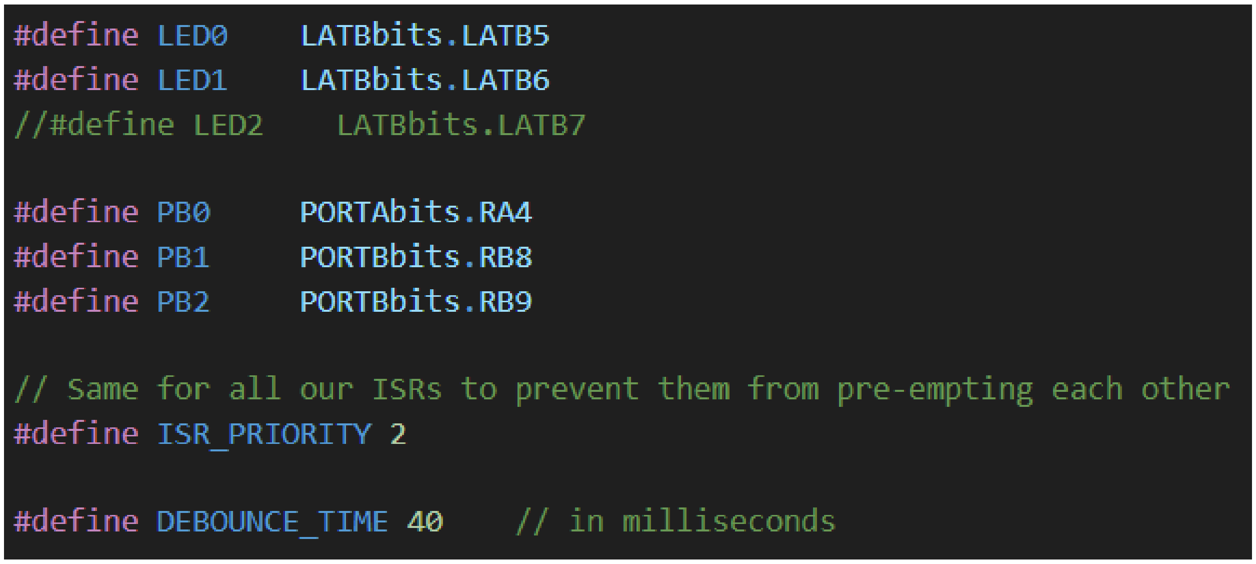
ENCM 511 Assignment2

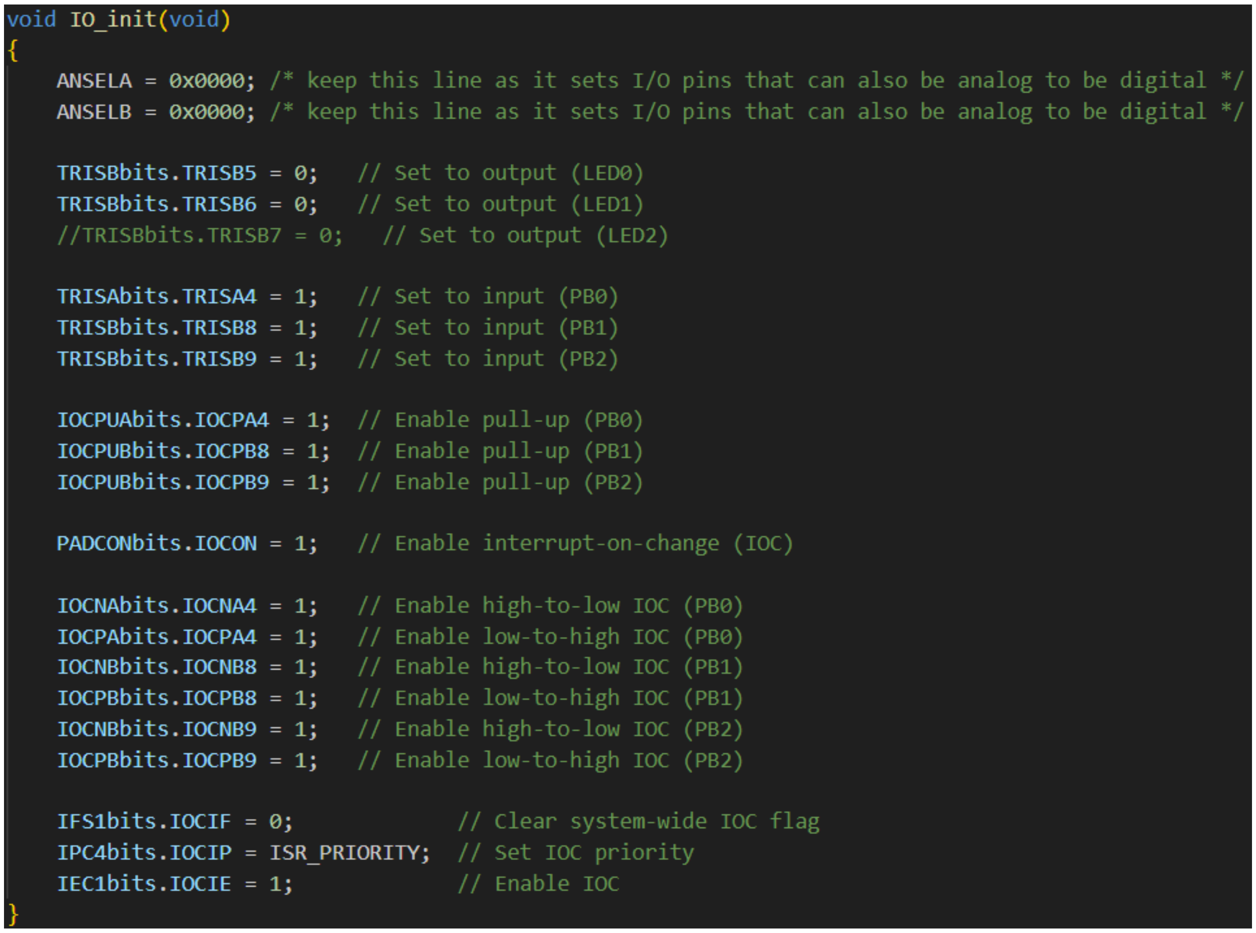
Preamble



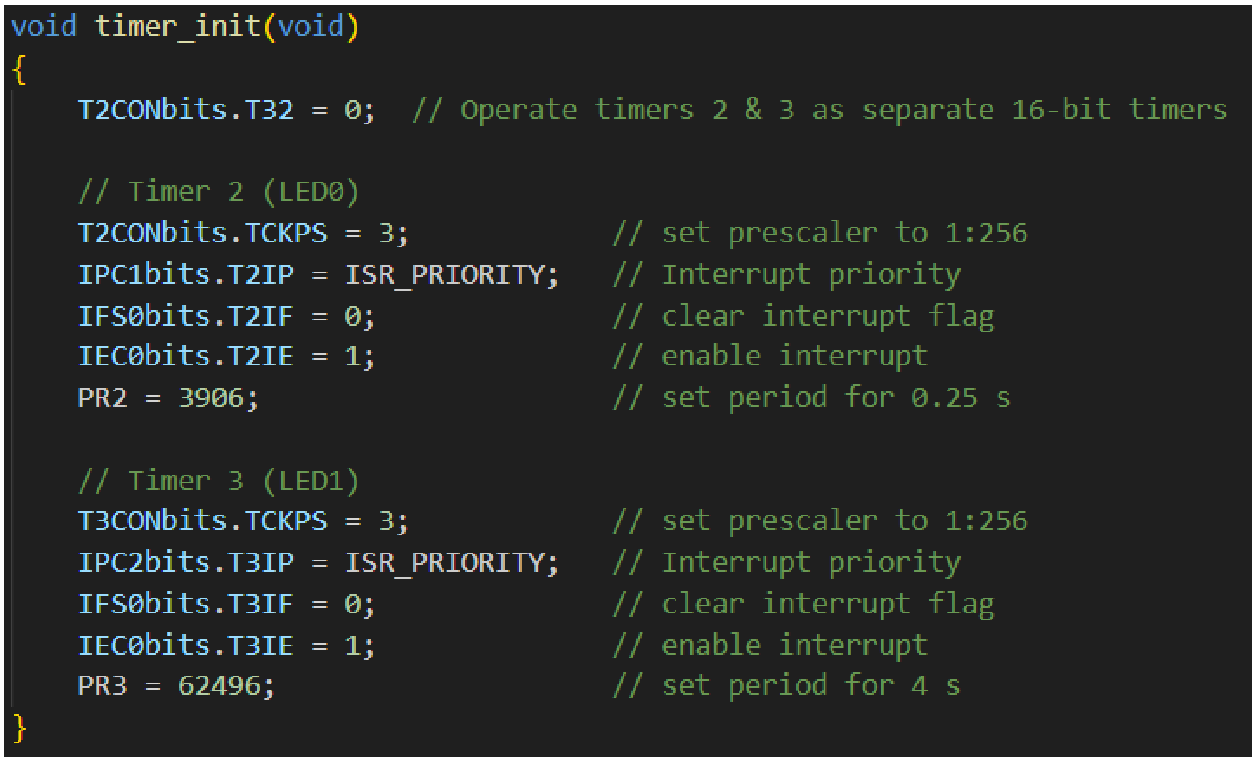
```
#define LED0    LATBbits.LATB5
#define LED1    LATBbits.LATB6
//#define LED2    LATBbits.LATB7

#define PB0     PORTAbits.RA4
#define PB1     PORTBbits.RB8
#define PB2     PORTBbits.RB9

// Same for all our ISRs to prevent them from pre-empting each other
#define ISR_PRIORITY 2

#define DEBOUNCE_TIME 40    // in milliseconds
```

The above definitions have been added at the start of the code to improve readability. Their corresponding pin mapping may be referenced in the image above. LED2 is unused in this lab and is commented out for that reason. Global ISR priority and debounce time variables have also been created to allow for easy adjustment.

