Lab 3 Timers and Interrupts

Introduction

Single Sentence Overview

You will learn how to deal with timer hardware and time-sensitive events called “Interrupts.”

Overview

Most embedded systems applications need to be able to report time accurately. They typically do this with built-in clocks called “timers.” The PIC16 has a built-in timer called TMR0. It is synchronized by main oscillator on your board. This high-frequency oscillator acts like the seconds hand on a wall clock, as shown in Figure 1 Timer prescaling on your microcontroller works like the ratios between second, minute and hour hands on a wall clock.

The “tick-tock” rate of the timer is set by two pre-scalers, thereby reducing its frequency with respect to the main oscillator. The first prescaling is a constant divide-by-four. The second pre-scaling is variable and is set by bits 2,1 and 0 in the PIC16’s OPTION_REG register. Using the pre-scaling you can reduce the frequency at which a timer reaches its maximum counting value, also called the “overflow.” With an internal 500 kHz oscillator and the prescaler setting in OPTION_REG set to 0b111, your reduce the counting frequency and thus the frequency of overflow can be viewed by the naked eye if you use an LED connected to the timer output. This can be used as the basis for a “heartbeat” indicator in a future embedded systems that you will want to work on.

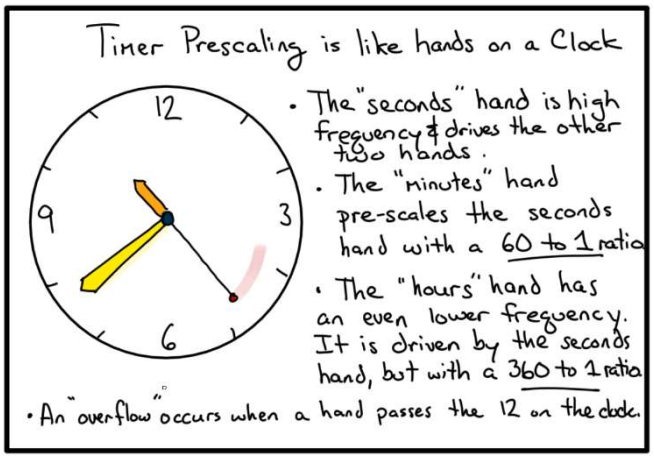

Figure 1 Timer prescaling on your microcontroller works like the ratios between second, minute and hour hands on a wall clock.

The best way to take action when the timer overflow occurs is to link your timer to an “interrupt service routine.” This is a special function that you write separately from the main function. When you configure your microcontroller to act on interruptions it will halt whatever it was doing, deal with the interruption using your interrupt service routine, and then return back to doing its original task. You’ll learn how to write a simple interrupt service routine in both C and Assembler in this lab.

Once you are done with interrupts in this lab you will also look at how disassembly tools permit you to analyze your code.

Learning Outcomes

At the end of this lab the student will be able to

- Set up a hardware timer and to compare it to the use of manual, blocking delays.
- Set up an interrupt service routine tied to the hardware timer
- Disassemble an existing microcontroller program

Success Criteria

The student will demonstrate working timer solutions through the use of flashing LEDs.

Prerequisites

Attending TD class and finishing the previous two TP labs.

Grading Rubric

- Part 1:
  - 0 if neither question is answered, or both are wrong. 1 for each correct answer.
  - 0 if the first demonstration does not work or is not attempted. 2.5 if the first demonstration partially works. 5 if the first demonstration fully works.
  - 0 if the second demonstration does not work or is not attempted. 2.5 if the second demonstration partially works. 5 if the second demonstration fully works.
- Part 2:
  - There are no questions in Part 2.
  - 0 if the demonstration does not work or is not attempted. 2.5 if the demonstration partially works. 5 if the demonstration fully works.
- Part 3:
  - 0 if neither question is answered, or both are wrong. 1 for each correct answer.
  - There is no demonstration in Part 3.

Part 1: The Timing of a Heartbeat

Single Sentence Overview

Create a visible "all is working" heartbeat on the Curiosity board that functions independently of the main function.

Background

Two of the most useful components inside the PIC16 microcontroller are the "timer" and the "interruption" mechanisms. We will effectively use "Timer0" like an alarm clock that can set off an alarm signal on a regular basis. We will let this alarm to interrupt the usual operation in the main loop of the microcontroller and to flash an LED with the same frequency as the timer alarms.

It is possible to use other timers on the PIC16, as well as other sources of data to interrupt the PIC16's functioning. However, Timer0 is one of the most straight-forward ways to learn about both timer operations and interrupt mechanisms.

One of the major configuration registers for Timer0 is `OPTION_REG`, while the major configuration register for interrupts is `INTCON`. When configuring we typically want to disable interrupts, configure the timer features and then re-enable interrupts, as shown Table 1.

In Part 1 of this lab you will contrast the creation of a "heartbeat" signal that uses a simple approximate delay versus a better "heartbeat" that combines the use of a Timer and an Interrupt Service Routine.

