Computer Science 230
Computer Architecture and Assembly Language
Spring 2018

Assignment 4

Due: Friday, April 6th, 11:55 pm by conneX submission
(Late submissions accepted up to Monday, April 9th, 11:55 pm with a 20% penalty)

Programming environment

For this assignment you must ensure your work executes correctly on Arduino boards in ECS 249.

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Zastre).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found and used in your solution must be cited in comments just before where such code has been used.

Objectives of this assignment

- Work with timers, interrupts and the LEDs using the C programming language.

C programming language

In this assignment you will write several functions for a C language program. This programs uses the LEDs on the lab boards (i.e., to keep this assignment simple, you will not need to use the LCD panel, and therefore need only work with a single C source file). The code given to you has already configured two of the mega2560 timers (timers 1 and 3). Interrupt handlers for these timers are already provided (i.e., you are not being asked to implement interrupt handlers). A brief demonstration illustrating the behavior for each of the parts can be seen in this video:

`https://youtu.be/iR6NfrghyjU`

The assignment is in four parts, ordered from easier to more difficult, each part building upon your confidence in completing the previous part.

A. Write the code for `lcd_state()`.
B. Write the code for `SOS()` which uses both your completed `lcd_state()` plus arrays with LED and duration data.
C. Write the code for `glow()`.
D. Write the code for `pulse_glow()`.

There is also a bonus available for those who have successfully completed all four parts. It is called `light_show()` and is to implement the LED pattern shown on the YouTube video (link above).

The ZIP file distributed with this assignment contains a single directory consisting of an AVR Studio 4 C-language project. The skeleton file `a4.c` is also contained within this project directory. I have set the configuration of the project such that paths for both the compiler and build tools are correctly set for machines in the lab.

Part A: `lcd_state()`

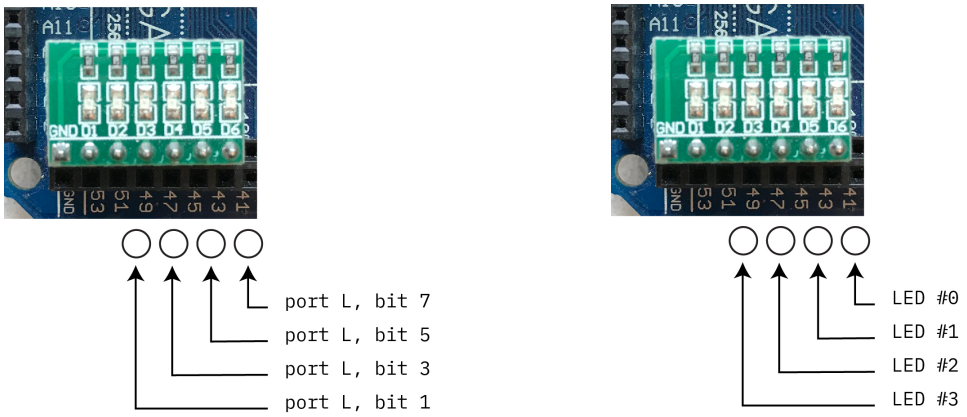
This part of the assignment does not involve interrupts.

In assignment #2 you wrote an assembly-language function named `leds_on`. For this part you will write a function that accepts two parameters:

1. The number of an LCD; and
2. A number indicating the state to which that LCD must be put (with a zero value meaning "off", and all other values meaning "on").

The function is therefore to turn an LED **on** or **off** and immediately return.

To simply your work, your function need only deal with PORTL LEDs as numbered in the following diagram.



You can also take advantage of notational convenience when writing to I/O registers in AVR C. Assuming the DDR register for the port has been properly configured, turning on LED #0 can be accomplished via:

```
PORTL |= 0b10000000;
```

while turning off LED #0 can be performed by:

```
PORTL &= 0b01111111;
```

which is equivalent to:

```
PORTL &= ~(0b10000000);
```

(Hint: This function might be a good candidate for including the use of a `switch` statement.)

Part B: `SOS()`

This part of the assignment does not directly involve the use interrupts.