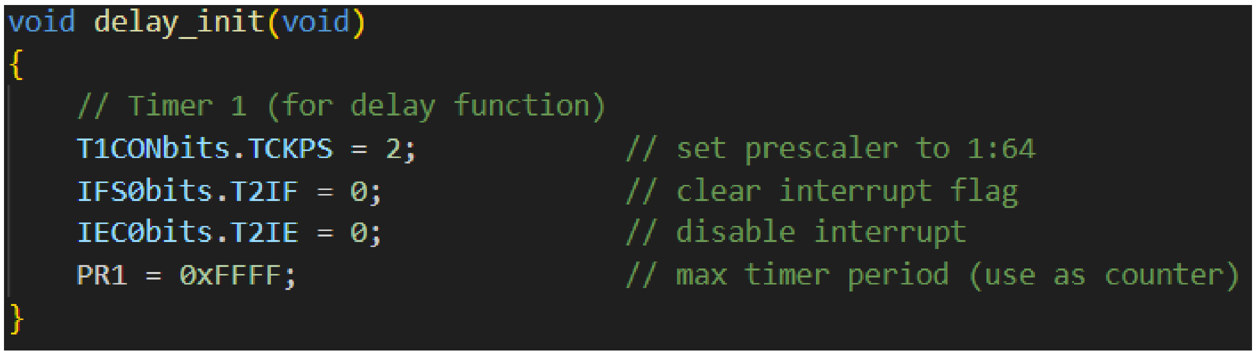
Peripherals



```c
void IO_init(void)
{
    ANSELA = 0x0000; /* keep this line as it sets I/O pins that can also be analog to be digital */
    ANSELB = 0x0000; /* keep this line as it sets I/O pins that can also be analog to be digital */

    TRISBbits.TRISB5 = 0;   // Set to output (LED0)
    TRISBbits.TRISB6 = 0;   // Set to output (LED1)
    //TRISBbits.TRISB7 = 0;   // Set to output (LED2)

    TRISAbits.TRISA4 = 1;   // Set to input (PB0)
    TRISBbits.TRISB8 = 1;   // Set to input (PB1)
    TRISBbits.TRISB9 = 1;   // Set to input (PB2)

    IOCPUAbits.IOCPA4 = 1;  // Enable pull-up (PB0)
    IOCPUBbits.IOCPB8 = 1;  // Enable pull-up (PB1)
    IOCPUBbits.IOCPB9 = 1;  // Enable pull-up (PB2)

    PADCONbits.IOCON = 1;   // Enable interrupt-on-change (IOC)

    IOCNAbits.IOCNA4 = 1;   // Enable high-to-low IOC (PB0)
    IOCPAbits.IOCPA4 = 1;   // Enable low-to-high IOC (PB0)
    IOCNBbits.IOCNB8 = 1;   // Enable high-to-low IOC (PB1)
    IOCPBbits.IOCPB8 = 1;   // Enable low-to-high IOC (PB1)
    IOCNBbits.IOCNB9 = 1;   // Enable high-to-low IOC (PB2)
    IOCPBbits.IOCPB9 = 1;   // Enable low-to-high IOC (PB2)

    IFS1bits.IOCIF = 0;             // Clear system-wide IOC flag
    IPC4bits.IOCIP = ISR_PRIORITY;  // Set IOC priority
    IEC1bits.IOCIE = 1;             // Enable IOC
}
```



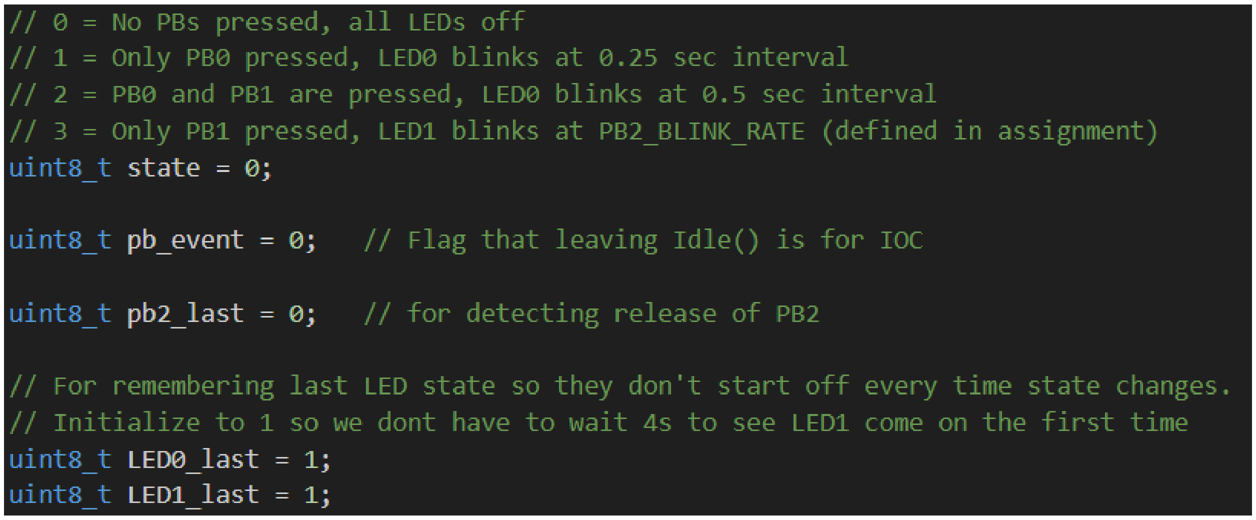
```c
void timer_init(void)
{
    T2CONbits.T32 = 0;  // Operate timers 2 & 3 as separate 16-bit timers

    // Timer 2 (LED0)
    T2CONbits.TCKPS = 3;            // set prescaler to 1:256
    IPC1bits.T2IP = ISR_PRIORITY;   // Interrupt priority
    IFS0bits.T2IF = 0;              // clear interrupt flag
    IEC0bits.T2IE = 1;              // enable interrupt
    PR2 = 3906;                     // set period for 0.25 s

    // Timer 3 (LED1)
    T3CONbits.TCKPS = 3;            // set prescaler to 1:256
    IPC2bits.T3IP = ISR_PRIORITY;   // Interrupt priority
    IFS0bits.T3IF = 0;              // clear interrupt flag
    IEC0bits.T3IE = 1;              // enable interrupt
    PR3 = 62496;                    // set period for 4 s
}
```



```c
void delay_init(void)
{
    // Timer 1 (for delay function)
    T1CONbits.TCKPS = 2;            // set prescaler to 1:64
    IFS0bits.T2IF = 0;              // clear interrupt flag
    IEC0bits.T2IE = 0;              // disable interrupt
    PR1 = 0xFFFF;                   // max timer period (use as counter)
}
```

