CSC 230 - SUMMER 2018
INTRODUCTION TO COMPUTER ARCHITECTURE
ASSIGNMENT 2
UNIVERSITY OF VICTORIA

Due: Tuesday July 3rd, 2018 before 11:55pm. **Late assignments will not be accepted.**

1 Overview

This assignment covers a wide variety of hardware programming topics, including I/O, polling and interrupt handling. It also covers new assembly programming techniques, like the use of the call stack and interrupt flags. You should start as early as possible; this assignment will be virtually impossible to complete if you leave it to the last minute, especially since you will have to use the lab hardware to test your implementation.

The basic requirement of the assignment is to write an AVR assembly program which, when uploaded to the ATmega2560 board, produces a simple "back and forth" pattern on the 6 blue LED lights connected to pins 42, 44, 46, 48, 50 and 52. Beyond the basic animated pattern, you will also have to include support for simple user input with the 5 black buttons to receive full marks.

Your assignment will be marked during an interactive demo with a member of the CSC 230 teaching team. You will be graded on the function of your implementation, the quality of your code, and your ability to justify the implementation decisions you made.

Section 2 describes the animated pattern. Section 3 describes the required user input features, and Section 4 describes a set of additional enhancements (for 100%, you must implement at least one of these extra enhancements).

2 Animated Pattern

The pattern consists of 10 steps which repeat indefinitely. By default, the delay between steps will be 1 second (and your code should strive to achieve timekeeping as accurately as possible). The program may be set to one of two display modes. In 'Regular Mode', one LED is lit at each step and the lit LED sweeps back and forth across the shield. In 'Inverted Mode', five LEDs are lit, with one LED unlit, and the unlit LED sweeps back and forth across the shield.

To receive the initial set of marks for the 'basic pattern' (see Section 7), your implementation must be able to achieve, at minimum, the Regular Mode pattern with a 1 second delay. In the following sections, the term 'step' is used to refer to the individual steps of the pattern, and the term 'transition' is used to refer to the progression between two adjacent steps in the pattern (with the understanding that the pattern repeats indefinitely and wraps around to step 0 after step 9).

Step	**LEDs** (by PIN number)												
	Regular Mode							Inverted Mode					
	52	50	48	46	44	42		52	50	48	46	44	42
0	●	○	○	○	○	○		○	●	●	●	●	●
1	○	●	○	○	○	○		●	○	●	●	●	●
2	○	○	●	○	○	○		●	●	○	●	●	●
3	○	○	○	●	○	○		●	●	●	○	●	●
4	○	○	○	○	●	○		●	●	●	●	○	●
5	○	○	○	○	○	●		●	●	●	●	●	○
6	○	○	○	○	●	○		●	●	●	●	○	●
7	○	○	○	●	○	○		●	●	●	○	●	●
8	○	○	●	○	○	○		●	●	○	●	●	●
9	○	●	○	○	○	○		●	○	●	●	●	●
After step 9, pattern repeats from step 0													

Legend	
●	= LED on
○	= LED off

3 Required Inputs

As you saw in Lab 5, the LCD shield used on the AVR boards in ECS 249 has five general purpose black buttons (plus the `RST` button which is not easily programmable). You are required to implement the following behaviors for the `UP`, `DOWN`, `LEFT` and `RIGHT` buttons. Note that this section does not use the `SELECT` button, since that button is used for the extra features in Section 4.

Button	Behavior
`LEFT`	**Enable Inverted Mode**: When the `LEFT` button is pressed, the program will **immediately** enter Inverted Mode and the set of lit LEDs will immediately reflect the inverted configuration for the current step of the pattern, as given by the table in Section 2. The change must take effect immediately (not at the beginning of the next step) to be correct[1]. If the button is pressed while Inverted Mode is already active, there is no effect (and the program remains in Inverted Mode).
`RIGHT`	**Enable Regular Mode**: When the `RIGHT` button is pressed, the program will **immediately** enter Regular Mode and the set of lit LEDs will immediately reflect the non-inverted configuration for the current step of the pattern, as given by the table in Section 2. The change must take effect immediately (not at the beginning of the next step) to be correct. If the button is pressed while Regular Mode is already active, there is no effect (and the program remains in Regular Mode).
`UP`	**Change Delay to 0.25 Seconds**: When the `UP` button is pressed, the delay between steps will become 0.25 seconds. If the delay is already set to 0.25 seconds, then pressing the `UP` button will have no effect (and the delay will remain at 0.25 seconds). The new timing must take effect at or before the next pattern transition.
`DOWN`	**Change Delay to 1 Second**: When the `DOWN` button is pressed, the delay between steps will become 1 second. If the delay is already set to 1 second, then pressing the `DOWN` button will have no effect (and the delay will remain at 1 second). The new timing must take effect at or before the next pattern transition.