Timers

When you need to know time, with a watch or cell phone, you either look at the display on demand or you set an alarm to warn you at a pre-arranged time. Timer0 on the PIC16 works the same way. You can either request to know its time or you can have it set off an alarm at a regular interval. We will use Timer0's "overflow alarm" setting here. The 8-bit counter inside Timer0 counts from 0 to 255 at a specific frequency. When it reaches 255 it sets off an alarm that can make an Interrupt Service Request and then starts counting again. The frequency at which these alarms go off is set by specifying a high-frequency source clock and then setting a prescale value to reduce the frequency, as per Figure 2.

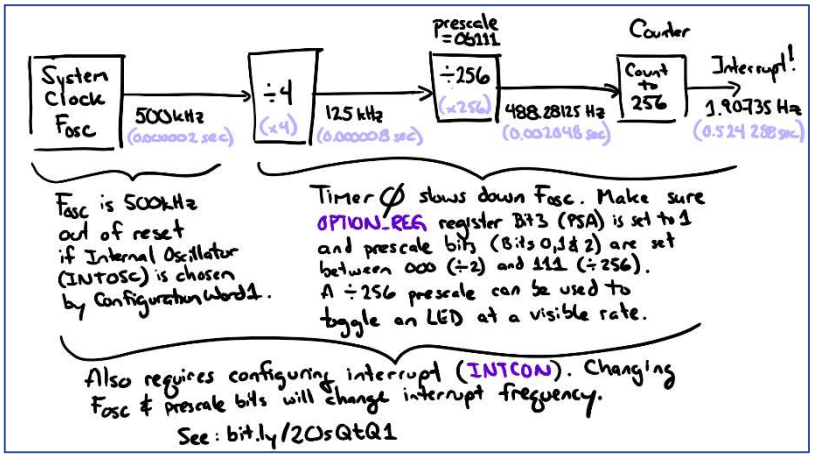

Figure 2 Timer0's interruption alarm frequency is based on a 500 kHz high frequency system clock and then two prescale values (÷4 and ÷256) that slow down the counting rate. Based on Fig. 11.1 in Section 11 of the PIC Mid-Range Micro manual. (http://ww1.microchip.com/downloads/en/DeviceDoc/31011a.pdf)

It's important to point out that the System Clock, by default out of reset, is *internal* to the chip and is set to *500 kHz*. You can adjust it to be up to 32 MHz *or* down to 32 *kHz*, depending on your application.

However, for Lab 3, 500 kHz is good as it allows us to have a Timer0 overflow frequency close to 2 Hz with the right prescale value, as shown Figure 2.

Interrupt like an Alarm Clock

It is important to be able to stop a task on the microcontroller when something more important needs to be dealt with. This is referred to as an "interrupt." The interruption can be caused by internal events (like a timer alarm) or external events (like an emergency button press in a factory). When an interrupt occurs on the PIC16, the microcontroller stops what it was doing and goes to memory location 0x04 to find instructions on how to proceed. It is your job to write the "interrupt service routine" (ISR) that this process engages. In Table 1 you can see the steps that you need to take care of in the ISR function that you write. In the case of Lab 3 you will be looking at how the Timer0 can *make a request* for an interrupt. This is similar to how an alarm clock can be set up to tell you when it's time to wake up in the morning.

Table 1 The Timer0 can be configured to generate interrupt requests each time it causes an overflow "alarm". These are the four steps to permit the PIC16 to call an interrupt service routine. The first and last steps are always done for any type of interrupt setup. The ones in the middle are dependant on the source of the interrupt, in this case the timer.

Step #	Task	Details	Reference
First Step (always)	Disable Interrupts	Clear GIE (Bit 7) in `INTCON`	https://bit.ly/2OsQtQ1 (Section 7.1 & 7.6)
Application-specific step	Timer prescale	`OPTION_REG` prescale settings in bits 0 to 5	https://bit.ly/2OsQtQ1 (Section 21.2 & 21.2)
Application-specific step	Timer interrupt enable	Set `INTCON`'s Timer Enable bit (Bit 5)	https://bit.ly/2OsQtQ1 (Section 14.2)
Last Step (always)	Enable Interrupts	Set `GIE` (Bit 7) in `INTCON`	https://bit.ly/2OsQtQ1 (Section 7.1 & 7.6)

References

For further reading you can refer to these pages:

- The TMR0 timer in Mid-Range PIC MCUs: https://bit.ly/2NyHXCB (URL verified Sept. 2018)
- Extra material on the Timer: https://bit.ly/2OR7VOA (URL verified Sept. 2018)
- Bit setting, toggling and clearing in C: https://bit.ly/2xKguDc (URL verified Sept. 2018)
- Blocking loops: https://bit.ly/2OKCtBf (URL verified Sept. 2018)
- PIC tutorial at CSAIL: https://bit.ly/2xMgXF9 (URL verified Sept. 2018)

Stimulate

1. Delays are easy to create on your PIC16. The easiest ones use loops and are referred to as "blocking." Why is a blocking loop a bad idea? Refer to https://bit.ly/2OKCtBf for discussion using Arduinos.

Explore

1. What is the built-in command in XC8 for creating a manual delay of 500 milliseconds?

Problem 1a: Make a Bad Heartbeat (with a Blocking Delay)

To appreciate a good, interrupt-based timer, you need to make its opposite: a bad "blocking" timer. You'll use this bad timer to flash an LED on and off, imitating a heartbeat on your microcontroller board.