Above is the IO_init, timer_init, and delay_init functions to be called at the beginning of main to configure IO pins and timers on the microcontroller. Line functions are elucidated by their comments.

Datatypes and Initialization



```c
// 0 = No PBs pressed, all LEDs off
// 1 = Only PB0 pressed, LED0 blinks at 0.25 sec interval
// 2 = PB0 and PB1 are pressed, LED0 blinks at 0.5 sec interval
// 3 = Only PB1 pressed, LED1 blinks at PB2_BLINK_RATE (defined in assignment)
uint8_t state = 0;

uint8_t pb_event = 0;   // Flag that leaving Idle() is for IOC

uint8_t pb2_last = 0;   // for detecting release of PB2

// For remembering last LED state so they don't start off every time state changes.
// Initialize to 1 so we dont have to wait 4s to see LED1 come on the first time
uint8_t LED0_last = 1;
uint8_t LED1_last = 1;
```

The state variable will be in the range of 0 to 3 and thus is an unsigned 8bit integer.

The pb_event variable will hold 1 or 0 and thus is an unsigned 8 bit integer.

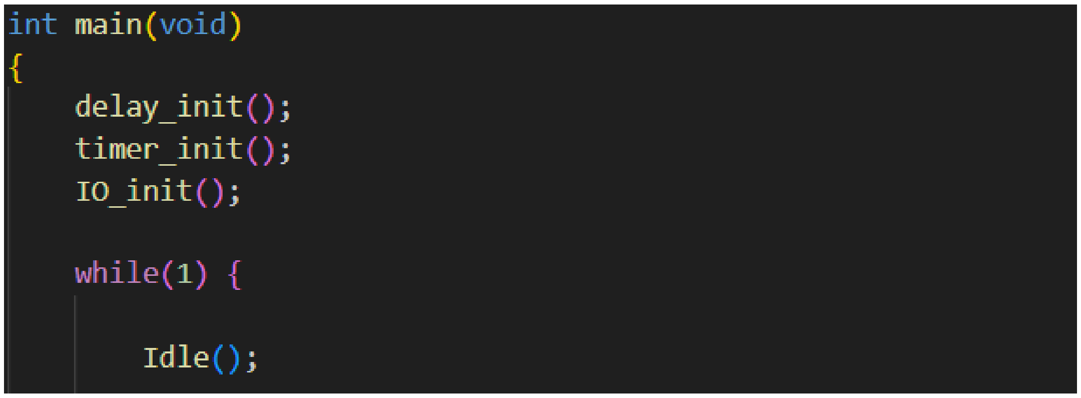
The bp2_last variable will hold 1 or 0 and thus is an unsigned 8 bit integer.

The LED0_last variable will hold 1 or 0 and thus is an unsigned 8 bit integer.

The LED1_last variable will hold 1 or 0 and thus is an unsigned 8 bit integer.

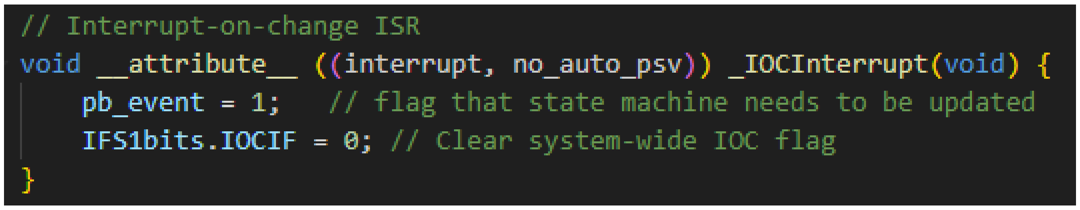
Operation

The code begins by calling all three initialization functions before entering its while loop and going into idle.



