ENSF 460 - FALL 2025

Assignment 2 (Driver Project 2)

Lab section: B02
Group #: 12
Lab Date: Friday, Sept 16 2025
Lab Due Date: Friday, Oct 3 2025

Group Members:
Patricia Agdamag
Aaron Montesines
Chase Mackenzie

Introduction:

The overall objective of this lab was to continue gaining experience programming with the microcontroller whilst also learning how to use the built-in timer peripheral and interrupt capabilities.

In Part 1, we were tasked with setting up the hardware similar to the previous lab but this time adding a new LED connection to RA6. The objective of this part was to change our system clock to 500kHz immediately after startup and implement a timer interrupt that triggers every 0.5 seconds, where the LEDs will blink on and off for 0.5 seconds.

In Part 2, we expanded the tasks performed in Part 1, by setting a certain delay depending on which push button is pressed. These tasks include:

- While PB1 is Pressed → LED1 blinks at approximately 0.25 sec intervals.
- While PB2 is Pressed → LED1 blinks at approximately 1 sec intervals.
- While PB3 is Pressed → LED1 blinks at approximately 6 sec intervals.
- While PB1 and PB2 are Pressed → LED1 blinks at approximately 1ms intervals.
- No PBs pressed → LED1 stays off
- LED2 is continuously blinking at a constant 0.5 sec when no PB is pressed.

Setup:

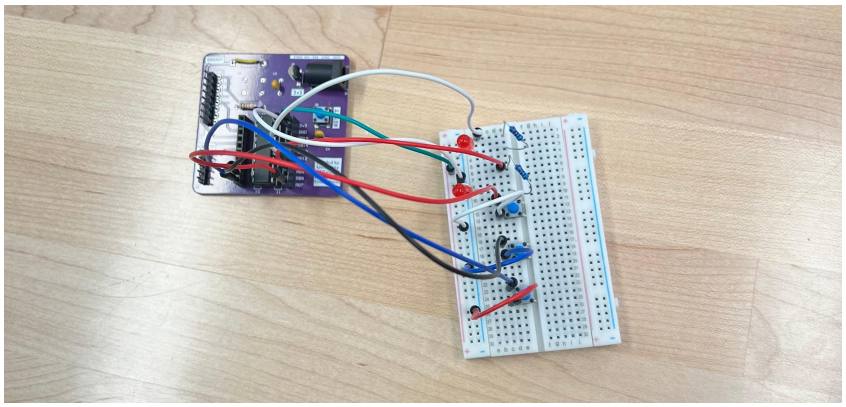


Figure 1: Breadboard and Circuit Configuration

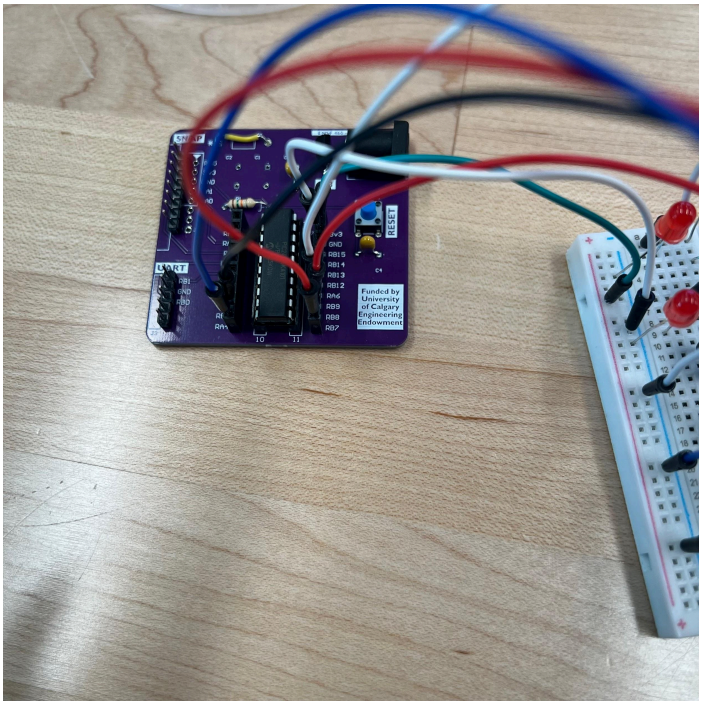


Figure 2: Pin Connection Configuration

Software Implementation:

I/O Initialization

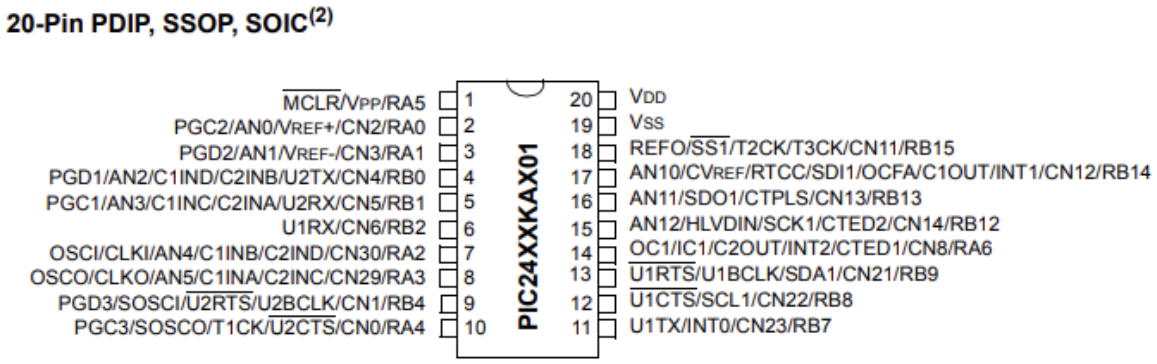


Figure 3: 20 Pin Microcontroller

For our software implementation, we configured our I/O pins similar to Lab 1. Noting Figure 3 above, we have LED1 and LED2 connected to RB9 and RA6 respectively. Their direction is set as output by setting the TRISBbits.TRISB9 and TRISAbits.TRISA6 both to equal 0. Similarly, we connected pushbuttons PB1, PB2, and PB3 to ports RB7, RB4, and RA4 respectively and set their TRIS bits to 1 to represent the pins as inputs.

Next we set the internal pull-up resistors for all the input pins to ensure they read as HIGH (1), when the buttons are not pressed. Since our pushbuttons are connected to ground, pressing on the buttons will read as LOW(0), and releasing the button will let the internal-pull up to be read as HIGH(1). This is done by using the following CNPU registers: CNPU1bits.CN0PUE = 1, CNPU2bits.CN23PUE = 1 and CNPU1bits.CN1PUE = 1 which are connected to pins A4, B7, and B4 respectively. For the output pins (LEDs), we initialized our LAT registers to LATBbits.LATB9 = 1 and LATAbits.LATA6 = 1, which turns on LED1 and LED2 respectively. Figure 4 below shows our code implementation of the initializations.

```
// I/O Initializations
TRISAbits.TRISA4 = 1;
CNPU1bits.CN0PUE = 1;

TRISBbits.TRISB7 = 1;
CNPU2bits.CN23PUE = 1;

TRISBbits.TRISB4 = 1;
CNPU1bits.CN1PUE = 1;

TRISBbits.TRISB9 = 0;
LATBbits.LATB9 = 1;

TRISAbits.TRISA6 = 0;
LATAbits.LATA6 = 1;
```

Figure 4: Input and Output Pin Configurations

Timer Configuration

```
    newClk(500);            // Sets clock to 500kHz

    // Timer 2 Config
    T2CONbits.T32 = 0;      //operate timer 2 as 16 bit timer
    T2CONbits.TCKPS = 1;    // Prescaler 1:8
    T2CONbits.TCS = 0;      // Use internal clock
    T2CONbits.TSIDL = 0;    //operate in idle mode.

    // Timer 2 interrupt Config
    IPC1bits.T2IP = 1;      // Set to lowest priority
    IFS0bits.T2IF = 0;
    IEC0bits.T2IE = 1;      // Enable timer interrupt

    PR2 = 15625;            // Set count value
    TMR2 = 0;               // Reset timer to 0.

    T2CONbits.TON = 1;      //Start timer2 LED2 since it blinks continuously

};
```

