ENCM 511 Assignment 3

Preamble

`common.h` contans the following which have been defined to be used throughout the program:

The LED and buttons have been abbreviated as shown.

```
#define LED0    LATBbits.LATB5
#define PB0     PORTAbits.RA4
#define PB1     PORTBbits.RB8
#define PB2     PORTBbits.RB9
```

The following bit masks have been defined to allow for enhanced readablity. Several variables are used throughout the program to hold multiple flags (bit-fields). These defines allow those flgas to be accessed easily.

```
#define SET_BIT(flags, n)       ((flags) |= (1 << (n)))
#define CHECK_BIT(flags, n)     (((flags) >> (n)) & 1)
#define CLEAR_BIT(flags, n)     ((flags) &= ~(1 << (n)))
#define TOGGLE_BIT(flags, n)    ((flags) ^= (1 << (n)))
```

This is one such bit field. Purpose of each flag elucidated by its comment. Since each flag is either a 0 or 1, we considered it would be a waste to use a separate uint8_t for each flag, as such we used one variable and each position in that variable has a specific button flag which its position is defined in the variable.

```
extern uint8_t pb_stat;      // Bit-field for button status flags
#define PB0_HELD_FLAG    0   // Flag set to indicate PB0 is being held and program has not acted on that yet
#define PB0_CLICKED_FLAG 1   // Flag set to indicate PB0 has been clicked
#define PB1_HELD_FLAG    2   // Flag set to indicate PB1 is being held and program has not acted on that yet
#define PB1_CLICKED_FLAG 3   // Flag set to indicate PB1 has been clicked
#define PB2_HELD_FLAG    4   // Flag set to indicate PB2 is being held and program has not acted on that yet
#define PB2_CLICKED_FLAG 5   // Flag set to indicate PB2 has been clicked
```

Here is an example of this bitfield being updated. In this case the PB0_HELD_FLAG of pb_stat is set to 1.

```
SET_BIT(pb_stat, PB0_HELD_FLAG);
```

The following have been created to simplify checking the flags in pb_stat for commonly used combinations.

```
// Define the transition combination to compare with pb_stat
#define PB0_CLICKED         (1 << PB0_CLICKED_FLAG)
#define PB1_CLICKED         (1 << PB1_CLICKED_FLAG)
#define PB2_CLICKED         (1 << PB2_CLICKED_FLAG)
#define PB0_HELD            (1 << PB0_HELD_FLAG)
#define PB1_HELD            (1 << PB1_HELD_FLAG)
#define PB2_HELD            (1 << PB2_HELD_FLAG)
#define PB0_PB1_HELD       ((1 << PB0_HELD_FLAG) | \
                            (1 << PB1_HELD_FLAG))
#define PB0_PB2_HELD       ((1 << PB0_HELD_FLAG) | \
                            (1 << PB2_HELD_FLAG))
#define PB1_PB2_HELD       ((1 << PB1_HELD_FLAG) | \
                            (1 << PB2_HELD_FLAG))
#define PB0_PB1_PB2_HELD   ((1 << PB0_HELD_FLAG) | \
                            (1 << PB1_HELD_FLAG) | \
                            (1 << PB2_HELD_FLAG))
```

Peripherals

`init_functions.c` contains the initailization functions for the IO pins and the timers. Line functionality elucidated by comments. ISR_PRIORITY is used to prevent ISRs from pre-empting each other.

```
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)

    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
    return;
}
```

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

    // Timer 1 (for LED0)
    T1CONbits.TCKPS = 3;            // set prescaler to 1:256
    IPC0bits.T1IP = ISR_PRIORITY;   // Interrupt priority
    IFS0bits.T1IF = 0;              // clear interrupt flag
    IEC0bits.T1IE = 1;              // enable interrupt
    PR1 = 62496;                    // set period for 4 s