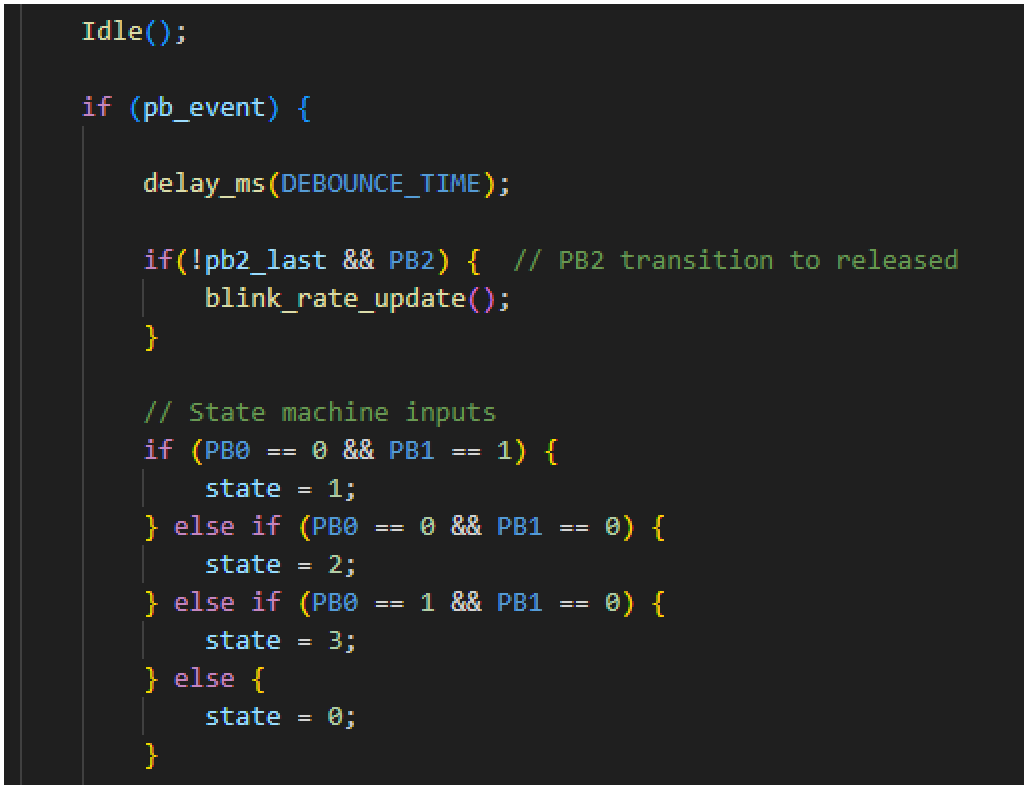
```c
int main(void)
{
    delay_init();
    timer_init();
    IO_init();

    while(1) {

        Idle();
```

Assuming this is the first run of the program, i.e. no timers are going, the microcontroller will remain in idle until one of the push buttons triggers an IOC interrupt.



```c
// Interrupt-on-change ISR
void __attribute__ ((interrupt, no_auto_psv)) _IOCInterrupt(void) {
    pb_event = 1;   // flag that state machine needs to be updated
    IFS1bits.IOCIF = 0; // Clear system-wide IOC flag
}
```

The IOC ISR will then cause the pb_event variable to be set. The flag is then cleared, and the program returns to main.