4 Additional Enhancements

For full marks, you must implement one of the three finishing touches documented in Sections 4.1, 4.2 and 4.3 below. You are welcome to implement more than one (if feasible), but you cannot receive more than 100% on the assignment. If you have your own idea for an extra feature, you may be allowed to implement that instead of one of the items below, but you must get written permission from your instructor before the due date or you will not receive any marks for the feature.

4.1 Accelerate Mode

This feature uses the `SELECT` button to enable a third speed setting, in addition to the standard 1 second delay provided by the `DOWN` button and the 0.25 second delay provided by the `UP` button. If this feature is implemented, the behaviors of `UP`, `DOWN` and `SELECT` must follow the table below. The behavior of `LEFT` and `RIGHT` are unchanged.

1. For example, if the delay between steps is 1 second and the button is pressed 0.5 seconds after the last transition, the inverted configuration of the current step should be displayed for 0.5 seconds before the next transition

Button	Behavior
UP	**Change Delay to 0.25 Seconds**: When the UP button is pressed, accelerate mode will be disabled (if active) and the delay between steps will become 0.25 seconds. If the delay is already set to 0.25 seconds, then pressing the UP button will have no effect (and the delay will remain at 0.25 seconds). The new timing must take effect at or before the next pattern transition.
DOWN	**Change Delay to 1 Second**: When the DOWN button is pressed, accelerate mode will be disabled (if active) and the delay between steps will become 1 second. If the delay is already set to 1 second, then pressing the DOWN button will have no effect (and the delay will remain at 1 second). The new timing must take effect at or before the next pattern transition.
SELECT	**Enable Accelerate Mode**: When the SELECT button is pressed, accelerate mode will be enabled and the delay between steps will become 1 second. Even if accelerate mode is already active, the delay will be changed to 1 second. The new timing must take effect at or before the next pattern transition.

When accelerate mode is active, the delay between steps will be **reduced by half** every time the pattern reaches step 0 until the delay is 1/32 seconds (after which point the delay will remain at 1/32 seconds until the delay is changed by an input. When accelerate mode is started, the delay will be set to 1 second. At each successive iteration of step 0 of the pattern, the delay will progress through $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$ and finally $\frac{1}{32}$ second intervals. The delay will not change on any other step besides step 0. Note that if the UP or DOWN buttons are pressed, accelerate mode must be disabled. Additionally, accelerate mode must be disabled when the program starts (so the only way to initiate accelerate mode is by pressing SELECT).

4.2 Variable Speeds

This feature alters the behavior of the UP and DOWN buttons to cycle through a set of different delays, instead of just switching between two preset speeds. When successfully implemented, this feature will allow delays of $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$ and $\frac{1}{32}$ seconds to be selected by pressing the UP and DOWN buttons. If this feature is implemented, the behaviors of UP and DOWN must follow the table below. The behavior of LEFT and RIGHT are unchanged.

Button	Behavior
UP	**Decrease delay by factor of 2**: When the UP button is pressed, the delay between steps will be reduced by a factor of 2, to a minimum delay of $\frac{1}{32}$ seconds. If the active delay is $\frac{1}{32}$ seconds, pressing the UP button will have no effect (under no circumstances should a delay of less than $\frac{1}{32}$ seconds be used). The new timing must take effect at or before the next pattern transition.
DOWN	**Increase delay by factor of 2**: When the DOWN button is pressed, the delay between steps will be increased by a factor of 2, to a maximum delay of 1 second. If the active delay is 1 second, pressing the DOWN button will have no effect (under no circumstances should a delay of more than 1 second be used). The new timing must take effect at or before the next pattern transition.

4.3 Pause Button

This feature uses the `SELECT` button to toggle a "pause" state, in which the pattern is frozen at the current step. When the program begins, pause mode is not active (so the pattern animates as usual). When the `SELECT` button is pressed, pause mode is toggled (so pressing the button when the program is paused will un-pause it, and pressing the button when the program is un-paused will pause it).

When pause mode is active, the pattern does not transition between steps, and remains fixed at the step that was active when the pause button was pressed. However, all other functionality still behaves as normal: In particular, the entire set of user inputs from Section 3 must continue to work. Specifically, if the display mode (Regular/Inverted) is changed, the new mode must be reflected immediately (even when paused), and if the speed is changed, the new speed must take effect as soon as the program is unpaused (however, changing the speed while pause mode is active does **not** un-pause the program).

4.4 Avoiding Erroneous Multiple Inputs

The features described in Sections 4.3 and 4.2 are sensitive to button inputs being read repeatedly. For example, consider the fact that a single press of a button (such as `SELECT`) might take 0.25 seconds (based on the typical time for a human to press a button). During those 0.25 seconds, your program may poll the buttons thousands of times, and each time read that the button was pressed. This is not a problem when the button has only one behavior, such as enabling Inverted Mode, since the multiple reads will just result in Inverted Mode being enabled repeatedly. However, when the same button has multiple behaviors (such as both pausing and unpausing the program), the multiple read problem can result in incorrect behavior.

