Overview

The following lab will consist of three parts: The first part asks you to apply the knowledge you’ve gained in Lab 2.1 to a more complex problem; The second part introduces you to timers using an example; The third part requires you to apply what you will learn in Part 2 to the system that you created in Part 1. The point breakdown for these three parts is as follows:

- Part 1: Traffic Light Utilizing Delay-Loops - 5 points
- Part 2: Blinking LED Utilizing A Timer – 5 points
- Part 3: Traffic Light Utilizing A Timer – 5 points
- Single report covering all three experiments – 5 points

Part 1: Traffic Light Utilizing Delay-Loops

Design a controller for the traffic light on Dole Street near the Parking Structure (in the Quarry) using the PIC16F84A. That traffic light controls the automobile traffic on Dole Street and the pedestrian traffic crossing the street. The outputs of the controller are Red (R), Yellow (Y), Green (G), Don't-Walk (D), and Walk (W). There are two inputs, B1 and B2, which are each connected to a Pedestrian-Walk button. The outputs and inputs correspond to the pins of the 16F84A as follows R = RB0, Y = RB1, G = RB2, D = RB3, W = RB4, B1 = RA0, and B2 = RA1.
There are four states of the traffic light as shown in Figure 1:

- **R/W1**: Traffic light is Red and pedestrian light is Walk. The controller is in this state for 3 seconds.
- **R/W2**: Traffic light is Red and pedestrian light is a blinking Walk. The Walk is blinking to caution pedestrian that it will soon turn to Don't Walk. The Walk goes on 1/2 sec, off 1/2 sec, on 1/2 sec and so on. The controller is in this state for 3 seconds.
- **G/D**: Traffic light is Green and pedestrian light is Don't Walk. The controller is in this state for at least 6 seconds. The controller will get out of this state when
  - 6 seconds have elapsed *and*
  - one of the pedestrian buttons has been pushed (goes on) since the end of the last R/W1 state, i.e., the beginning of the last R/W2 state.
- **Y/D**: Traffic light is Yellow and pedestrian light is Don't Walk. The controller is in this state for 3 seconds.

![Figure 1. Transitions of traffic light controller.](image)

To check if a pedestrian button were pressed, the controller samples RA0 and RA1 every 1 ms to check if they are on. *(Hint: To facilitate debugging, have an additional output (LED) to indicate if a pedestrian button (B1 and B2) has been pressed.)*

**Part 2: Blinking LED Utilizing A Timer**

In real-world applications, delay loops are not normally used. One reason is that they are inefficient. They use CPU clock cycles just waiting rather than doing work. A second reason is it can complicate programming since a detailed knowledge of the machine instructions is required. To help alleviate these complications, some microprocessors include modules called timers. This part of the lab is designed to introduce you to timers. Use the code contained in the file “timerLED.c” as a template to create a program that causes an LED to blink every other second. In order to test your knowledge of how to set a timer, you must also change each timer “time-tick” to equate to roughly 50,000 us, instead of 1 ms as in the sample code. See Appendix A for an explanation of the code.
Part 3: Traffic Light Utilizing A Timer

Update your traffic light controller in Part 1 to use a timer module instead of delay loops.