Figure 5: Timer 2 Configuration

Timer 2 was configured to operate as a 16-bit timer using the internal clock. A 1:8 prescaler was selected. The timer is set to continue operating in idle mode as according to T2CONbits.TSIDL being set to 0.The timing calculation is based on

the system clock of 500 kHz set with a 1:8 prescaler Given that the timer increments at 62500 hz the period interval is:

$$\frac{15625}{62500hz} = 0.5s$$

When the timer reaches the period value the interrupt service routine toggles LED2, causing it to blink at a 0.5 second on/off interval without blocking the main program.
Because timer 2 only dictates LED2 blinking, the timer 2 interrupt is set to the lowest priority, ensuring that other timers can preempt timer 2 if needed. To prevent false triggering the interrupt flag is cleared shortly after.

```c
void delay_ms(uint16_t time_ms) {

    // Step 1: Select an appropriate clock speed.
    //          Was set to 500kHz in IOinit();

    // Reset flag before starting.
    TMR1flag = 0;

    // Step 2: Configure T1CON register bits.
    T1CONbits.TCS = 0;
    T1CONbits.TSIDL = 0; //operate in idle mode.

    // Step 3: Clear TMR1
    TMR1 = 0;

    // Step 4: Configure Timer1 specific bits in interrupt registers IPC0, IEC0
    //          IFS1
    IPC0bits.T1IP = 1;
    IFS0bits.T1IF = 0;
    IEC0bits.T1IE = 1;  // Enable timer interrupt

    // Step 5: Compute PR1 based on desired time delay.
    // Checks if prescaler needs to be increased depending on delay.

    if (time_ms <= 2000) {
    T1CONbits.TCKPS = 0b01; // 1:8 prescaler
    // Fcy = 250kHz.
    PR1 = (uint16_t)((time_ms * 250000 / 8) / 1000);
    } else {
        T1CONbits.TCKPS = 0b10; // 1:64 prescaler
        PR1 = (uint16_t)((time_ms * 250000 / 64) / 1000);
    }

    // Step 6: Start Timer
    T1CONbits.TON = 1;

    // Step 7: Checks if interrupt finished, otherwise stay in idle mode.
    while (!TMR1flag) {
        Idle();
    }

```

Figure 6: Delay_ms function

Our delay_ms function focuses on handling the blocking delay for LED1 by using Timer1 and its interrupt. First, the function will reset the global flag TMR1flag to 0. Then it will clear the TMR1 count to 0 and set up the interrupt registers IPCO, IFSO, and IECO to enable the Timer1 interrupts at low priority. Next we need to check what prescaler to use depending on the desired delay. For delays 2s and less will use the prescaler 1:8 and delays more than 2s will use the prescaler 1:64. It will then calculate the timer period PR1 by taking the desired delay in ms, multiplied by the Fcy, and dividing by the chosen prescaler, and dividing 1000 to convert the milliseconds to seconds. Note: for the configuration of Timer1, we did not have to set it to a 16-bit timer as its default is 16bit.