You will lose marks if the multiple read problem affects your program's behavior (for example, if pressing the pause button does not reliably toggle the pause state). You can avert the problem by using a button handling routine like the one given in the pseudocode below, which sets an `IGNORE_BUTTON` variable whenever a button is pressed and clears it when the button is released.

```
//Input loop
while(1){
    Read the button value from the ADC
    if (no button was pressed){
        IGNORE_BUTTON = 0 //Clear the ignore flag
    }else{
        //Handle the button normally

        //Set the ignore flag
        IGNORE_BUTTON = 1
    }
}
```

5 Implementation Suggestions

This section contains a general guide for implementing a solution. You are not required to follow this progression, but may find it helpful if you don't know where to start.

5.1 The Most Basic Option

If you can implement the basic pattern with no input, using a delay loop for the time delay, you can receive up to 10/18 (see Section 7 below for details). You are encouraged to start by implementing this variant, and then add other features (timer-based delays, user-inputs, etc.) once you have tested the basic pattern. The simple loop-based delay can be modelled by the C-style pseudocode below.

```
//Initialization
//Use a variable CURRENT_LED to track the currently lit LED with
//an index in the range 0-5.
//You will need to write some conversion code, probably a function,
//which converts the LED index to a port and bit number.
CURRENT_LED = 0
DIRECTION = 1 //Use a variable to track the "direction" of the LED
while(1){ // The program continues running indefinitely
    Clear all LEDs
    Set CURRENT_LED to be lit.
    CURRENT_LED += DIRECTION
    //If we have reached LED 0, set the DIRECTION to be 1
    if (CURRENT_LED == 0)
        DIRECTION = 1
    //If we have reached LED 5, set the DIRECTION to be -1
    else if (CURRENT_LED == 5)
        DIRECTION = -1
    Wait for one second (using a loop)
}
```

If you use a loop-based delay, you should ensure that its running time is as close to one second as possible (by counting clock cycles). You may lose marks for inaccurate timing code.

5.2 Using Timers

Once you have tested the loop-based delay and verified that the logic for incrementing and lighting the LEDs is correct, add an interrupt service routine (ISR) for one of the on-board timers (you are not required to use a particular timer, but may find it easier to use timer 0 or timer 2 on this assignment). You can then modify your code to use one of the two following program designs.

The first design uses a 'light' interrupt service routine which only increments the LED number (and leaves all of the other work, such as actually changing the lit LED) to the main loop). In the pseudocode below, C functions are used to model the different program components; note that interrupt service routines do not behave like normal functions since they require the use of the `RETI` instruction. The variables `CURRENT_LED` and `DIRECTION` are assumed to be global variables accessible by all parts of the code.

```
void interrupt_service_routine(){
    Compute whether the correct delay has elapsed
    (since the ISR may be called repeatedly before the delay has elapsed)

    if (the correct delay has elapsed){
        CURRENT_LED += DIRECTION
        //If we have reached LED 0, set the DIRECTION to be 1
        if (CURRENT_LED == 0)
            DIRECTION = 1
        //If we have reached LED 5, set the DIRECTION to be -1
        else if (CURRENT_LED == 5)
            DIRECTION = -1
    }
}

void main_program(){
    //Initialization
    CURRENT_LED = 0
    DIRECTION = 1

    Set up a timer and enable interrupts (both timer interrupt and the global interrupt flag)

    while(1){
        Clear all LEDs
        Set CURRENT_LED to be lit.

        //Add button input handlers here when ready
    }
}
```

The second design uses a 'heavy' interrupt service routine which handles both the incrementation and lighting of the LEDs.

```
void interrupt_service_routine(){
    Compute whether the correct delay has elapsed
    (since the ISR may be called repeatedly before the delay has elapsed)

    if (the correct delay has elapsed){
        CURRENT_LED += DIRECTION
        //If we have reached LED 0, set the DIRECTION to be 1
        if (CURRENT_LED == 0)
            DIRECTION = 1
        //If we have reached LED 5, set the DIRECTION to be -1
        else if (CURRENT_LED == 5)
            DIRECTION = -1
```

```
        Clear all LEDs
        Set CURRENT_LED to be lit

    }
}

void main_program(){
    //Initialization
    CURRENT_LED = 0
    DIRECTION = 1

    Set up a timer and enable interrupts (both timer interrupt and the global interrupt flag)

    while(1){
        //Add button input handlers here when ready
    }
}
```

6 Adding an Input Handler

Since button input is detected with polling, you will need to use a busy-wait loop to test for button inputs. If you followed one of the patterns in the previous section, you should consider adding the busy-wait loop to the main program in the main loop (as a nested loop inside the `while(1)` infinite loop). You are strongly encouraged to write a function which reads the ADC and returns an index between 0 and 5 (inclusive) corresponding to the button pressed (for example, with 0 meaning no button was pressed, 1 meaning `RIGHT` was pressed, etc.), then calling that function from the main loop and handling the result appropriately.