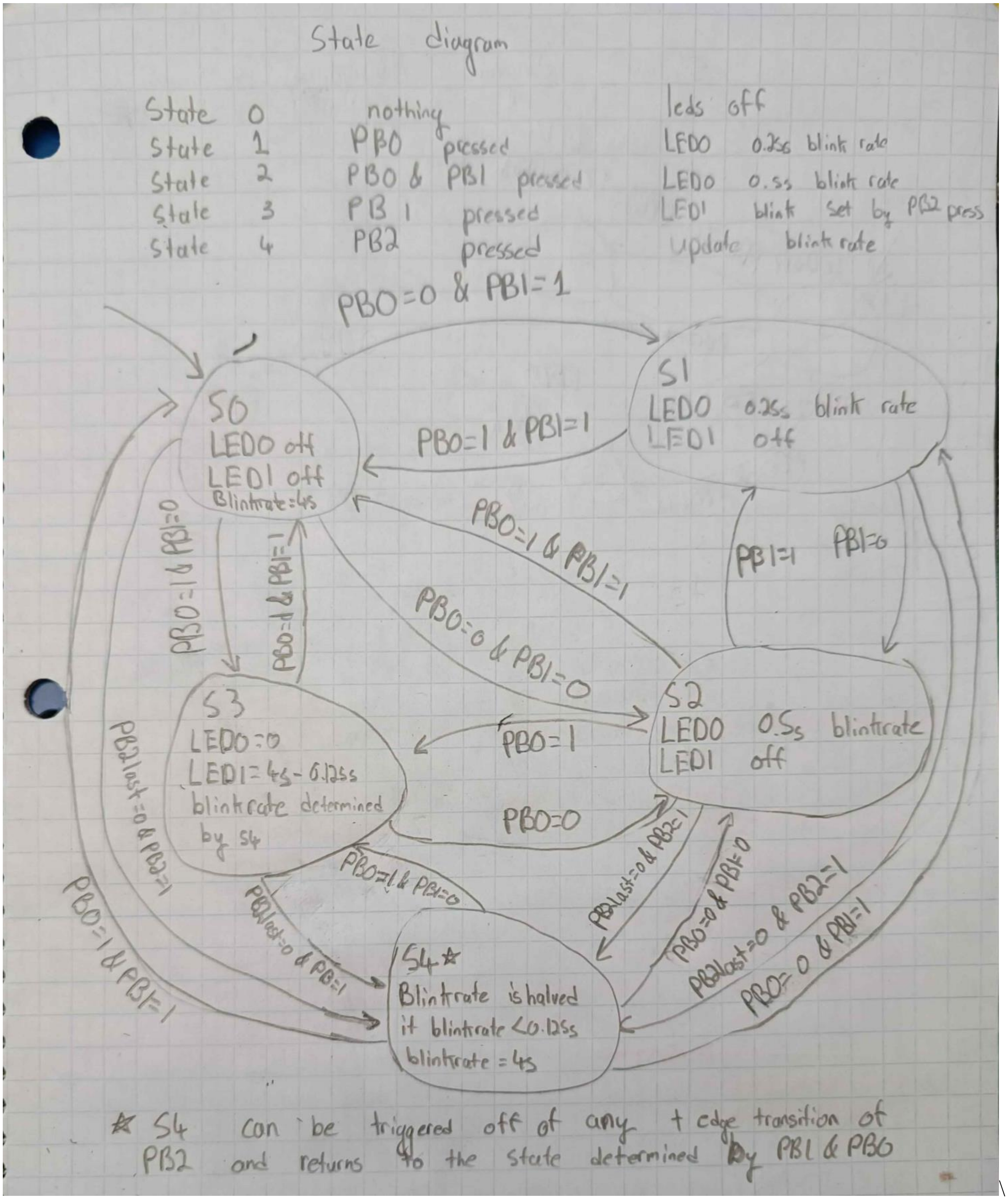


```c
        Idle();

        if (pb_event) {

            delay_ms(DEBOUNCE_TIME);

            if(!pb2_last && PB2) {  // PB2 transition to released
                blink_rate_update();
            }

            // State machine inputs
            if (PB0 == 0 && PB1 == 1) {
                state = 1;
            } else if (PB0 == 0 && PB1 == 0) {
                state = 2;
            } else if (PB0 == 1 && PB1 == 0) {
                state = 3;
            } else {
                state = 0;
            }
```

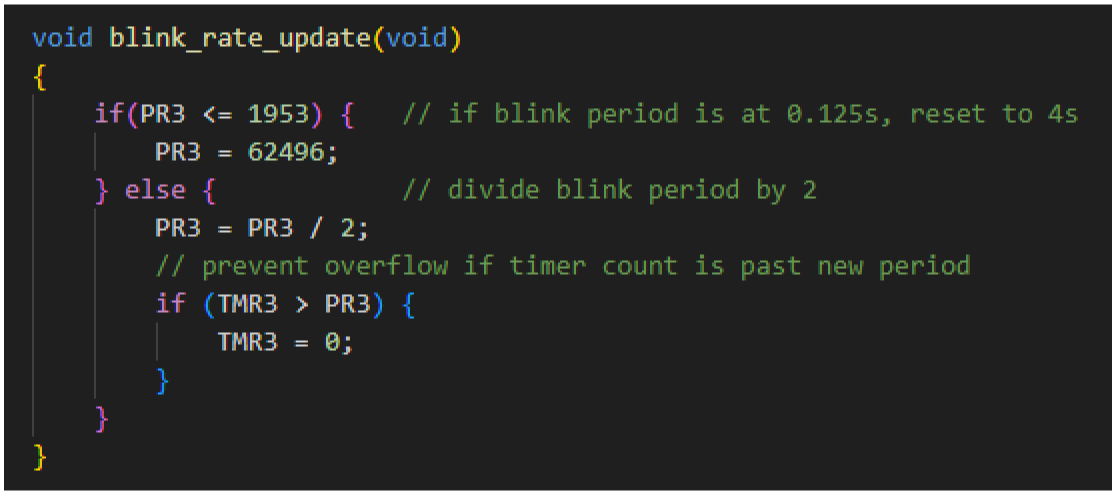
After getting out of idle, the programs checks to see if the pb_event flag was asserted. This is to verify that the reason for exiting idle was a pb event rather than a timer before doing any logic.

After verifying a pb event occurred, a debounce delay is incurred using the delay_ms function, this helps to avoid multiple high and low transitions on a button press.

After the debounce the program reevaluates which state it should be in based on the buttons, this is illustrated in the state diagram below.

