Lab 4

Interrupts & Low-Power Modes

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.

4.1 Timer's Continuous Mode with Interrupt

In this part, we will program Timer_A in the continuous mode so that it raises periodic interrupts. When we program interrupts, it's not necessary to poll the timer's flag continuously. Instead, when the timer's flag is raised, an interrupt event occurs and the hardware launches a special function that responds to the interrupt.

Interrupt Basics

Let's start by reviewing the basics of interrupts in the MSP430 architecture.

Global Interrupt Enable (GIE): The GIE bit is the master on/off switch for all the interrupts in the chip. It is located in the Status Register (SR), which is register R2. In the C code, the GIE bit is set or cleared using the intrinsic functions: `_enable_interrupts()` and `_disable_interrupts()`.

Interrupt Enable bits (xIE): The microcontroller has multiple events that can be enabled to raise interrupts individually with their corresponding xIE bit. For example, the timer's rollback-to-zero event is configured to raise an interrupt by setting TAIE to 1. Port 1 has eight pins that can be configured to raise interrupts individually with the 8-bit variable P1IE. We note that for an interrupt to be enabled, the Global Interrupt Enable (GIE) bit must be 1 and the event's own interrupt enable (xIE) bit must be 1 as well.

Interrupt Flag bits (xIFG): Each event that can raise an interrupt has an interrupt flag (xIFG) associated with it. For example, when the timer rolls back to zero, the flag TAIFG is set to 1, which then raises an interrupt if TAIE=1. For Port 1, the 8-bit variable P1IFG contains eight flags that are set to 1 when the corresponding pin experiences an edge (can be configured to rising edge or falling edge).

We note that, if the interrupt event occurs and its enable bit, xIE, is zero, the flag will raise but no interrupt event occurs. This is what we did in an earlier lab: when the timer rolled back to zero, the flag TAIFG became 1 and we detected this by polling the flag. There was no interrupt since TAIE was zero.

Interrupt Service Routine (ISR): When an interrupt event occurs, the hardware finds and launches a special function, the ISR, that responds to the interrupt. Usually each interrupt event has a corresponding ISR function. However, multiple interrupt events sometimes share the same ISR function. Here is a relevant question: how does the hardware find the ISR that is associated with an interrupt event?

Vector Table: A vector is the start address, or pointer, to an ISR. The vector table contains the addresses of all the ISRs. The vector table's format and location are fixed and provided in the chip's data sheet. Therefore, the hardware performs a lookup in the vector table in order to find the ISRs in the memory. The programmer is responsible for linking the ISRs to vectors so the vector table is filled correctly. This is done with a simple syntax that we'll see below. The idea of vector table allows the ISRs to be scattered around the memory and the hardware can find them quickly by doing a lookup in the vector table (it works like a small phone book that provides the addresses of ISRs).

Table 4.1 summarizes information that's related to the timer's rollback-to-zero interrupt event. In the code, we start by enabling the interrupt event (TAIE=1). We then clear the flag (TAIFG=0) initially. Note that TAIE and TAIFG are two bits located in the timer's configuration register TACTL. We then write the ISR and link its start address to the vector. This event's vector is the timer's A1 vector. We then ensure that the flag is cleared each time the interrupt is processed. If the flag is not cleared, the hardware will raise repetitive interrupts infinitely. We finally enable the global interrupt bit (GIE=1); it's better to do this at the end when everything has been configured.

Table 4.1: Timer_A Rollback-to-Zero Interrupt Event

Event	**Enable Bit**	**Interrupt Flag**	**Vector**
TAR rollback-to-zero	TAIE	TAIFG	Timer's A1 vector

Code that Flashes the LEDs

Let's write a code that runs the timer in the continuous mode and configures an interrupt for the rollback-to-zero event. When the interrupt occurs the red LED is toggled. Use the ACLK clock signal and configure it to the 32 KHz crystal using the function we used in an earlier lab.

