CSCE 236 Embedded Systems, Spring 2019
Lab 4

Due: Thursday, April 4th, 2019 (end of class)

Names of Group Members: _____

_____

1. Instructions

This is a group assignment to work on during class. You only need to hand in one copy of this, but make sure that the names of all of your group members are on this sheet to receive credit. Complete all of the sections below and make sure to get the instructor or TA to sign off where required. You should keep your own notes on what you complete since parts of future homework will build on this lab.

Include a printouts of the code with your assignment. **Each team member must turn in a copy of your code on Canvas!** Failing to electronically turn in your code will result in up to a 20 point penalty on this assignment. Points may also be deducted for coding errors, poor style, or poor commenting. There will be two code files for this lab.

2. Button Interrupt

In this section you will implement an interrupt handler for button presses. There are two pins, labeled **INT0** and **INT1** on the Arduino Schematic, that can be configured to trigger an interrupt on a transition, rising edge, or falling edge[1]. Start by identifying the pin on the Arduino that corresponds to **INT1**. Connect your button to this pin (if it isn't already). In the code, make sure you enable the internal pullup for this pin (if you don't have an external pullup) and that it is set as an input. Make sure that your button is working by using the standard non-interrupt based code to turn on an LED when the button is pressed.

Now look at section 17 (External Interrupts) of the datasheet to determine how to enable interrupts on **INT1**. Pay particular attention to the descriptions of registers **EICRA** and **EIMSK**. Configure these registers to:

- Generate an interrupt on INT1 on the falling edge (this is when the button is pressed).
- Enable an interrupt on INT1.

In your code, insert the following Interrupt Service Routine for the External Interrupt Request:

```
ISR(INT1_vect){
   //INT1 interrupt handling code goes here
}
```

[1] Actually all pins can be configured to trigger an interrupt on a changing state, but there is only a single interrupt handler for all of the other pins, so we will use one of the pins that has a dedicated interrupt handler.

This is a special macro function (note that using **SIGNAL** instead of **ISR** also works, but that is deprecated) that tells the compiler to properly configure the interrupt vector to point to this function whenever the interrupt **INT1** occurs. Now write code to do the following:

- In the **INT1** interrupt handler, turn on the green LED whenever the handler is executed (this is whenever the button is pressed).
- In your main loop() if the button is not pressed, turn off the green LED.

In this configuration, the interrupt handler turns on the LED and the main loop turns off the LED.

Checkoff: *(You can get checked off for this question along with the other questions at the end of this section.) The LED does not always turn on and off as expected sometimes. Why?*

It is possible to disable interrupts by using the function cli() and to enable interrupts using the function sei(). Use these to fix part of the above problem.

It is also possible to fix this problem without disabling/enabling interrupts by having the interrupt trigger whenever any logic change occurs on the pin (by changing register **EICRA**). Use this approach as well to fix this problem.

Checkoff: *Describe the advantages and disadvantages of both solutions used to correct the above problem. Make sure to save the solutions as two different sketches so you can show the instructor both approaches.*

Instructor sign off: _____

3. Real-Time Events

Sometimes it is important to run a particular event at a specific frequency. One way to do it is to periodically check the millis() command to see if the desired number of milliseconds has elapsed and if so, you can run the event. This is fine if this is the only task you are performing, but you can run into trouble if other tasks are occurring. You may have noticed during the project competition that if you printed a lot of debug information the rate at which you could read the sensors decreased. In this section, you will blink the LED at a fixed rate using the timers and interrupts, but first you will implement blinking the "old fashion" way of using millis().

Download the lab sample code from the course website. For the moment, ignore the timer interrupt setup code and interrupt handler. You will see that normally the LED toggles every 500ms. When the button is pressed (you may need to change the pin your button and LED are connected to in the defines at the top of the code) the values of the analog input pins and digital input pins are printed.

Checkoff: *What happens to the blinking rate when you press the button? Is it consistent? (You can get checked off for this question with the next checkoff.)*

Now configure the timer to do the blinking in an interrupt. You should comment out the blinking code from the main loop and uncomment the **toggleLED()** function call in the interrupt handler. In the function **setupTimerInterrupt()** you need to determine the proper value of the register **OCR1A** to cause an interrupt to be generated every 500ms (this is the only thing you need to change). You can remind yourself how the timers work by looking at the register descriptions in section 20.

Checkoff: *What value did you use for OCR1A and how did you figure this out? What happens now when you press the button? Does the blinking rate change?*

Instructor sign off: _____

Using interrupts generated by the timers is a good way to make sure events happen at the desired frequency. For instance, the millis() function uses an interrupt based on Timer0 to count the number of elapsed milliseconds. However, you must be careful not to put too much code in the interrupt handler since that may prevent the main loop from executing or interrupts may be missed.