Create a loop in your main function that turns an LED on and off in a repeatable way. An example of an infinite loop is shown in Figure 3 Basic layout of a typical C program in embedded applications.. This method doesn't use interrupts, because verifying that your I/O hardware is working helps reduce possible sources of bugs when you've engaged the interruption mechanism.

```
/* my c file */

// put #pragma and #define statements here

// put #include statements here

// Need to define an interrupt service routine?  You can do that here.

// main function
int void main(void)
{
    // need a variable?  Define it here.

    // initialize registers here

    while(1)        // beginning of while loop.
    {

        // write boring microcontroller jobs here
        // to be done from now until the end of time.


    }               // end of while loop.


    return 0;
}
```

Figure 3 Basic layout of a typical C program in embedded applications.

Make sure that you set up your configuration bits using #pragma statements placed at the top of your C source file (above the main function). You can either do it from scratch using the

```
Window -> Target Memory View -> Configuration Bits
```

or copy-and-paste from the following:

```
// PIC16F1619 Configuration Bit Settings
// 'C' source line config statements

// CONFIG1
#pragma config FOSC = INTOSC    // Oscillator Selection Bits (INTOSC oscillator: I/O function on CLKIN pin)
#pragma config PWRTE = OFF      // Power-up Timer Enable (PWRT disabled)
#pragma config MCLRE = ON       // MCLR Pin Function Select (MCLR/VPP pin function is MCLR)
#pragma config CP = OFF         // Flash Program Memory Code Protection (Program memory code protection is disabled)
#pragma config BOREN = ON       // Brown-out Reset Enable (Brown-out Reset enabled)
#pragma config CLKOUTEN = OFF   // Clock Out Enable (CLKOUT function is disabled. I/O or oscillator function on the CLKOUT pin)
#pragma config IESO = ON        // Internal/External Switch Over (Internal External Switch Over mode is enabled)
#pragma config FCMEN = ON       // Fail-Safe Clock Monitor Enable (Fail-Safe Clock Monitor is enabled)

// CONFIG2
#pragma config WRT = OFF        // Flash Memory Self-Write Protection (Write protection off)
#pragma config PPS1WAY = ON     // Peripheral Pin Select one-way control (The PPSLOCK bit cannot be cleared once it is set by software)
#pragma config ZCD = OFF        // Zero Cross Detect Disable Bit (ZCD disable.  ZCD can be enabled by setting the ZCDSEN bit of ZCDCON)
#pragma config PLLEN = ON       // PLL Enable Bit (4x PLL is always enabled)
#pragma config STVREN = ON      // Stack Overflow/Underflow Reset Enable (Stack Overflow or Underflow will cause a Reset)
#pragma config BORV = LO        // Brown-out Reset Voltage Selection (Brown-out Reset Voltage (Vbor), low trip point selected.)
#pragma config LPBOR = OFF      // Low-Power Brown Out Reset (Low-Power BOR is disabled)
#pragma config LVP = ON         // Low-Voltage Programming Enable (Low-voltage programming enabled)

// CONFIG3
#pragma config WDTCPS = WDTCPS1F// WDT Period Select (Software Control (WDTPS))
#pragma config WDTE = OFF       // Watchdog Timer Enable (WDT disabled)
#pragma config WDTCWS = WDTCWSSW// WDT Window Select (Software WDT window size control (WDTWS bits))
#pragma config WDTCCS = SWC     // WDT Input Clock Selector (Software control, controlled by WDTCS bits)

#define _XTAL_FREQ 500000     // Define PIC16F1619 clock freq - 500 kHz is default with internal clock.
```

The Configuration Bits method doesn't specify the crystal frequency. So make sure to include the last line (`#define _XTAL_FREQ 500000`) otherwise the delay function you wish to call won't work.

Then, do the following:

1. In your main function, create an infinite while loop.
2. Before the while loop, turn off interrupts *globally* by making Bit 7 (GIE) of `INTCON` to 0.
3. Like in Lab 1, initialise `PORTC`, `ANSELC` and `TRISC` registers to make the Curiosity board's LED D7 (PIC's RC5) available to flash on and off.
4. ~~*Modify the PSA, PS2, PS1 and PS0 to use the `TMR0` timer to use the prescaler of your choice.*~~
5. ~~*Modify the `INTCON` register to enable Global and Timer0 interrupts*~~

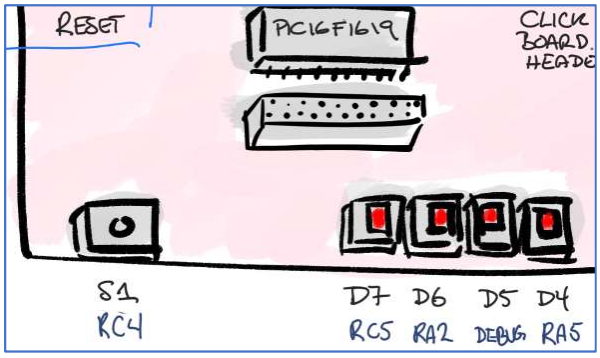

Figure 4 For your heartbeat, make LED D7 (connected to the PIC's RC5) blink on and off.