As we saw in earlier labs, the microcontroller has multiple Timer_A modules and we'll use the timer module #0, called Timer0_A. Therefore, the timer's A1 vector is called **TIMER0_A1_VECTOR** and we will link the ISR to it so the hardware can find the ISR. All the vector names can be found in the .h file (e.g. msp430fr6989.h).

In earlier labs, we used the timer's configuration variable TA0CTL which contains the fields: TASSEL (clock select), ID (clock divider), MC (mode), the interrupt flag (TAIFG) and the interrupt enable bit (TAIE). Therefore, the interrupt-related bits are in this variable.

```
// Timer_A continuous mode, with interrupts, flashes LEDs
#include <msp430fr6989.h>
#define redLED BIT0                // Red LED at P1.0
#define greenLED BIT7              // Green LED at P9.7

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

  P1DIR |= redLED;                 // Configure pin as output
  P9DIR |= greenLED;               // Configure pin as output
  P1OUT &= ~redLED;                // Turn LED Off
  P9OUT &= ~greenLED;              // Turn LED Off

  // Configure ACLK to the 32 KHz crystal
  ...

  // Configure Timer_A
  // Use ACLK, divide by 1, continuous mode, TAR cleared, enable interrupt
     for rollback-to-zero event
  TA0CTL = ...

  // Ensure the flag is cleared at the start
  TA0CTL &= ~TAIFG;

  // Enable the global interrupt bit (call an intrinsic function)
  ...

  // Infinite loop... the code waits here between interrupts
  for(;;) {}
}

//******* Writing the ISR *******
#pragma vector = TIMER0_A1_VECTOR      // Link the ISR to the vector
__interrupt void T0A1_ISR() {
  // Interrupt response goes here
  ...
}
```

Let's review the syntax of the ISR. The keyword pragma is used to add a functionality to the compiler. In this case, the start address of the ISR is linked to the vector TIMER0_A1_VECTOR. The vector name should be exactly as spelled in the .h file. The keyword __interrupt indicates to the compiler that this is an ISR. ISRs have a special return procedure in that they restore the PC and the low-power mode (regular functions only restore the PC). The function's name, T0A1_ISR, is not a keyword and can be changed to any name. It's not necessary to write a function header above the main function as this function won't be called from the software.

Perform the following and submit the answers in your report:

- Complete the code and demo it to the TA.
- Compute the period of interrupts that you should observe.
- Compare the timing to your phone's stopwatch for at least 20 seconds and ensure it matches closely (the crystal is accurate).
- What happens if we don't clear the flag each time an interrupt occurs? Explain.
- What is the CPU doing between interrupts?
- Who is calling the ISR? Is it the software? Explain.
- Submit the code in your report.

4.2 Timer's Up Mode with Interrupt

In this part, we'll run the timer in the up mode and raise periodic interrupts. Before we get in the details, let's look at Table 4.2 which summarizes multiple interrupt events of Timer_A.

The first line in the table corresponds to the rollback-to-zero interrupt event that we used in the earlier part. The following lines show that the timer has multiple channels and each one has an interrupt event called the compare event. For example, Channel 0 has a register called TACCR0. If we set this register to 200, when the counting register, TAR, passes by this value, Channel 0 raises its flag CCIFG (Capture/-Compare Interrupt Flag) and, if CCIE=1 (Capture/Compare Interrupt Enable), an interrupt is raised. Note that CCIE and CCIFG of Channel 0 are located in register TACCTL0, which is Channel 0's configuration register. Similarly, the two other channels have their own compare interrupt events.

Table 4.2: Interrupt Events of Timer_A

Event	**Bits**	**Vector**	**MSP430 Policy**
Rollback-to-zero Channel 1 (TAR=TACCR1) Channel 2 (TAR=TACCR2)	TAIE / TAIFG in TACTL CCIE / CCIFG in TACCTL1 CCIE / CCIFG in TACCTL2	A1	Programmer clears the flag (shared vector)
Channel 0 (TAR=TACCR0)	CCIE / CCIFG in TACCTL0	A0	Hardware clears the flag (non-shared vector)

The table also shows that the rollback-to-zero event and the compare events of Channel 1 and Channel 2 share the timer's A1 vector. It means that these three events are handled in the same ISR. One vector corresponds to one ISR. On the other hand, Channel 0 has its own dedicated vector, the A0 vector, and this means that it has its own ISR. In MSP430, when the vector is not shared the hardware clears the flag. That is, for Channel 0, when the ISR is launched, the hardware clears the flag CCIFG in TACCTL0. For the other events, the programmer is responsible for clearing the flag.