    // Timer 2 (for button timing)
    T2CONbits.TCKPS = 1;            // set prescaler to 1:8
    IPC1bits.T2IP = ISR_PRIORITY;   // Interrupt priority
    IFS0bits.T2IF = 0;              // clear interrupt flag
    IEC0bits.T2IE = 1;              // enable interrupt
    PR2 = 500;                      // set period for 1ms
    T2CONbits.TON = 1;

    // Timer 3 (for delay function)
    T3CONbits.TCKPS = 2;            // set prescaler to 1:64
    IFS0bits.T3IF = 0;              // clear interrupt flag
    IEC0bits.T3IE = 0;              // disable interrupt
    PR3 = 0xFFFF;                   // max timer period (use as counter)

    return;
}
```

`uart.c` contains the initilization for UART2 which is taken from the provided lab files.

```
void InitUART2(void)
{

    RPINR19bits.U2RXR = 11; // Assign U2RX to RP11 (pin 22)
    RPOR5bits.RP10R = 5;    // Assign RP10 (pin 21) to U2TX

    U2MODE = 0b0000000010001000;

    U2BRG = 103;            // Baud rate = 9600

    U2STAbits.UTXISEL0 = 0;
    U2STAbits.UTXISEL1 = 0;
    U2STAbits.URXEN = 1;
    U2STAbits.UTXEN = 1;
    U2STAbits.URXISEL = 0b00;

    IFS1bits.U2TXIF = 0;
    IPC7bits.U2TXIP = 3;

    IEC1bits.U2TXIE = 1;
    IFS1bits.U2RXIF = 0;
    IPC7bits.U2RXIP = 4;
    IEC1bits.U2RXIE = 1;

    U2MODEbits.UARTEN = 1;

    U2STAbits.UTXEN = 1;
    return;
}
```

Datatypes and Initialization

The pb_stat bit-field is initialized to 0.

```
uint8_t pb_stat = 0;    // extern in header, initialized to zero here
```

The states enum is used to control the state machine. Initial and next state are initialized to fast_mode_idle. The state_changed variable is used to indicate when the machine has changed states. The blink setting controls the speed of LED blinking in certain states and is initialized to a char 0.

```
typedef enum
{
    fast_mode_idle,
    fast_mode_PB0,
    fast_mode_PB1,
    fast_mode_PB2,
    fast_mode_PB0_PB1,
    fast_mode_PB0_PB2,
    fast_mode_PB1_PB2,
    prog_mode_idle,
    prog_mode_PB0,
    prog_mode_PB1,
    prog_mode_PB2
} states;


states next_state = fast_mode_idle;
states current_state = fast_mode_idle;

uint8_t state_changed;

char blink_setting = '0';
```

Key Functions

The delay_ms function is held in the **`delay.c`** file and is used to for the debouncing of button inputs.

```
void delay_ms(uint16_t ms)
{
    if(ms>1057)                 // avoid overflow
        ms = 1057;
    TMR3 = 0;                   // reset timer
    T3CONbits.TON = 1;          // start timer
    uint16_t period = ms * (uint16_t)62;
    while(TMR3 < period) {
        Nop();                  // wait until delay has elapsed
    }
    T3CONbits.TON = 0;          // stop timer
}
```

The **`buttons.c`** file contains the buttons_update function. The function uses a bitfield pb_manager_flags as well as three unsigned 16 bit integers to record the number of milliseconds each button has been held for.

```
uint8_t pb_manager_flags = 0;   // extern in header, initialized to zero here
uint16_t pb0_time = 0;  // extern in header, initialized to zero here
uint16_t pb1_time = 0;  // extern in header, initialized to zero here
uint16_t pb2_time = 0;  // extern in header, initialized to zero here
```

The buttons_update function is called periodically (every 1ms) by main when the timer 2 ISR occurs or whenever the button ISR occurs.

Initially it debounces the buttons.