(Don't do the last two steps yet!)

Next, inside the infinite while loop, write four lines, in sequence:

1. Turn on LED D7 (PIC's RC5) by assigning a binary value to LATC.
    - You can either assign values to all the bits in Port B at the same time, or
    - You can use a bit mask and logic function to "set a bit" (https://bit.ly/2xKguDc).
2. Add a 500 millisecond delay
3. Turn off the LED from Step 1.
    - You can either "clear a bit" or "toggle a bit" here (https://bit.ly/2xKguDc)
4. Add a second 500 millisecond delay.

Demonstrate to the lab instructor that you've got the heartbeat working.

Problem 1b: Make a Good Heartbeat (with Timer0 & an Interrupt)

We all have busy days and lose track of time. Your main function can also get super busy and if you rely on it to track time using delay functions you'll find that it will eventually fail. The day before a big exam it's a good idea to set an alarm on your cell phone or on your *Réveil* so that you wake up on time. It will interrupt your sleep to make sure you get to your exam.

Like your *Réveil* the PIC16 contains an alarm system called Timer 0 (`TMR0`). It reads a frequency based on crystal or main oscillator and then slows it down in three stages, as per the diagram in Figure 2 on page 3.

The Timer acts like a clock. When you enable interrupts (set the `TMR0IE` bit in `INTCON`) it becomes an *alarm* clock. The alarm goes off each time the clock finishes counting to 255 from 0. The main source of timing comes from the `FOSC` oscillator. This is initially divided by four to create a slower frequency. If the `PSA` bit in the `OPTION_REG` is set to zero then it is slowed down even more via the three bits of the prescaler – up to 256 times slower if you `PS2`, `PS1` and `PS0` to 1 each! The equation for determining frequency, in Hz, of the interrupt-based alarm is:

$$f_{alarm} = \frac{f_{osc}}{4} \cdot \frac{1}{Prescale} \cdot \frac{1}{(2^8 - 1)}$$

The alarm counts 256 times each cycle, from 0 to 255. Each "tick" in the timer has a period T_{tick}, in seconds, defined by:

$$T_{tick} = \frac{1}{\frac{f_{osc}}{4} \cdot \frac{1}{Prescale} \cdot \frac{1}{(2^8 - 1)}}$$

Table 2 Prescale bits slow down the Timer

Prescale bits (PS2,PS1,PS0)	*Prescale*	*Effect on Timer*
000	2	*A bit slower*
001	4	
010	8	
011	16	
100	32	
101	64	
110	128	
111	256	*Much slower*

By setting up the `OPTION_REG` register you can automatically have the timer operate at the frequency you desire, using the equations above. By setting up the `INTCON` register to create an alarm every time the timer counts to 255 you can have your microcontroller keep track of time very well. Similar to Part 1a, here's how you do it:

1. In your main function, create an infinite while loop.
2. Before the while loop, turn off interrupts *globally* by making Bit 7 (`GIE`) of `INTCON` to 0.
3. Like in Lab 1, initialise `PORTC`, `ANSELC` and `TRISC` registers to make the Curiosity board's LED D7 (PIC's `RC5`) available to flash on and off.
4. *Modify the `PSA`, `PS2`, `PS1` and `PS0` to use the `TMR0` timer to use the prescaler of your choice.*
5. *Modify the INTCON register to enable Global and Timer0 interrupts*

Next you need to set up a special function called an "Interrupt Service Routine." It's like your main function except it needs to be define in a very special way, as shown by the following figure.