6.1 Mitigating Button Aliasing

Constantly polling the buttons to detect button presses can result in unusual behavior. In particular, when buttons with low voltages (such as `RIGHT`) are pressed, polling the buttons might result in the voltage for a different button being read. For this assignment, it is expected that your code will proactively prevent button aliasing to the greatest extent possible (although it may be impossible to prevent the aliasing in all cases). To reduce the impact of aliasing, consider modifying your button polling logic to use one of the following techniques (or a different technique of your own design).

- Add a short delay (e.g. 0.05 seconds) between consecutive polls to reduce the probability that an erroneous transient value will be read.
- Instead of polling 'constantly', poll the buttons at regular intervals (e.g. 10 times per second). As long as the buttons are polled more often than the timer ticks, there will be no distinguishable difference in behavior if the frequency of polling is reduced.
- Use a function to poll the ADC and return a number in the range $[0, 5]$ based on which button was detected. Instead of simply calling the function to poll the buttons, call the function twice

(or three times) and only acknowledge the button press if the return value was the same for all calls.

7 Evaluation

Submit all `.asm` and `.inc` files needed to assemble your assignment electronically via conneX. Your code must assemble, upload and run correctly on the ATmega2560 boards in ECS 249 using the toolchain and methodology described in Lab 4. If your code does not assemble as submitted, you will be allowed to spend some of your demo time attempting to fix it, but the demo will be limited to 10 minutes, and you will lose marks if you miss parts of the evaluation by spending time fixing your code. If your code cannot be assembled during the demo, you will receive a mark of at most 2.

This assignment is worth 9% of your final grade and will be marked out of 18 during an interactive demo with an instructor. Demos must be scheduled in advance (through an electronic system available on conneX). If you do not schedule a demo time, or if you do not attend your scheduled demo, you will receive a mark of zero.

The marks are distributed among the aspects of the assignment as follows.

Marks	Aspect
6	The 'basic pattern' described in Section 2 (non-inverted with a 1 second interval) functions correctly.
4	The code uses functions and arrays to avoid unnecessary duplication and cumbersome control flow structures (such as long chains of `if`/`else` logic). In particular, logic for button polling and setting the LEDs should be encapsulated in a function, and the code to distinguish between the different steps of the animated pattern should be implemented without a long `if`/`else` construct (instead, consider using arrays to store the bit patterns for the LED configuration at each step of the pattern). Functions are expected to be documented with comments indicating the set of parameters and return value. It is acceptable to use registers for parameters and return values instead of the stack, but the stack should be used correctly for pushing and popping register and return values.
2	Timekeeping is achieved with one of the timer peripherals (using interrupts) instead of a delay loop. You may be expected to explain the timekeeping scheme and justify its accuracy.
4	The behaviors specified in Section 3 for the up/down/left/right buttons are correctly implemented, and pressing a particular button (including the `SELECT` button if applicable) produces **only** the prescribed behavior for that button. If button aliasing is observed during the demo, marks will be deducted.
2	One (or more) of the 'extra features' described in Section 4 is implemented and functions correctly. If you implement multiple extra features, your total mark is capped at 100% (18/18), but you can make up marks lost in other sections. If your implementation does not **exactly** reflect the requirements in the relevant subsection of Section 4, you will lose marks in this component (even for seemingly minor issues like omitting one of the possible delay values in accelerate mode or variable speed mode). You will not receive any credit for extra features which aren't described in Section 4 unless you have received prior permission from your instructor.

If your code is not well organized, or if it is poorly documented, the evaluator may ask you explain any aspects that are unclear. If you are unable to do so, up to 2 marks may be deducted. To be

clear, you are not required to have spotless, perfectly organized code, but you should be prepared to explain any hard-to-read parts of your code.

You are permitted to delete and resubmit your assignment as many times as you want before the due date, but no submissions or resubmissions will be accepted after the due date has passed. You will receive a mark of zero if you have not officially submitted your assignment (and received a confirmation email) before the due date. Ensure that each submitted file contains a comment with your name and student number.

Ensure that all code files needed to assemble, upload and run your code in ECS 249 are submitted. Only the files that you submit through conneX will be marked. The best way to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. **If you do not receive such an email, you did not submit the assignment.** If you have problems with the submission process, send an email to the instructor **before** the due date.