```
void buttons_update(void)
{
    delay_ms(DEBOUNCE_TIME);
```

Next the program evaluates the condition of PB0. This logic is then run for PB1 and PB2. **Note** that we choose to implement a hold to be a press that is longer than 1 second, if you stop holding it down the held flag is then reset to indicate that an active hold is not occurring.

```
// If PB0 becomes pressed, start timing the press
if(CHECK_BIT(pb_manager_flags, PB0_LAST) && !PB0)
    SET_BIT(pb_manager_flags, PB0_ON);

// Once PB0 has been pressed for long enough, set "held" flag if not already set, and stop timing press
if(!PB0 && pb0_time >= HELD_TIME && !CHECK_BIT(pb_stat, PB0_HELD_FLAG)) {
    SET_BIT(pb_stat, PB0_HELD_FLAG);
    CLEAR_BIT(pb_manager_flags, PB0_ON);
    pb0_time = 0;
}

// If PB0 is released:
if(!CHECK_BIT(pb_manager_flags, PB0_LAST) && PB0){
    // If PB0 was pressed only briefly, set "clicked" flag
    if(CHECK_BIT(pb_manager_flags, PB0_ON) && pb0_time < HELD_TIME)
        SET_BIT(pb_stat, PB0_CLICKED_FLAG);
    // Stop timing the press
    CLEAR_BIT(pb_manager_flags, PB0_ON);
    pb0_time = 0;
    // Make sure "held" flag is cleared since button is no longer being pressed
    CLEAR_BIT(pb_stat, PB0_HELD_FLAG);
}
```

After all buttons have been checked their states are updated to be the new “last” flags.

```
    // Update "last" flags
    if(PB0)
        SET_BIT(pb_manager_flags, PB0_LAST);
    else
        CLEAR_BIT(pb_manager_flags, PB0_LAST);
    if(PB1)
        SET_BIT(pb_manager_flags, PB1_LAST);
    else
        CLEAR_BIT(pb_manager_flags, PB1_LAST);
    if(PB2)
        SET_BIT(pb_manager_flags, PB2_LAST);
    else
        CLEAR_BIT(pb_manager_flags, PB2_LAST);
}
```

The buttons_reset function is used to reset the millisecond counter of each button.

```
void buttons_reset(void)
{
    pb0_time = 0;
    pb1_time = 0;
    pb2_time = 0;
}
```

Operation

The code begins by calling all three initialization functions and printing the initial state to the terminal before entering its while loop and going into idle.

```
int main(void) {

    IO_init();
    timer_init();
    InitUART2();

    Disp2String("Fast Mode: IDLE");
    XmitUART2('\r',1);
    XmitUART2('\n',1);

    while(1) {
        Idle();
```

Assuming this is the first run of the program, i.e. the program is in a state where the LED blink timer is not running. The Idle will be interrupted by either a button press or the button manager ISR timer triggering.

```
// Interrupt-on-change ISR
void __attribute__ ((interrupt, no_auto_psv)) _IOCInterrupt(void) {
    SET_BIT(pb_manager_flags, PB_UPDATE);   // flag that the button press logic needs to be run
    IFS1bits.IOCIF = 0; // Clear system-wide IOC flag
}
```

```
// Timer 2 (button manager) ISR
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void){
    SET_BIT(pb_manager_flags, PB_UPDATE);   // flag that the button press logic needs to be run
    if(CHECK_BIT(pb_manager_flags, PB0_ON))
        pb0_time++;
    if(CHECK_BIT(pb_manager_flags, PB1_ON))
        pb1_time++;
    if(CHECK_BIT(pb_manager_flags, PB2_ON))
        pb2_time++;
    IFS0bits.T2IF = 0; // Clear Timer 2 interrupt flag
}
```

If the program is in a state with the LED timer running there may also be an interrupt generated to toggle the LED.

