True every 2 timer interrupts (500ms)
190             uint8_t temp = P2OUT; //Store value of P2OUT in temporary variable
191             temp &= 0b00000111; //Clear bits unrelated to RGB LED
192
193             if (temp < 7){
194                 P2OUT++; //Advance RGB LED to next state
195                 TA0CCTL0 &= ~CCIFG; //Clear timer interrupt flag
196             }
197             else if (temp == 7){
198                 P2OUT ^= 0x07; //Rollover RGB LED to first state
199                 TA0CCTL0 &= ~CCIFG; //Clear timer interrupt flag
200                 TA0CCTL0 &= ~BIT4; //Disable timer interrupts
201                 P1IE |= BIT1; //Re-enable port 1 interrupts (re-enable buttons)
202             }
203         }
204 #endif
205     }
206 }
207
208
209
```