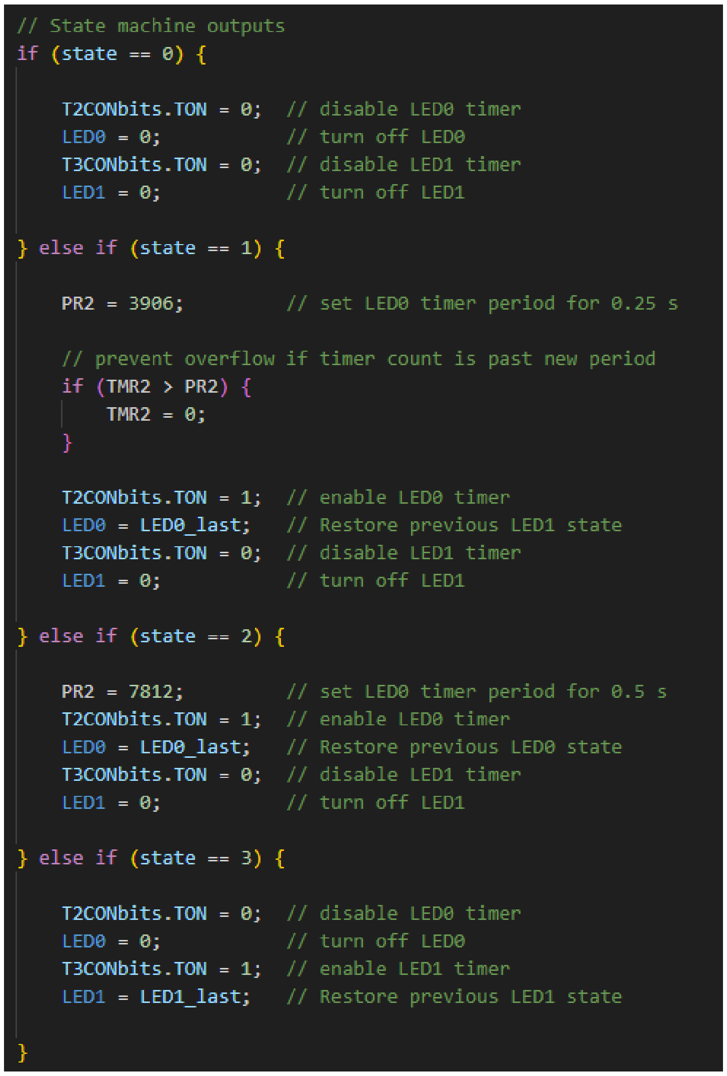


\



```c
void blink_rate_update(void)
{
    if(PR3 <= 1953) {   // if blink period is at 0.125s, reset to 4s
        PR3 = 62496;
    } else {            // divide blink period by 2
        PR3 = PR3 / 2;
        // prevent overflow if timer count is past new period
        if (TMR3 > PR3) {
            TMR3 = 0;
        }
    }
}
```

In the case that the program detects that PB2 has been released, it enters the blink_rate_update function which updates the period of timer 3 which is the blink rate of LED1. First it checks to see if the blink rate is less than or equal to the lowest blink rate of 0.125s, if that's true it resets the blink rate to 4s. Otherwise it simply divides the blink rate by 2. TMR3 is also check to catch the case of it exceeding the value of PR3, if PR3 ends up above the value of TMR3 the timer does not trigger an interrupt until PR3 reaches its makes value and resets before counting to TMR3 again.



```c
// State machine outputs
if (state == 0) {

    T2CONbits.TON = 0;  // disable LED0 timer
    LED0 = 0;           // turn off LED0
    T3CONbits.TON = 0;  // disable LED1 timer
    LED1 = 0;           // turn off LED1

} else if (state == 1) {

    PR2 = 3906;         // set LED0 timer period for 0.25 s

    // prevent overflow if timer count is past new period
    if (TMR2 > PR2) {
        TMR2 = 0;
    }

    T2CONbits.TON = 1;  // enable LED0 timer
    LED0 = LED0_last;   // Restore previous LED1 state
    T3CONbits.TON = 0;  // disable LED1 timer
    LED1 = 0;           // turn off LED1

} else if (state == 2) {

    PR2 = 7812;         // set LED0 timer period for 0.5 s
    T2CONbits.TON = 1;  // enable LED0 timer
    LED0 = LED0_last;   // Restore previous LED0 state
    T3CONbits.TON = 0;  // disable LED1 timer
    LED1 = 0;           // turn off LED1

} else if (state == 3) {

    T2CONbits.TON = 0;  // disable LED0 timer
    LED0 = 0;           // turn off LED0
    T3CONbits.TON = 1;  // enable LED1 timer
    LED1 = LED1_last;   // Restore previous LED1 state

}
```

The program then updates the appropriate timers and LEDs as per the assignment specifications. See state diagram for more details.

```
pb_event = 0;
pb2_last = PB2;
```

Finally, the pb_event is reset so it can wait for the next button input and the state of PB2 is saved in pb2_last so it can be compared in the future to determine in a press-and-release has occurred. The microcontroller then returns to idle.

```
// Timer 2 (LED0) ISR
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void){
    LED0 ^= 1; // toggle LED0
    LED0_last ^= 1;
    IFS0bits.T2IF = 0; // Clear Timer 2 interrupt flag
}

// Timer 3 (LED1) ISR
void __attribute__((interrupt, no_auto_psv)) _T3Interrupt(void){
    LED1 ^= 1; // toggle LED1
    LED1_last ^= 1;
    IFS0bits.T3IF = 0; // Clear Timer 3 interrupt flag
}
```

Idle may now be exited by one of Timer 2 or Timer 3 (depending on the current state) as well as by a button input. Upon a timer interrupt the ISR toggles one of LED0 or LED1 and saves its state in LEDx_last before clearing its interrupt flag and returning to main.