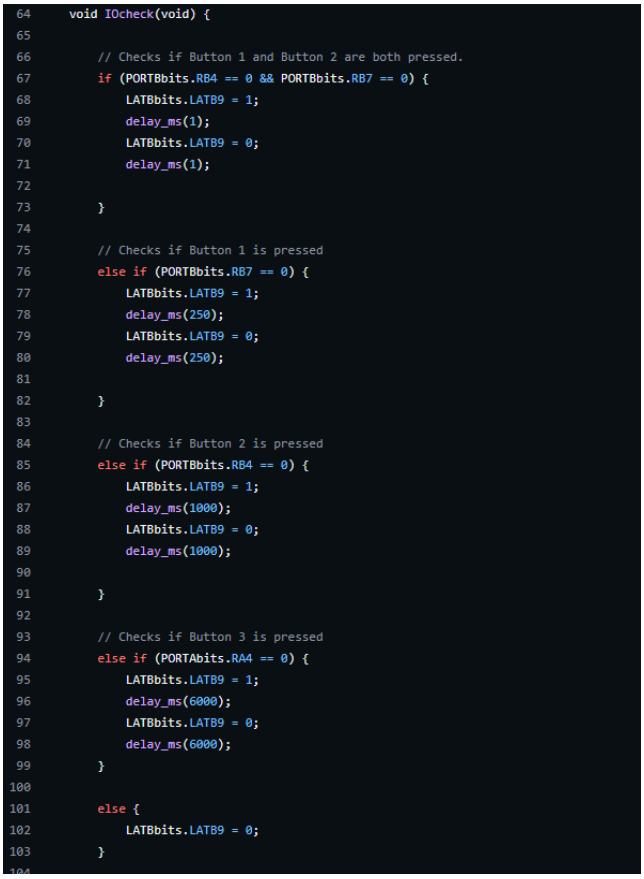
Next, we will turn on Timer1 interrupt by setting T1CONbits.TON = 1. It will then enter a while loop called Idle() which will pause the CPU until the Timer1 interrupt occurs. The TMR1 flag is used to indicate that the interrupt has fired and the delay has been completed so that the program will only continue after the requested time has elapsed. When the interrupt then fires, the timer will be stopped and sets the TMR1flag = 1, telling the program that the delay period has finished as seen in Figure 5 for the ISR of Timer 1.

```
// Timer 2 interrupt service routine
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void){
    //Don't forget to clear the timer 2 interrupt flag!
    IFS0bits.T2IF = 0; // Clear timer 2 interrupt flag

    if (led2State) {
        LATAbits.LATA6 = 0; // Turn off LED2
        led2State = 0;      // Set State of LED2 as Off
    } else {
        LATAbits.LATA6 = 1; // Turn on LED2
        led2State = 1;      // Set State of LED2 as ON
    }
};

// You might it helpful to define the interrupt service routine for Timer 1 here
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void){
    //Don't forget to clear the timer 1 interrupt flag!
    IFS0bits.T1IF = 0; // Clearing timer1 flag
    T1CONbits.TON = 0; // stops timer until next button press
        TMR1flag = 1; // Sets flag to 1 to let delay_ms() know that interrupt finished.

};
```

Figure 7: System Interrupts

Our system uses two timers with their own respective interrupt service routines to handle each LED blinking. Timer2 is configured to blink LED2 continuously at a fixed 0.5 second interval. When Timer2 interrupt fires, the ISR will toggle the LED2 output. It will then immediately clear the Timer2 interrupt flag, and check that current states of the LED, if its currently on, it will turn off the LED then set the state to be OFF, or if it's currently off, it will turn on the LED and then set the state to be ON.

For Timer1, the ISR is triggered once the timer period has elapsed. After this, In the ISR, the Timer1 interrupt flag is cleared and then the timer is stopped by setting T1CONbits.TON = 0. This is so it stops the timer until the next button press. It then sets the global TMR1flag = 1 to indicate that the desired delay is complete. Unlike Timer2, timer1 does not run continuously, only running while the button is pressed. This design allows LED2 to blink continuously through Timer2 while LED1 only blinks when required.



```
void IOcheck(void) {

    // Checks if Button 1 and Button 2 are both pressed.
    if (PORTBbits.RB4 == 0 && PORTBbits.RB7 == 0) {
        LATBbits.LATB9 = 1;
        delay_ms(1);
        LATBbits.LATB9 = 0;
        delay_ms(1);

    }

    // Checks if Button 1 is pressed
    else if (PORTBbits.RB7 == 0) {
        LATBbits.LATB9 = 1;
        delay_ms(250);
        LATBbits.LATB9 = 0;
        delay_ms(250);

    }

    // Checks if Button 2 is pressed
    else if (PORTBbits.RB4 == 0) {
        LATBbits.LATB9 = 1;
        delay_ms(1000);
        LATBbits.LATB9 = 0;
        delay_ms(1000);

    }

    // Checks if Button 3 is pressed
    else if (PORTAbits.RA4 == 0) {
        LATBbits.LATB9 = 1;
        delay_ms(6000);
        LATBbits.LATB9 = 0;
        delay_ms(6000);
    }

    else {
        LATBbits.LATB9 = 0;
    }
```