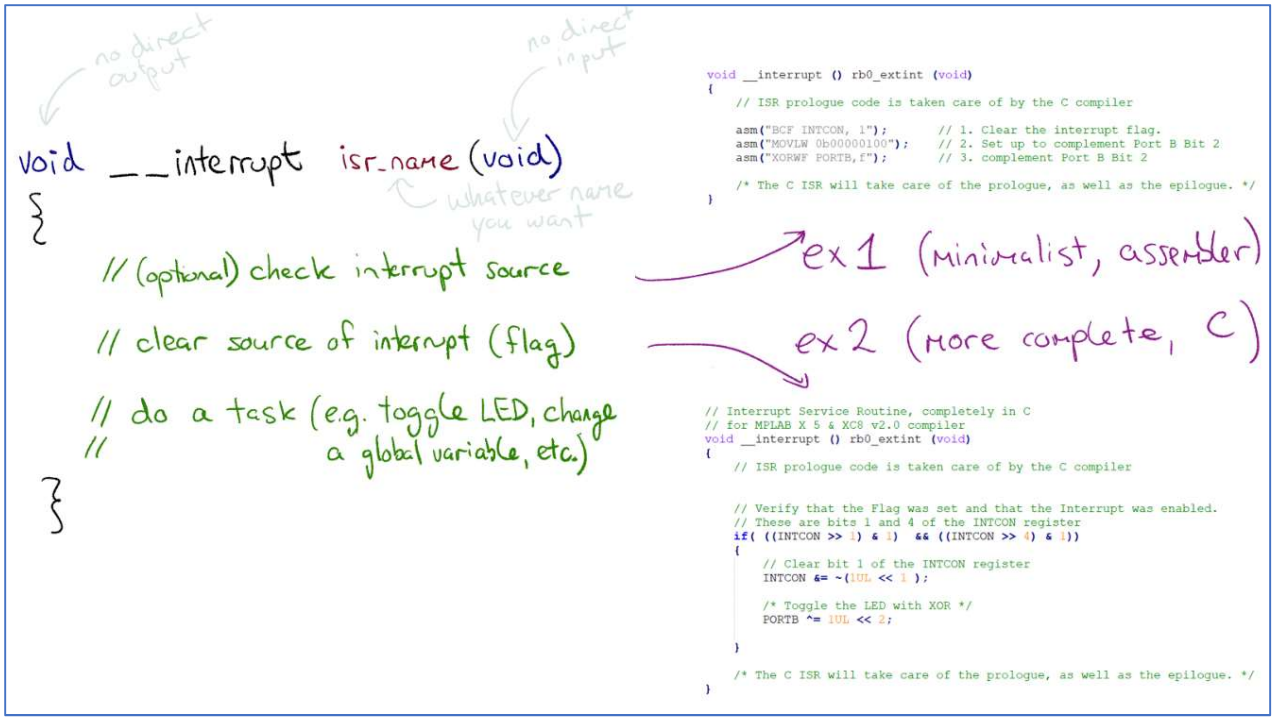

Figure 5 Structure of an Interrupt Service Routine (ISR). This is compatible with the latest MPLAB X compiler, XC8 v2.0 (mid-2018). The assembler example is minimalist and doesn't include interrupt enable and flag bits (INTCON's INT0IE and INT0IF bits) that are included in the C example.

You can name the function anything you want, as shown by the text in blue. However, the function name needs to be preceded by `__interrupt()` in order to tell the compiler that this is an Interrupt Service Routine. You are also not permitted to pass it any information directly or to return anything from it, thus use of the two voids.

Inside your Interrupt Service Routine your ISR, follow the following steps:

1. Check the source of the interrupt. Do this by checking the status of specific bits in the `INTCON` register. For Timer0, check the `TMR0IF` bit (The "F" stands for "Flag"). If `TMR0IF` is "1" then the interrupt signal from the `TMR0` "alarm" has been raised.
    a. Look at Table 3 for a reminder of how to **check** the status of a bit in a register.
2. Clear the TMR0IF ("Timer 0 Interrupt Flag") that had been set by making it "0."
    a. Look at Table 3 for a reminder of how to **clear** a bit in a register.
3. Flash an LED. For instance, use `RC5` and its LED on the Curiosity board.
    a. Look at Table 3 for a reminder of how to **toggle** a bit in a register.

Table 3 Common C bitwise operations[1], with equivalent modern alternate operator spelling (ISO646 variant).[2]

	Major Bitwise Logic Operation	**Generic form** *(Replace myvar or mybit with your own variable or register and n with the bit number)*	**Explanation of Example**	**Typical** *(explicit)*	**Typical** *(compact with shifting)*	**"Alt ISO646"** *(explicit)*	**"Alt ISO646"** *(compact with shifting)*
***Set**: Assign a bit to 1*	*Or*	`myvar |= (1 << n);`	*Make Bit 6 a "1".*	`myvar = myvar	0b01000000;`	`myvar |= (1 << 6);`	`myvar = myvar bitor 0b01000000;`	`myvar or_eq (1<<3);`
***Clear**: Assign a bit to 0*	*And, Complement*	`myvar &= ~(1 << n);`	*Make Bit 5 a "0".*	`myvar = myvar & 0b11011111;`	`myvar &= ~(1 << 5);`	`myvar = myvar bitand 0b10111111;`	`myvar and_eq compl(1<<5);`
***Toggle**: Invert a bit*	*Xor*	`myvar ^= (1<<n);`	*Change Bit 4 to its opposite.*	`myvar = myvar ^ 0b01000000;`	`myvar ^= (1 << 4);`	`myvar xor_eq 0b00100000;`	`myvar xor_eq (1UL << 5);`
*Examine **status** of a bit*	*And*	`mybit = (number >> n) & 1;`	*What is the value of bit 5?*	-	`mybit = (myvar >> 5) & 1;`	-	`bit = (myvar >> 5) bitand 1;`

Keep the ISR *short* because it needs to do its job *quickly*. Don't add delays, don't add complicated equations in them. The "rule of thumb" for a typical ISR should be between two and ten lines of C code.