```
// Timer 1 (LED0) ISR
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void){
    LED0 ^= 1; // toggle LED0
    IFS0bits.T1IF = 0; // Clear Timer 2 interrupt flag
}
```

If one of the ISRs set the PB_UPDATE flag then main will call buttons_update to check on the status of the buttons.

```
while(1) {
    Idle();

    if(CHECK_BIT(pb_manager_flags, PB_UPDATE))
    {
        CLEAR_BIT(pb_manager_flags, PB_UPDATE);
        buttons_update();
    }
```

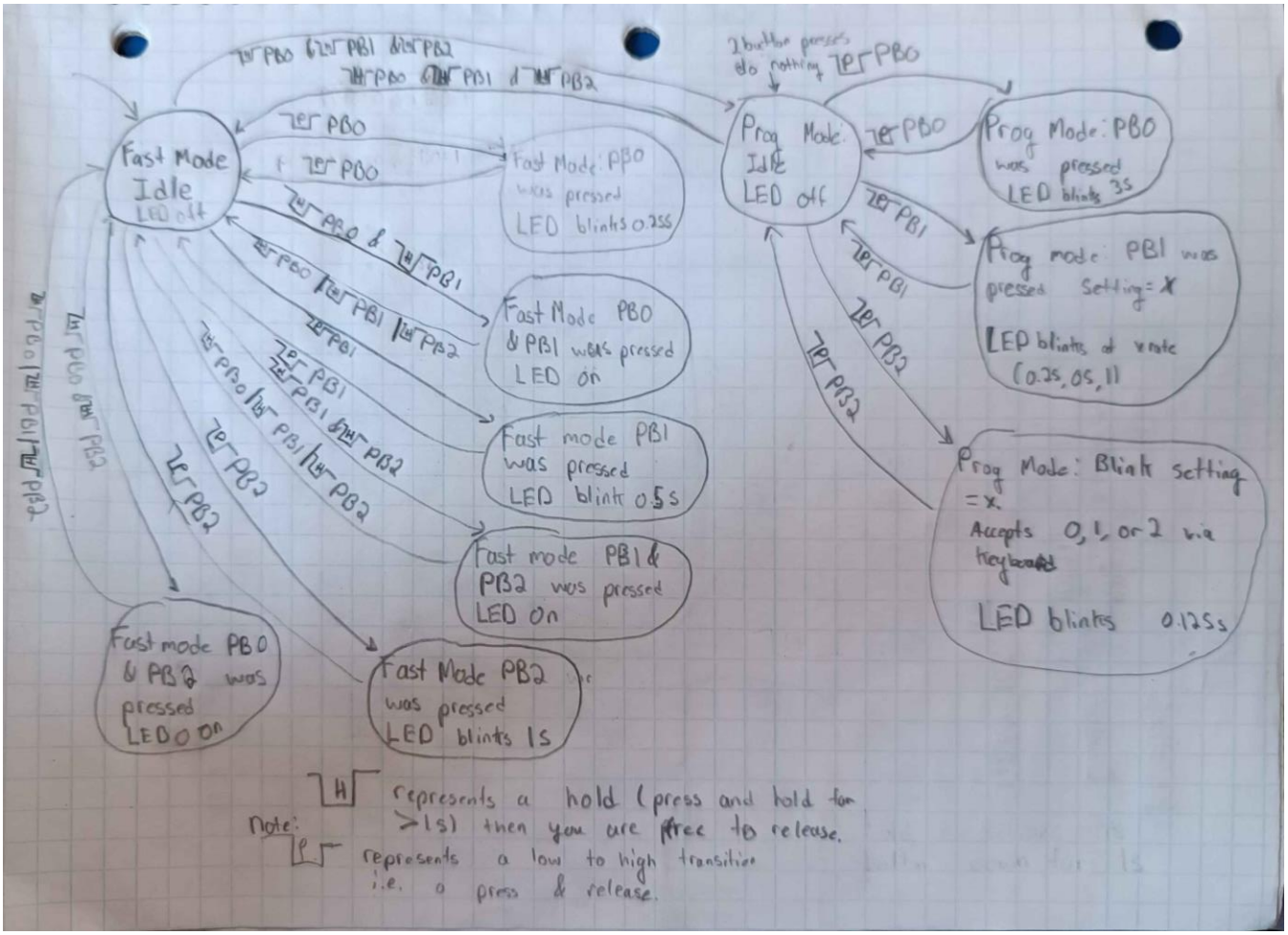
Next, the program checks to see if a button flag was raised by buttons_update, if a flag is set, it will go through the state logic implemented as per the provided state machine drawing. A screenshot is provided to show the state transitions from idle mode. Once in a state which is not idle, the state transitions are simple as only one action can cause a state change back to idle.