Figure 8: Button Logic

For our button logic, we implemented a function called IOCheck which is responsible for reading the pushbutton inputs and controlling LED1 based on the user's input. This function polls each button by checking the corresponding port pin. Since the pushbuttons are connected to ground with internal pull-up resistors enabled, pressing a button will read as LOW (0), and a released button will be read as HIGH (1).

The function will first check if both PB1 (connected to RB7) and PB2 (connected to RB4) are pressed at the same time. If so, the LED1 is toggled on and and off with a short delay of 1ms using the delay_ms() function. Since this delay is 1ms, it is very difficult to see the difference with the naked eye and thus will appear to just be turned on. We would also like to note that it is possible that the CPU detects one button pressed first over the other even though it's done at the same time and may do that respectively delay once and then go through the PB1 and PB2 both pressed condition.

Next it checks if only PB is pressed, where the LED1 will blink at approx. 0.25 seconds. When only PB2 is pressed, LED1 will blink at 1 sec, and PB3 is pressed, will blink at 6

seconds. If no buttons are pressed, then the function will ensure that LED1 is turned off by setting LED1 connected LATB9 to 0.

Short Answer:

In Part 1 of the lab, what values of PR2, prescaler bit, etc. did you use? Why can/can't you simply use the timer with no prescaler (prescaler set to 1:1) to time 0.5 seconds when the system is operating at 500 kHz?

- Fosc = 500 kHz
- Fcy = 250 kHz
- PR2: 15625
- Prescaler bit: 01 (1:8)
- We simply can't use a 1:1 prescaler because the timer would need to count 250 000 ticks to achieve a 0.5 second delay. Since the timer is 16-bit, the maximum count is 65535 which is not enough. Thus, we used a 1:8 prescaler to slow down the timer to allow it to reach the desired 0.5 second delay without exceeding the 16-bit limit.

In Part 2, did you use any power or time saving features (i.e. interrupts, clock switching, sleep/idle)? If so, explain where you did so, and why.

- In Part 2, we used both interrupts and the CPU's idle mode to save time and power. Timer 2 interrupts were used to toggle LED2 continuously without blocking the main program. Timer 1 interrupts were used in the delay_ms() function (refer to fig.6) to signal when a delay is complete. While waiting for Timer1 interrupt, the CPU executes Idle() to switch to low-power mode until the timer interrupt occurs.

Is it possible to put "too big a number" into your delay_ms() function? If so, why? If not, how did you prevent this?

- Consider the following formula: $T = PR / Fcy *$
    - T = Time

    - Fcy* = Fcy / prescaler

- Yes it is possible to put “too big of a number” into delay_ms() because the timer period for PR1 is 16 bits. This means the maximum value for PR1 is 65535. For delays up to 2 seconds, we used a 1:8 prescaler, which allowed PR1 to fit within the 16 bit limit.

$$T \approx 65535/(250000/8) \approx 2s \approx 2000ms$$

- For delays longer than 2 seconds, we switched to a 1:64 prescaler. The maximum input number four our 1:64 prescaler is:

$$T \approx 65535/(250000/64)\, ms \approx 16s \approx 16000ms$$

  Anything above ~16s will cause a “wrap around” and have the timer be much shorter than intended.