Show the interrupt-based heartbeat flashing LED to the lab instructor when the three prescale bits are set to 1, 1 and 1. The LED should be flashing about once a second. Then, modify your code so that the ISR heartbeat runs at the following frequency:

- **Group GE4-1:** Make the interrupt-driven heartbeat LED flash (RC5) about twice a second (2 Hz)
  - The ISR frequency should be *about* 4 Hz
- **Group GE4-2:** Make the interrupt-driven heartbeat LED flash (RC5) about four times a second (4 Hz)
  - The ISR frequency should be *about* 8 Hz
- **Group GE4-3:** Make the interrupt-driven heartbeat LED flash (RA2) about twice a second (2 Hz)
  - The ISR frequency should be *about* 4 Hz
- **Group MIQ4-1:** Make the interrupt-driven heartbeat LED flash (RA2) about four times a second (4 Hz)
  - The ISR frequency should be *about* 8 Hz
- **Group MIQ4-2:** Make the interrupt-driven heartbeat LED flash (RC5) about twice a second (2 Hz)
  - The ISR frequency should be *about* 4 Hz

[1] Standard bitwise operations are explained here: https://en.wikipedia.org/wiki/Bitwise_operations_in_C

[2] The alternate operator spelling is common in modern C++. To use the alternate operator spelling in C make sure to use **#include <iso646.h>** in your code.

Part 2: Mixing Assembler and C in an ISR

Single Sentence Overview

Modify the Interrupt Service Routine in Part 1b to be *hybrid* C *and* Assembler.

Background material

Embedded systems are often written in a mixture of C and Assembler. C helps speed up the code writing, makes the code more understandable, easier to maintain and portable from one chip to another. However, some routines that might be misunderstood by the compiler and optimizer can be written in Assembler. The best practice, in these cases, is to write the majority of your program in C and to write specific elements in Assembler.

Interrupt service routines are normally short and are not usually meant to have variables passed in or returned from. Efficiency is important and so they are good candidates for the use of Assembler. Simple bit manipulation commands are especially efficient in Assembler. If you needed to clear Bit 5 in a register called `MYREGISTER` then the command would be

```
BCF MYREGISTER, 5
```

In an Interrupt Service Routine you can call this assembler command using the `asm()` function:

```
asm("BCF MYREGISTER, 5);
```

In this way, you can write most of your program in C, with select calls in Assembler. This hybrid approach is a reasonable compromise between development and code execution efficiencies. An example ISR would look something like

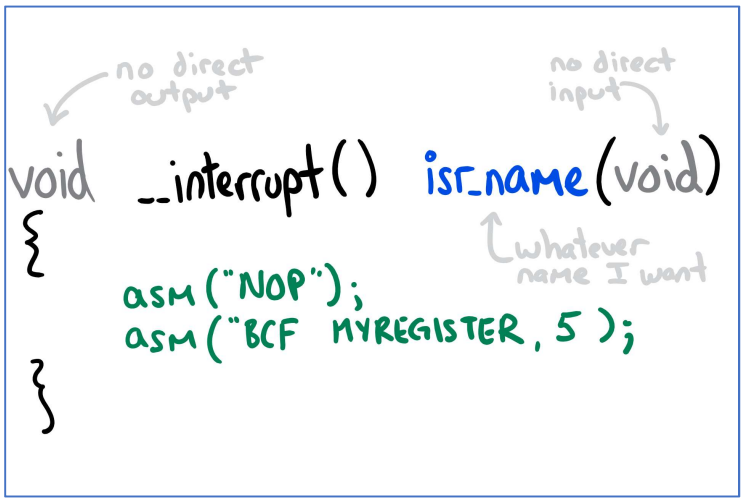

Figure 6 A hybrid approach to writing code involves calling Assembler commands using the C function, asm().

Do you need to set all eight bits in register `MYREGISTER`? Do it like this:

```
asm("MOVLW      0b00101100");    // Store 8 bits in Register W
```

```
asm("MOVWF      MYREGISTER");    // W's contents go to MYREGISTER
```

whereas, if you want to toggle (inverse) the value of Bit 2 in a register you can use XOR logic like this:

```
asm("MOVLW      0b00000100");    // I want Bit 2.
asm("XORWF      MYREGISTER,f"); // Toggle Bit 2 in MYREGISTER
```

The first line is a "mask" where the 1 signifies the position of the bit in register `MYREGISTER` that you wish to toggle. The second line XOR's the contents of `Register W` with the value of `MYREGISTER`. The "f" at the end says to store the result back in the file register, `MYREGISTER`.

Don't forget assembler operations like "Bit Clear", "Bit Set", "Bank Select", "No Operation", etc!

Problem

Using Assembler commands in the ISR, create an interrupt service routine that does the following. Demonstrate to the lab instructor.

- *Section GE4 Group 1*: Make the interrupt-driven heartbeat LED flash (RC5) about four times a second (4 Hz)
- *Section GE4 Group* 2: Make the interrupt-driven heartbeat LED flash (RC5) about twice a second (2 Hz)
- *Section GE4 Group 3*: Make the interrupt-driven heartbeat LED flash (RA2) about four times a second (4 Hz)
- *Section MIQ4 Group 1*: Make the interrupt-driven heartbeat LED flash (RA2) about twice a second (2 Hz)
- *Section MIQ4 Group 2*: Make the interrupt-driven heartbeat LED flash (RC5) about four times a second (4Hz)

Part 3: Analyzing your Code using Disassembly.

Single Sentence Overview

Examine the disassembled view of your code to see what the PIC16 is really doing behind the scenes.