```
if(pb_stat)
{
    if(current_state == fast_mode_idle)
    {
        if      (pb_stat == PB0_CLICKED)
            next_state = fast_mode_PB0;
        else if (pb_stat == PB1_CLICKED)
            next_state = fast_mode_PB1;
        else if (pb_stat == PB2_CLICKED)
            next_state = fast_mode_PB2;
        else if (pb_stat == PB0_PB1_PB2_HELD)
            next_state = prog_mode_idle;
        else if (pb_stat == PB0_PB1_HELD)
            next_state = fast_mode_PB0_PB1;
        else if (pb_stat == PB0_PB2_HELD)
            next_state = fast_mode_PB0_PB2;
        else if (pb_stat == PB1_PB2_HELD)
            next_state = fast_mode_PB1_PB2;
        else
            next_state = fast_mode_idle;
    }
    else if(current_state == prog_mode_idle)
    {
        if      (pb_stat == PB0_CLICKED)
            next_state = prog_mode_PB0;
        else if (pb_stat == PB1_CLICKED)
            next_state = prog_mode_PB1;
        else if (pb_stat == PB2_CLICKED)
            next_state = prog_mode_PB2;
        else if (pb_stat == PB0_PB1_PB2_HELD)
            next_state = fast_mode_idle;
        else
            next_state = prog_mode_idle;
    }
    else if(current_state == fast_mode_PB0)
    {
        if(pb_stat & PB0_CLICKED)
            next_state = fast_mode_idle;
        else
            next_state = current_state;
    }
```

For the modes that are entered via a hold, holding any button can get it back to the idle state which is shown below.

```
else if(current_state == fast_mode_PB0_PB2)
{
    if(pb_stat & PB0_PB1_PB2_HELD)  // if any of PBs are held
        next_state = fast_mode_idle;
    else
        next_state = current_state;
}
```

The state machine drawing is provided below.



The next section of code determines if a state transition occurs and raises a flag to switch the case which has been used to implement state functionality. This and the following section are not under the pb_state flag and happens each time the microcontroller goes out of idle.

```
if(current_state != next_state)
    state_changed = 1;

current_state = next_state;
```

If the above code has detected a change of state, it switches the case implemented in the state logic. We used switch statements and each section has its own PR1 value depending on the required blink rate needed. If TMR1>PR1, we reset TMR1 this is done to catch the case in which PR1 is changed from a larger value to a smaller value and TMR has surpassed the new PR1 value in which case the timer will go until it overflows before interrupting.

```
if(state_changed)
{
    switch(current_state)
    {
        case fast_mode_PB0:
            Disp2String("Fast Mode: PB0 was pressed");
            XmitUART2('\r',1);
            XmitUART2('\n',1);
            PR1 = 3906;              // 0.25s blinkrate
            if (TMR1 > PR1)
                TMR1 = 0;
            T1CONbits.TON = 1;
            break;
```

The only case that can directly control the next state is prog_mode_PB2, as after you push PB2 to confirm your setting value, it should go into idle mode. We have implemented the button logic inside a while loop in RecvUartChar012 and thus once that function has exited, the program returns to prog_mode_idle.