Let's recap on how the up mode works in order to select the interrupt event that we'll use. Below is how TAR counts in the up mode. The upperbound of the up mode is always set by Channel 0's register, TACCR0. At the end of the count, Channel 0's compare event triggers and CCIFG is raised. One cycle later, TAR rolls back to zero and TAIFG is raised. We can use either of these interrupt events. We'll use Channel 0's interrupt event.

```
TAR:     0, 1, 2, ..., TACCR0, 0, 1, 2, ..., TACCR0, 0, 1, ...
                          ↑     ↑                ↑     ↑
                        CCIFG  TAIFG           CCIFG  TAIFG
```

Our plan is the following. We'll start the timer in the up mode (setting TACCR0), clear the interrupt flag of Channel 0 (CCIFG), enable the interrupt bit of Channel 0 (CCIE) and then enable the global interrupt bit (GIE=1). Finally, we'll write the ISR and link it to the A0 vector; the hardware clears the flag CCIFG each time this function is called since this ISR is not shared with other interrupt events.

Below is the code. Start the timer in the up mode and set a delay of 1 second. Configure Channel 0's interrupt. Use the ACLK clock signal based on the 32 KHz crystal using the function from earlier labs. Remember that 32 KHz is 32,768 Hz. Complete the missing parts.

```
// 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 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 On (alternate flashing)

  // Configure ACLK to the 32 KHz crystal
  ...

  // Configure Channel 0 for up mode with interrupts
  TA0CCR0 = ...                   // 1 second @ 32 KHz
  TA0CCTL0 ...                    // Enable Channel 0 CCIE bit
  TA0CCTL0 ...                    // Clear Channel 0 CCIFG bit

  // Timer_A: ACLK, div by 1, up mode, clear TAR
  ...

  // Enable the global interrupt bit (call an intrinsic function)
  ...

  for(;;) {}
}

//*******************************
#pragma vector = TIMER0_A0_VECTOR
__interrupt void T0A0_ISR() {
    // Action goes here
    ...
}
```

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- Compare the timing to your phone's stopwatch for at least 20 seconds and make sure the timing matches closely (the crystal is accurate).
- Should we set TAIE to 1 in this code? Explain.
- Should the ISR clear the flag of Channel 0? Explain.
- Modify the code so that the delay is 0.5 seconds (then try 0.1 seconds).

4.3 Push Button with Interrupt

In this part, we will read the push buttons via interrupts and program the following: when button S1 is pushed, the red LED is toggled and when button S2 is pushed, the green LED is toggled.

As we have seen in earlier labs, the push buttons on our board are in the active low configuration. Therefore, we'll configure a falling edge interrupt so the interrupt occurs as soon the button is pushed. The piece of code below shows how the buttons are configured for interrupts.

```
#define BUT1 BIT1        // Button S1 at Port 1.1
#define BUT2 BIT2        // Button S2 at Port 1.2
...
// Configure the buttons for interrupts
P1DIR &= ~(BUT1|BUT2);   // 0: input
P1REN |= (BUT1|BUT2);    // 1: enable built-in resistors
P1OUT ...                // 1: built-in resistor is pulled up to Vcc
P1IES ...                // 1: interrupt on falling edge (0 for rising edge)
P1IFG ...                // 0: clear the interrupt flags
P1IE ...                 // 1: enable the interrupts
```

Since Port 1 is 8-bit and each of its bits can be configured individually, all of the variables above (P1DIR, P1IFG, P1IE...) are 8 bits. Therefore, we do AND and OR operations to configure the pins.

The code should look like the following. Complete the button configuration by following the directions in the comments. Then, enable the global interrupts bit (GIE bit) and write an empty infinite for-loop at the end of main function so the code waits there between interrupts. There's no need to use the timer in this code.