Background material

While we can often trust our C compiler to make the microcontroller do what we intended, sometimes it does not. The optimization schemes used can sometimes remove code, especially when the process is non-linear and depends on unpredictable external (to the chip) events that the compiler and optimizer have no way of predicting or understanding. It is for these reasons that we sometimes add explicit lines of Assembler code. These Assembler lines bypass the predictive work done by the compiler. In other cases you will need to review the output of the compiler to verify that it did the right thing.

There are multiple was to view your C program code from an Assembler perspective:

1. Window -> Debugging -> Disassembly
    a. Usually requires you to switch to Simulation mode and to launch the Debugging View
2. Window -> Debugging -> Output -> Disassembly Listing File
3. Window -> Target Memory Views -> Program Memory Views

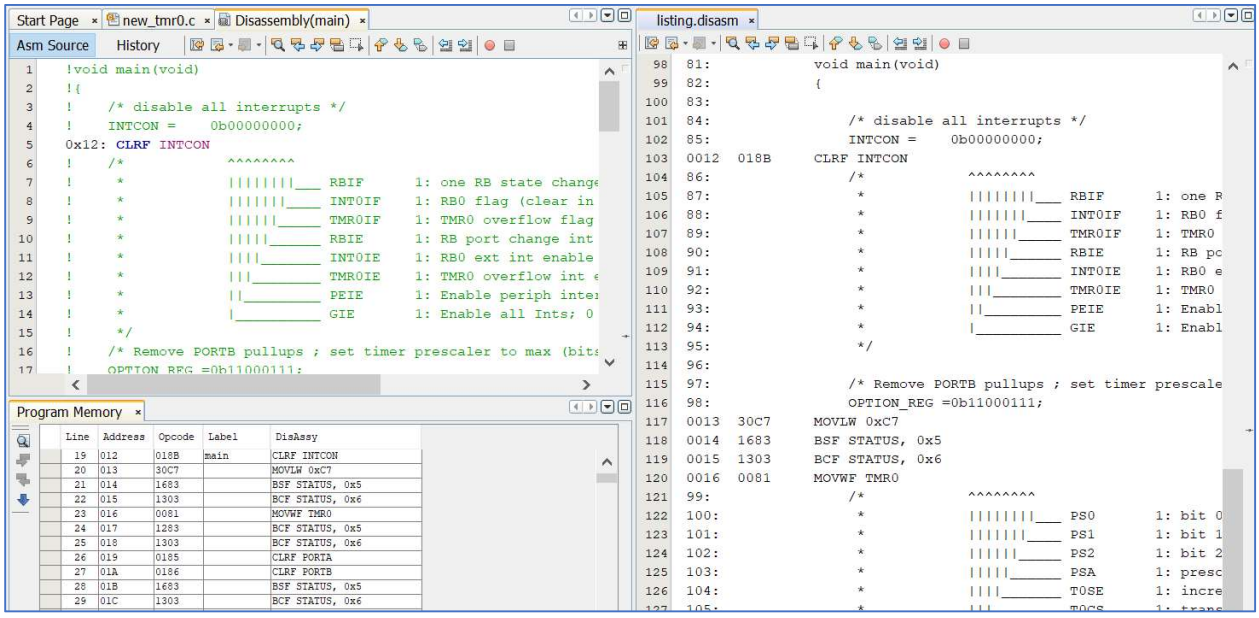

Figure 7 The Disassembly (upper left), Disassembly Listing (right) and Program Memory (lower left) views of the source code. You can view the Assembler code generated from your C code and, during debugging, you can use the debugger's stepping functions to examine execution one Assembler line at a time.

Stimulate

1. Does the main function that you wrote in Part 2 end with the last curly brace? Check the Disassembly Listing.

Explore

1. What is the last Assembler command at the end of the Interrupt Service Routine that you wrote in Part 2? *[Hint: The Disassembly Listing is the easiest way to view it.]*
2. What is the last Assembler command after the end of the Main function from Part 2?

There is no demonstration for this part. Simply answer the questions.

Part 4: Wrap-up

Reflection on Learning

Speak to your lab partner and other members of the class about the problem of multiple sources of interruption. What do you think you would have to do if three or four internal and external events occurred that attempted to interrupt the processor at nearly the same time.

Communication – Reporting

Read the grading rubric at the beginning of this document. Make sure that you demonstrate your working programs as discussed. Submit a file to your lab instructor with the answers to the "Stimulate" and "Explore" questions.