```
case prog_mode_PB2:
    T1CONbits.TON = 1;
    PR1 = 1953;              // 0.125s blinkrate
    if (TMR1 > PR1)
        TMR1 = 0;
    Disp2String("Prog Mode: Blink setting = ");
    blink_setting = RecvUartChar012();
    XmitUART2('\r',1);
    XmitUART2('\n',1);
    next_state = prog_mode_idle;
    break;
```

At the end of the case statements, but before breaking from if(state_changed), the pb_state variable and the state_changed variable are reset.

```
state_changed = 0;
pb_stat = 0;
buttons_reset();
```

Below is attached a screenshot of RecvUartChar012 function. This reuses code from the regular button function found in buttons.c, this time only looking for a click to exit (PB2 click). Since the user has already clicked PB2 the next case should be set prog_mode_idle which is done in the case statement for prog_mode_pb2.

```
char RecvUartChar012()
{
    pb_stat = 0;
    buttons_reset();
    char last_char;
    XmitUART2(' ',1);
    // wait for enter key
    while (!(pb_stat & PB2_CLICKED)) {

        // If PB2 becomes pressed, start timing the press
        if(CHECK_BIT(pb_manager_flags, PB2_LAST) && !PB2)
            SET_BIT(pb_manager_flags, PB2_ON);

        // If PB2 is released:
        if(!CHECK_BIT(pb_manager_flags, PB2_LAST) && PB2){
            // If PB2 was pressed only briefly, set "clicked" flag
            if(CHECK_BIT(pb_manager_flags, PB2_ON) && pb2_time < HELD_TIME && pb2_time > 40)
                SET_BIT(pb_stat, PB2_CLICKED_FLAG);
            // Stop timing the press
            CLEAR_BIT(pb_manager_flags, PB2_ON);
            pb2_time = 0;
        }

        if(PB2)
            SET_BIT(pb_manager_flags, PB2_LAST);
        else
            CLEAR_BIT(pb_manager_flags, PB2_LAST);

        if (RXFlag == 1) {

            // only store chars '0', '1', '2'
            if (received_char >= 48 && received_char <= 50) {
                XmitUART2(0x08,1); // send backspace
                last_char = received_char;
                XmitUART2(received_char,1); // loop back display
            }
            U2STAbits.OERR = 0;
            RXFlag = 0;
        }
    }
    return last_char;
}
```

Answers to Lab Questions

In part 1 (the lab portion), what was the target baud rates? What is the % error of the actual baud rates?

The target baud rate was the UART standard 9600. The actual baud rate (with U2BRG = 103 and BRGH = 1) was:

$$Baud\ Rate = \frac{4MHz}{4 * (103 + 1)} = 9615$$

The percent error is then:

$$Percent\ Error = \frac{|9615 - 9600|}{9600} = 0.16\%$$

What are your comments/explanations for XmitUART2?

```
void XmitUART2(char CharNum, unsigned int repeatNo)
{
    U2STAbits.UTXEN = 1;
    while(repeatNo!=0)
    {
        while(U2STAbits.UTXBF==1)   // wait while uart transmit buffer is full
        {
        }
        U2TXREG=CharNum;            // load char into 8bit transmit buffer
        repeatNo--;
    }
    while(U2STAbits.TRMT==0)        // wait for transmit shift register to finish to avoid disable mid message
    {
    }

    U2STAbits.UTXEN = 0;
}
```

In your assignment part, how do you get the LED to blink while waiting for a character to be input? Is the character receive function a "busy wait," i.e., does it stop/block the CPU from doing something else?

The character receive function has a while that is constantly polling if an enter key has been received, inside that while loop there is function that only attempts to read a character when the data in the Rx buffer interrupts occurs that there is, otherwise it is not doing anything. Therefore, this is considered a busy wait as its constantly polling if the received char is an enter and is blocking other sections of code from occurring.