By now you are familiar with Morse code, specifically that for SOS (*dot dot dot*, *dash dash dash*, *dot dot dot*). We can implement dots and dashes by calls to `led_state()` and `_delay_ms()`.

You are to write a function to cause SOS to be displayed, but there is a twist here. In the `SOS()` function are three variables:

1. `uint8_t light[]`: an array of 8-bit values indicate the LED pattern. An array value indicates LEDs on or off by bits set or cleared, with bit 0 the state for LED #0, bit 1 the state for LED #1, etc.
2. `int duration[]`: an array of `int`s (i.e., 16-bit values) representating the duration in milliseconds for an LED pattern.
3. `int length`: the number of elements in `light[]` and the number of elements in `duration[]`.

For example, at index 6 the bit pattern indicates LED #0 is turned on for 100 milliseconds).

So the twist is that you must cause the LEDs to flash appropriately for SOS by using these arrays: they already contain the patterns and durations required for SOS. You can use the arrays by writing a loop in which defined/declared appropriate counter variables, plus calls to `led_state()` and `_delay_ms()`. (A `for`-loop is perhaps a better choice here than a `while`-loop).

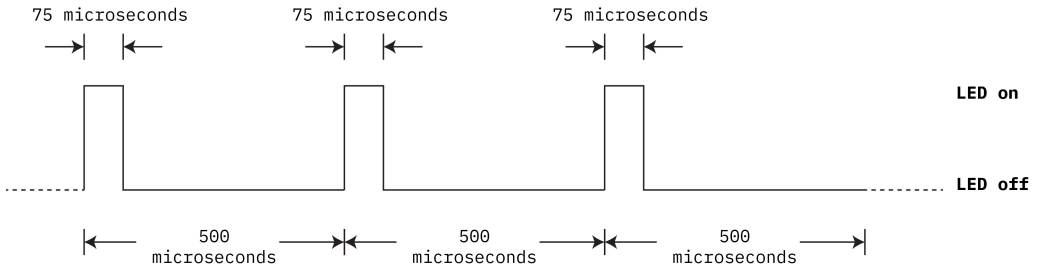
Part C: `glow()`

This part of the assignment **must** use the program-scope variable named `count` that is incremented by the timer 1 interrupt handler. When testing your solution, we will evaluate `glow()` separately from `pulse_glow()`, i.e., we will call one or the other function not both in the same test run.

Our current LEDs can be either on or off. There is, however, no easy way to set their relative brightness. Put differently, we control LED brightness by adjusting current through the LED, but we cannot adjust the current on our boards.

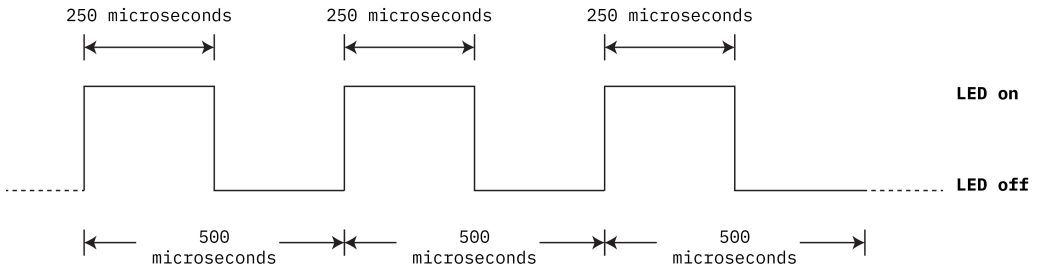
There is, however, another way to get something like a brightness control that exploits something called **pulse-width modulation**. In essence it means turning an LED on and off, doing so very rapidly, and in a way that gives the appearance of brightness or dimness.

Suppose we wish to have a somewhat dim LED. Consider the following diagram.



This shows that every 500 microseconds, the LED is turned on for 75 microseconds and then turned off for 425 microseconds. This cycle of 500 microseconds goes on for as long as dimmer LED intensity is needed.

And suppose we want a brighter LED, but not as bright as the fully-on LED. Consider the next diagram.