Datasheet pages

7.6 Register Definitions: Interrupt Control

REGISTER 7-1: INTCON: INTERRUPT CONTROL REGISTER

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R-0/0
GIE(1)	PEIE(2)	TMR0IE	INTE	IOCIE	TMR0IF	INTF	IOCIF(3)
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
'1' = Bit is set	'0' = Bit is cleared	

bit 7 **GIE:** Global Interrupt Enable bit(1)
1 = Enables all active interrupts
0 = Disables all interrupts

bit 6 **PEIE:** Peripheral Interrupt Enable bit(2)
1 = Enables all active peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **TMR0IE:** Timer0 Overflow Interrupt Enable bit
1 = Enables the Timer0 interrupt
0 = Disables the Timer0 interrupt

bit 4 **INTE:** INT External Interrupt Enable bit
1 = Enables the INT external interrupt
0 = Disables the INT external interrupt

bit 3 **IOCIE:** Interrupt-on-Change Enable bit
1 = Enables the interrupt-on-change
0 = Disables the interrupt-on-change

bit 2 **TMR0IF:** Timer0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed
0 = TMR0 register did not overflow

bit 1 **INTF:** INT External Interrupt Flag bit
1 = The INT external interrupt occurred
0 = The INT external interrupt did not occur

bit 0 **IOCIF:** Interrupt-on-Change Interrupt Flag bit(3)
1 = When at least one of the interrupt-on-change pins changed state
0 = None of the interrupt-on-change pins have changed state

Note 1: Interrupt flag bits are set when an interrupt condition occurs, regardless of the state of its corresponding enable bit or the Global Interrupt Enable bit, GIE of the INTCON register. User software should ensure the appropriate interrupt flag bits are clear prior to enabling an interrupt.

2: Bit PEIE of the INTCON register must be set to enable any peripheral interrupt.

3: The IOCIF Flag bit is read-only and cleared when all the interrupt-on-change flags in the IOCxF registers have been cleared by software.

Figure 8 Interrupt Control (INTCON) details. Source: https://bit.ly/2OsQtQ1 Section 7.6

TABLE 34-3: ENHANCED MID-RANGE INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C, DC, Z	2
ADDWFC	f, d	Add with Carry W and f	1	11	1101	dfff	ffff	C, DC, Z	2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	2
ASRF	f, d	Arithmetic Right Shift	1	11	0111	dfff	ffff	C, Z	2
LSLF	f, d	Logical Left Shift	1	11	0101	dfff	ffff	C, Z	2
LSRF	f, d	Logical Right Shift	1	11	0110	dfff	ffff	C, Z	2
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z	2
CLRW	–	Clear W	1	00	0001	0000	00xx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	2
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	2
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	2
MOVWF	f	Move W to f	1	00	0000	1fff	ffff		2
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C, DC, Z	2
SUBWFB	f, d	Subtract with Borrow W from f	1	11	1011	dfff	ffff	C, DC, Z	2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	2
BYTE ORIENTED SKIP OPERATIONS									
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1, 2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1, 2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		2
BIT-ORIENTED SKIP OPERATIONS									
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		1, 2
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		1, 2
LITERAL OPERATIONS									
ADDLW	k	Add literal and W	1	11	1110	kkkk	kkkk	C, DC, Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLB	k	Move literal to BSR	1	00	0000	001k	kkkk		
MOVLP	k	Move literal to PCLATH	1	11	0001	1kkk	kkkk		
MOVLW	k	Move literal to W	1	11	0000	kkkk	kkkk		
SUBLW	k	Subtract W from literal	1	11	1100	kkkk	kkkk	C, DC, Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: If the Program Counter (PC) is modified, or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

2: If this instruction addresses an INDF register and the MSb of the corresponding FSR is set, this instruction will require one additional instruction cycle.

TABLE 34-3: ENHANCED MID-RANGE INSTRUCTION SET (CONTINUED)

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
CONTROL OPERATIONS									
BRA	k	Relative Branch	2	11	001k	kkkk	kkkk		
BRW	–	Relative Branch with W	2	00	0000	0000	1011		
CALL	k	Call Subroutine	2	10	0kkk	kkkk	kkkk		
CALLW	–	Call Subroutine with W	2	00	0000	0000	1010		
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
RETFIE	k	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	0100	kkkk	kkkk		
RETURN	–	Return from Subroutine	2	00	0000	0000	1000		
INHERENT OPERATIONS									
CLRWDT	–	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$, $\overline{PD}$	
NOP	–	No Operation	1	00	0000	0000	0000		
OPTION	–	Load OPTION_REG register with W	1	00	0000	0110	0010		
RESET	–	Software device Reset	1	00	0000	0000	0001		
SLEEP	–	Go into Standby mode	1	00	0000	0110	0011	$\overline{TO}$, $\overline{PD}$	
TRIS	f	Load TRIS register with W	1	00	0000	0110	0fff		
C-COMPILER OPTIMIZED									
ADDFSR	n, k	Add Literal k to FSRn	1	11	0001	0nkk	kkkk		
MOVIW	n mm	Move Indirect FSRn to W with pre/post inc/dec modifier, mm	1	00	0000	0001	0nmm kkkk	Z	**2, 3**
	k[n]	Move INDFn to W, Indexed Indirect.	1	11	1111	0nkk	1nmm	Z	**2**
MOVWI	n mm	Move W to Indirect FSRn with pre/post inc/dec modifier, mm	1	00	0000	0001	kkkk		**2, 3**
	k[n]	Move W to INDFn, Indexed Indirect.	1	11	1111	1nkk			**2**

Note 1: If the Program Counter (PC) is modified, or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

2: If this instruction addresses an INDF register and the MSb of the corresponding FSR is set, this instruction will require one additional instruction cycle.

3: See Table in the MOVIW and MOVWI instruction descriptions.