Answers to Lab Questions

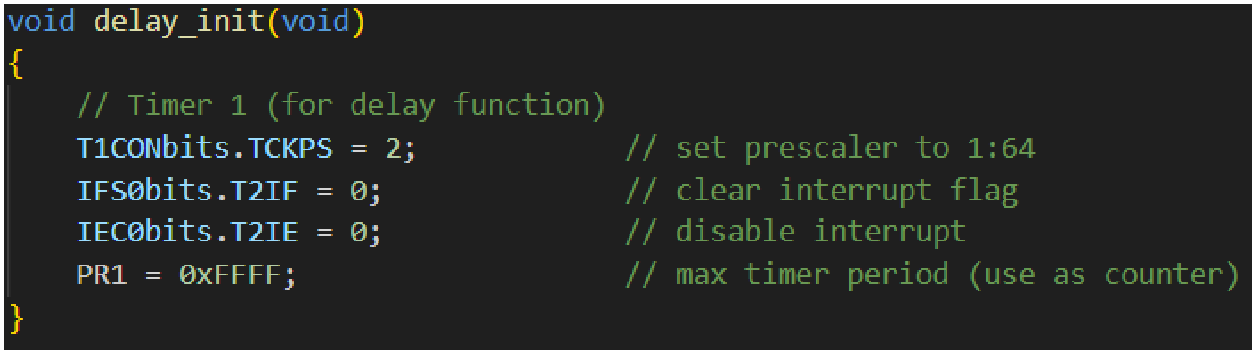
In the original provided program, why does the LED appear to "flash" every time you "click" a button (quickly)?

Because multiple interrupts are occurring as the button "bounces" which causes multiple high-low and low-high transitions.

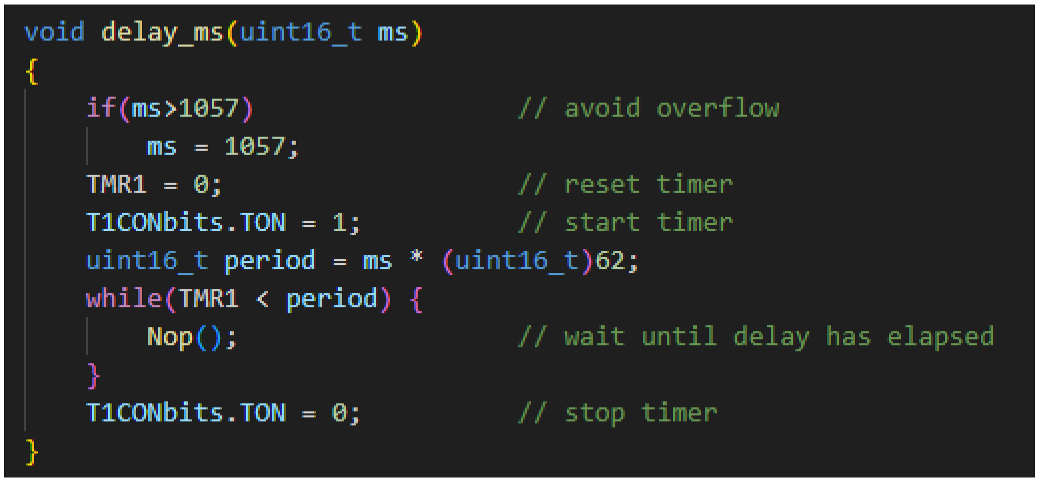
If you press a button and hold it there, does the IOC interrupt service routine keep executing repeatedly? If so, why? If not, why not?

No, the IOC does not repeatedly trigger on a button being held down as it only interrupts on a change (i.e. low to high or high to low transition).

Explain your implementation of void delay_ms(uint16_t ms). How do you guarantee that the delay works across the range of possible input values? What combination of clock/prescaler/PRx values do you use? What is the maximum delay you can use for the FOSC and timer prescaler setting?



```
void delay_init(void)
{
    // Timer 1 (for delay function)
    T1CONbits.TCKPS = 2;            // set prescaler to 1:64
    IFS0bits.T2IF = 0;              // clear interrupt flag
    IEC0bits.T2IE = 0;              // disable interrupt
    PR1 = 0xFFFF;                   // max timer period (use as counter)
}
```



```
void delay_ms(uint16_t ms)
{
    if(ms>1057)                 // avoid overflow
        ms = 1057;
    TMR1 = 0;                   // reset timer
    T1CONbits.TON = 1;          // start timer
    uint16_t period = ms * (uint16_t)62;
    while(TMR1 < period) {
        Nop();                  // wait until delay has elapsed
    }
    T1CONbits.TON = 0;          // stop timer
}
```

The maximum delay for the delay_ms function is 1057ms as any number larger will cause in overflow in the calculation. A prescaler of 64 was used to maximize resolution while still allowing up to 1 sec of delay.

How many timers do you use? What do you use each timer for?

Three timers are used, Timer 1 is used for the delay_ms function (which is used to debounce), Timer 2 and Timer 3 are used for blinking LEDs and specified rates.

Explain your implementation of the IOC interrupt service routine, and your choice of IOC settings. How do you distinguish between a button press and a release? How about the different buttons?

The IOC register is configured to trigger on high and low going transitions on all three buttons, this allows the detection of any change of state in the pressed buttons. To detect a press and release on push button 2 the previous state is stored. Iif the button was pressed in the previous state and is now high it is considered a press and release.