Appendix A: Description of timerLED.c

The following program is similar to timerLED.c but simpler. We will go over this first. The program will drive an LED to blink on and off, where the duration of a complete cycle is 1 second.

```c
#include <pic1684.h>
#define INSTR_CYCLE 1 // The instruction cycle in microseconds (us). Note that this
                   //   this is the duration between increments of Timer 0 without
                   //   the prescalar. We will control the prescalar so Timer 0
                   //   will increment 1:32 times slower than the instruction cycle
#define MSDELAY 1000/(32*INSTR_CYCLE) // The number of times Timer 0 increments
                                       //   for a 1 ms delay assuming prescalar 1:32

void main(void)
{
    unsigned char LEDstatus=0;  // Keeps track of LED status. We need this because
                                 // we cannot read from an output port. Recall that a port can
                                 // be either an input or output but not both.
    unsigned int elapsed;  // Keeps track of elapsed time in milliseconds.

    // *** Configuring the output to the LED ***
    TRISA2=0;       // Configure RA2 to be an output -- which is attached to the LED
    RA2=LEDstatus;  // Set RA2 output to LEDstatus

    // **** Configuring the OPTION REGISTER ****
    T0CS=0; // Sets up Timer 0 clock signal source to come from the instruction cycle
             // clock. The instruction cycle clock speed can be found in data sheet.
             // This is explained in timerLED.c in its comments. For now just assume that the
             // instruction cycle rate is 1 MHz.
    PSA=0;  // Assign the prescalar to timer0 rather than the watchdog timer.
    PS2=1; PS1=0; PS0=0; // Sets the prescalar to be 1:32 -- see Sec 2.3.2 in data sheet.
                         // This means that Timer 0 will increment at a rate 1:32 slower than the
                         // instruction cycle rate.

    // *** Main program ***
    while (1) {
        // Delay of 0.5 s
        for (elapsed = 0; elapsed < 500; elapsed++) { // Delay = 500 * 1ms = 0.5s
            TMR0 = 0; // Clear Timer 0.
            while (TMR0 < MSDELAY) ; // Delay of 1 ms. Recall that TMR0 increments every
                                      //   32*INSTR_CYCLE us
        }

        // Change LED
        LEDstatus = ~LEDstatus;
        RA2 = LEDstatus;
    }
}
```

Notice that the program is written completely in the C language. There are no machine instructions such as in Lab 2.1, where delays were created by NOP instructions. Having the program completely written in C language has at least two
advantages. First, it simplifies programming. A user who is unfamiliar with the microprocessor is not required to learn the details of the machine instructions. Second, the code is more portable. Thus, if a better processor is found or if there is another implementation of the PIC1684A with different clock cycles per instruction, the code could be reused for them as well. The code may still have to be changed for the particular clock rate and register sizes of the new processor but the changes will be minimized.

Now the actual program timerLED.c is slightly different. It takes advantage of features of Timer 0. Recall that Timer 0 is an 8-bit counter. Thus, its range of values is 0 through 255 (in hexadecimal, it’s 0x0 through 0xff; here, the prefix “0x” means hexadecimal). When the counter reaches 255, incrementing will cause it to wrap around down to 0. This is often called “overflow” since the value has overflowed the bit-size of the counter.

Timer 0 overflowing will cause two (possible) events. First, a flag TOIF is set to 1 to indicate the occurrence of the overflow. A “flag” is a 1-bit register which is used to indicate the status of something. The TOIF flag can be read and written to. The next lines show how the program can be modified to use this flag. The modifications are shown in yellow.

```c
// *** Main program ***
while (1) {
    // Delay of 0.5 s
    for (delayms = 0; delayms < 500; delayms++) { // Delay = 500 * 1ms = 0.5s
        TMR0 = 256 - MSDELAY; // Now TMR0 must increment MSDELAY times before
        //     causing an overflow
        TOIF = 0;           // TOIF can be cleared only by software
        while (TOIF == 0) ; // Wait for overflow, which indicates a delay of 1 ms.
        // Recall that TMR0 increments every 4us
    }

    // Change LED
    LEDstatus = ~LEDstatus;
    RA2 = LEDstatus;
}
```

Timer 0 overflow will cause another (possible) event called an interrupt. You can ignore this for now but we will cover interrupts in Lab 2.3. We should mention that in order for the program to run properly, the interrupt should be disabled. (Apparently, the default for Timer 0 interrupt is to be disabled. If this is not the case then it may cause problems. You can check with the TA.)

Finally, we will briefly discuss how the prescalar works for Timer 0. Figure A.1(a) shows a 3-bit counter circuit COUNTER, which will increment every clock period (the prescalar for the PIC 16F84A uses an 8-bit counter). Assume the clock period is 1 second. Figure A.1(b) has COUNTER’s output (Y2,Y1,Y0) every clock period. Notice that Y0 is a signal that oscillates between 0 and 1, and has a period of 2 seconds. Similarly, Y1 is a signal that has a period of 4 seconds; and Y2 has a period of 8 seconds. Thus, (Y2, Y1, Y0) generates 3 clock signals with slower rates of 8:1, 4:1,
and 2:1, respectively. Figure A.1(c) shows how we can use a multiplexer (Mux) to select a rate for a Timer counter. The inputs (PS1,PS0) select the clock signal rate for the Timer.