Then, write the ISR of Port 1 and link it to the correct vector. All the eight interrupt events of Port 1 share the same vector/ISR. The vector name should be something like P1_VECTOR or PORT_1_VECTOR. To find the vector's exact name, look in the file `msp430fr6989.h` and search for the word VECTOR. You can locate this file by searching for it in the folder where CCS is installed, which should like the path below. An easier way is to hover the mouse pointer over the include line in CCS then click the Control button on the keyboard and click the left button the mouse and the file will open.

```
C:\...\ccsv7\ccs_base\msp430\include_gcc\msp430fr6989.h
```

Let's take a closer look at how the ISR of Port 1 works. If any of Port 1 interrupt occurs, i.e. any bit in P1IFG goes to 1, the ISR is called. Since we enabled two interrupt events in Port 1, we should check which bit(s) in P1IFG is set and service the corresponding interrupt. This is done with the two if-statements in the code template below. Since multiple interrupt events share the ISR, it is the programmer's responsibility to clear the flags (bits in P1IFG). If the flags are not cleared, the hardware will raise repetitive interrupts infinitely.

```
#pragma vector = ...               // Write the vector name
__interrupt void Port1_ISR() {
  // Detect button 1 interrupt flag
  if ( ... ) {
     // Button 1 action
     ...
  }

  // Detect button 2 interrupt flag
  if ( ... ) {
     // Button 2 action
     ...
  }
}
```

We note that bouncing can be addressed by implementing a debouncing algorithms (that we'll see in future labs) or by using buttons with debouncing hardware built-in. A simple debouncing approach is to add a delay loop at the end of the ISR to wait out the bounces (do not use that when you report the failure rate of the code).

Perform the following and answer the questions in your report:

- Complete the code and demo it to the TA.
- 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.
- Roughly, what is the success rate of this code?
- Submit the code in your report.

4.4 Low-Power Modes

The MSP430 microcontroller supports multiple low-power modes. They work by shutting down more and more components that are not needed in order to save power. The most popular low-power modes are summarized in Table 4.3.

MCLK (Master Clock) is the clock that drives the CPU. SMCLK (SubMaster Clock) and ACLK (Auxiliary Clock) drive peripherals such as the timer. SMCLK is usually a high-frequency clock (e.g. 1 MHz) while ACLK is usually a low-frequency clock (e.g. 32 KHz). The active mode is the default mode and, in this mode, all the clocks are running. Engaging LPM0 shuts down MCLK and, therefore, the CPU is off. However, peripherals can run based on SMCLK or ACLK. LPM3 further shuts down SMCLK, but

Table 4.3: Popular Low-Power Modes in MSP430

Mode	**MCLK**	**SMCLK**	**ACLK**
Active mode	On	On	On
LPM0	x	On	On
LPM3	x	x	On
LPM4	x	x	x

peripherals can run using ACLK. Finally, engaging LPM4 disables all the clock signals.

Here is an interesting question: what happens if the programmer intends to use the timer but mistakenly engages LPM4 which shuts down all the clock signals? Our chip (MSP430FR6989) supports the overriding feature which means, if a low-power mode shuts down a clock but a peripheral requests that clock, that clock will turn on for as long as it's requested. Therefore, if the timer uses ACLK and we mistakenly engage LPM4, ACLK will turn on to run the timer. The overriding feature is enabled by default but can be disabled. Some basic MSP430 chips don't support the overriding feature.

Interrupts and low-power modes go hand-in-hand because, when a low-power mode shuts down the CPU and clock signals, the only way to reactive the CPU and the clock signals is via an interrupt. If we engage a low-power mode without configuring any interrupt event, the system will remain in low-power mode forever (or until a new software is flashed). Therefore, when we engage a low-power mode we configure at least one interrupt event.

The following is the flow of the program when a low-power mode is used. We start by doing initial configurations and configure at least one interrupt event then enter a low-power mode at the end of main. When an interrupt occurs, the hardware saves the state of the CPU, engages the active mode (CPU and all clocks reactivate), then launches the corresponding ISR. The hardware runs the ISR then restores the low-power mode that was active before the interrupt had occurred.