Now for 500 microseconds, the LED is turned on for 250 microseconds and then turned off for 250 microseconds.

So when the LED is on, we say the signal is *high*, and the time for which is the signal is high is called the *on time*. Given than our period here is 500 microseconds, our first diagram shows an *on time* of 75 microseconds, and the percentage of *on time* to

the overall period is called the *duty cycle* (i.e., 75/500 = 0.15 = 15% duty cycle). For our second diagram, the duty cycle is 50% (i.e., 250/500 = 0.5 = 50% duty cycle). Going further, a duty cycle of 0% would be an LED that is completely off, while a duty cycle of 100% would be an LED that is completely on.

You are to implement this notion of a duty cycle. Function `glow()` takes two parameters:

1. The number of an LCD; and
2. The duty cycle for that LCD expressed as a floating-point value between 0.0 and 1.0.

In order to implement this function you must use an infinite loop, and therefore you will only be able to ever set the glow for a single LED. The infinite loop will take advantage of the fact that the global variable `count` is incremented once a microsecond. The loop's form will most likely need to be somewhat similar to:

```
threshold = PWM_PERIOD * given duty cycle
do forever {
    if (count < threshold and LED is not already on) {
        turn the LED on;
    } else if (count < PWM_PERIOD and LED is not already off) {
        turn the LED off;
    } else { /* count must be greater than PWM_PERIOD */
       count = 0;
       turn the LED on;
    }
}
```

Note that `PWM_PERIOD` is given in the top-most "DO NO TOUCH" section of `a4.c`.

Warning: Even though we might expect the LEDs to respond in a visually linear fashion as we change the duty cycle, unfortunately our own visual systems *do not* respond in a similar way. That is, the differences in brightness that we see as increase the duty cycle from 10% to 20% to 30% to 40% etc. will not "appear" as evenly-spaced changes to our eyes and brains. Don't worry about this (or if you insist on worrying about this, please spend some time at `https://bit.ly/2pGn61A` **after** you have finished the assignment).

Part D: `pulse_glow()`

This part of the assignment **must** use the program-scope variables named `count` and `slow_count` that are incremented by the timer 1 and timer 3 interrupt handlers respectively. When testing your solution, we will evaluate `pulse_glow()` separately from `glow()`, i.e., we will call one or the other function but not both in the same test run.

Once you have completed `glow()`, you might have some intuition of how to implement code to mimic those glowing LEDs appearing on some devices. That is, to *increase* and *decrease* the brightness over time, we need to *increase* and *decrease* the duty cycle over time. As our proxy for duty time in Part C was the value assigned to the variable `threshold`, this means changing the value of `threshold` over time.

As with Part C, in order to implement Part D you must use an infinite loop, and therefore you will only be able have one LED pulsing. The infinite loop will take advantage of the fact that the global variables `count` and `slow_count` are incremented once a microsecond and once ever 10 milliseconds respectively. Your loop will need to modify `threshold` by increasing it to `PWM_PERIOD` or decreasing it to `0` in some way related to changes in `slow_count`. While your solution need not mimic exactly the rate of pulsing that is shown in the YouTube video, it must be very similar.

Bonus: `light_show()`

(This part of the assignment does not involve interrupts.)

Using the array idea from Part B, implement the light-show pattern using a loop that iterates through you own versions of `light[]` and `duration[]` to produce the pattern in the YouTube video.

What you must submit

- Your completed `a4.c`. **Do not change the name of this file!** Do not submit any other project files.
- Your work must use the provided skeleton files. Any other kinds of solutions will not be accepted.

Evaluation

- 4 marks: Solution for part A
- 4 marks: solution for part B
- 6 marks: solution for part C
- 6 marks: solution for part D
- 2 marks: solution for the bonus

The total mark for this assignment is 20 (i.e., if full marks given for parts A, B, C and D, and the bonus is correctly implemented, the mark will be 22/20).

Some of the evaluation will also take into account whether or not submitted code is properly formatted (i.e., indenting and commenting are suitably used), and the file correctly named.