How does the hardware restore the low-power mode after the interrupt is processed? The low-power mode is indicated by four bits in the Status Register (SR). When an interrupt occurs, PC and SR are saved on the stack. The hardware runs the ISR, then restores PC and SR, therefore, restoring the low-power mode.

How does the programmer engage a specific low-power mode? The low-power modes are engaged via four bits in the Status Register (SR) called SCG1, SCG2, CPUOFF, OSCOFF. Instead of dealing with these bits directly, we use the intrinsic function below. For example, we call `_low_power_mode_0();` to engage LPM0. Note that this function also enables the global interrupts (GIE bit) because interrupts are always used when a low-power mode is engaged. Therefore, there's no need to call `_enable_interrupts();` when this function is used.

```
_low_power_mode_x();
```

Engaging Low-Power Modes

Revisit the three codes of this experiment and engage the appropriate low-power mode for each code. The goal is to minimize the power consumed, therefore, choose the lowest consuming power mode that keeps the code operational. Consult Table 4.3. Don't rely on the overriding feature, if a clock signal is needed, choose a low-power mode that keeps it active.

The only change needed for the earlier codes is replacing the for-loop at the end of the main() with the low-power mode intrinsic function. We can also erase the interrupt enable line since the low-power mode function automatically enables the global interrupt bit. By doing these changes, instead of having the CPU cycle in the infinite for-loop between interrupts, the CPU now is shut down while we're waiting for the interrupt events.

Perform the following and answer the questions in your report:

- Complete the codes and demo them to the TA.
- Which low-power mode did you choose for each of the three codes? Explain your choices.
- Test the three codes and ensure they remain operational.
- Submit only the few lines of code that you modified.

4.5 Application: Crawler Guidance System

You are employed at Kennedy Space Center and you are asked to design a guidance system that guides the massive crawlers that transport rockets to launch pads. The guidance system will be operated by a spotter who has visual awareness of the crawler's position. The spotter gives guidance to the driver to move left, right or keep going straight.

The spotter uses two push buttons to request movement to the left or right. The driver sees two LEDs, the red LED indicating a movement to the left is needed and the green LED indicating a movement to the right is needed. The LEDs also indicate the extent to which correction is needed. The LED flashing pattern is the following. Both LEDs flashing at a slow pace indicates the vehicle is spot on and no correction is needed. The red LED may flash at one of three speeds indicating how much correction is needed to the left. Faster flashing indicates more correction is needed. The green LED works similarly by supporting three flashing speeds. In total, there are seven flashing patterns that are represented below where R and G designate Red and Green, respectively. A '+' sign indicates a faster flashing speed, that is, R++ flashes faster than R+ and, similarly, R+ flashes faster than RG. The transitions occur only between adjacent states.

```
R+++     R++     R+     RG     G+     G++     G+++
```

In this code, we are required to use the timer with interrupts to create the flashing. We are also required to use the buttons with interrupt. Polling the timer's flag, polling the buttons or using a delay loop to flash the LEDs is not permitted. One problem we may encounter is button bouncing, where one button push creates two or more interrupts. To combat this, we can add a delay loop at the end of the button ISR to wait out the bounces. A button's bounce duration may last up to about 20 ms. We can use the function `_delay_cycles();` to create a delay. The cycles we request are at 1 MHz, the default CPU clock. This is the only place in the code we're permitted to use a delay loop.

A demo of the application can be found at the video below. You have the flexibility to define the details but your design should largely resemble the demo.

https://youtu.be/SLjYTANZmO0

Perform the following:

- Describe your design in the report
- Demo your program to the TA & submit your code.

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?
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?
3. A vector, therefore the ISR, is shared between multiple interrupt events. Who is responsible for clearing the interrupt flags?
4. A vector, and its corresponding ISR, is used by one interrupt event exclusively. Who 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?
6. What happens if the ISR is supposed to clear the interrupt flag and it didn't?