

Part I: (72 pts)

- a. (5 pts) Write C code that configures PORTB for the IO shown in the figure for problem (a) on the Figure sheet. The internal weak pullup must be enabled. Do not assume any default bit values.

```
//one solution  
RBPB = 0;  
TRISB0 = 1; //input  
TRISB6 = 1; //input  
TRISB5 = 0; /input
```

- b. (15 pts) Assuming the IO configuration of the previous problem, write a *while(1){}* loop that implements the LED/Switch IO state machine shown for problem (b) in the figures. Either use a *switch()* statement approach or a *if-then-else* approach. Assume you have available the *DelayMs()* function for blinking the LED; you do NOT have to include debounce delays for the switch input.

```
char state;  
main(){  
  //configure ports, not shown, debounce is not done  
  state = START;  
  while (1){  
    switch(state) {  
      case  
      case START:  
        RB5 = 0; //turn off  
        while(RB0); while(!RB0); //wait for press & release  
        if (RB6) state = TGL; else state = BLINK;  
        break;  
      case TGL:  
        RB5 = 1 ;  
        while(RB0); while(!RB0); //wait for press & release  
        state = START;  
        break;  
      case BLINK  
        while (RB0) { // while not pressed  
          if (LB5) LB5 = 0; else LB5 = 1; //blink  
          DelayMs(200); //blink delay  
        }  
        state = ON; //pressed, go to next state  
        break;  
      case ON:  
        RB5 = 1; // turn on  
        while (!RB0); //stay here while pressed  
        state = START;  
        break;  
    }//end switch  
  }//end while
```

- c. ( 20 pts) For the LED/Switch configuration shown in Figure (a), implement the flowchart of figure (c) in an *interrupt driven manner*. Divide your solution into two code segments -- an ISR, and *main()* code that includes the declarations of any variables used by the ISR and *main()*, initialization code for the interrupt system, and initializes the LED to OFF. You do not have to debounce the input switch. You cannot have any delay code in the ISR to that waits for input. Your infinite loop in *main()* has to be an infinite loop that is an EMPTY infinite loop *while(1){}*; your ISR has to do all of the work.
1. (13 pts) ISR code (do not worry about debouncing the switch inputs).

```

interrupt my_isr() {
  if (INT0IF) {
    INT0IF = 0; //clear flag
    switch (state) {
      case START: //switch has been pressed
        RB5 = 1; //seen falling edge of press, turn on LED
        INTEDG0 = 1; //change edge to rising edge for release
        state = ON;
        break;
      case ON: //switch has been released
        count = 0;
        //go ahead and toggle LED once
        if (LB5) RB5 = 0; else RB5 = 1;
        state = TGL;
        break;
      case TGL: //press and release has occurred
        count++;
        if (count < 10) {
          if (LB5) RB5 = 0; else RB5 = 1; //toggle LED
        } else {
          state = START;
          RB5 = 0; //turn off
          INTEDG0 = 0; //select falling edge
        }
        break;
      //end switch
    } //end INT0IF
  } end my_isr

```

The key to understanding this code is that the ISR state code is executed AFTER THE ACTIVE EDGE occurs.

This means that you are writing code in a state for events that are in the next state of the ASM chart since the code is executed after the edge occurs.

There are no 'while(RB0)' or 'while(!RB0)' tests in the ISR because an ISR does not wait for input; it is triggered when the Input (active edge) occurs. You also don't test the value of RB0 in the ISR because you know what has happened – an active edge has been seen.

2. (7 pts) *main()* code, your *while(1){}* has to be empty; the ISR must do all of the work.

```

main() {
  RBPU = 0; TRISB0 = 1; TRISB6 = 1; TRISB5 = 0;
  state = START;
  INT0IF = 0;
  INTEDG0 = 0; //falling edge
  IPEN = 0; INT0IE = 1; GIE = 1; //enable INT0 interrupt
  while(1); //infinite while(1) loop does nothing, ISR does work.
}

```

- d. (7 pts) Assume an asynchronous serial channel with a data format of 1 start bit, 8 data bits, and 1 stop bit between characters. If I wanted to guarantee that eight characters would be transmitted in 3 ms, what is the MINIMUM baud rate I could use from the standard baud rates of 4800, 9600, 19200, 38400, 57600, 76800 or 115200? You must show your WORK in order to get any credit for this problem. Assume the receiver accepts data as fast as I transmit it.

```
8 chars * 10 bits/char * bit_time < 3 ms  
8 chars * 10 bits/char * 1/baud_rate < 3 ms  
( 8 chars * 10 bits/char ) /3 ms < Baud Rate  
Baud Rate > 80/(0.003) > 26667
```

**So baud rate must be at least 38400 from list above.**

- e. (7 pts) Write C code that implements the *void putch(char c)* function (transmits one character to the serial port). *No interrupts are enabled.*

```
void putch (char c)  
{  
    while (!TXIF); //wait for TX to be ready  
    TXREG = c; //send character  
}
```

- f. (9 pts) Write a C code function that waits for availability of data in a circular buffer, then reads data out the circular buffer and returns it. Assume the variables declared below; the head pointer is used to place data into the buffer, the tail pointer is used to take data out of the buffer.

```
#define BUFSIZE 16  
char head, tail;  
char buf[BUFSIZE] // implements storage for circular buffer
```

```
char cbuff_read() {  
    while(head == tail) {}; //wait for data  
    tail++;  
    if (tail == BUFSIZE) tail = 0; //wrap the pointer  
    return(buf[tail]); //return the data  
}
```

g. (9 pts) Given the code below and Figure (g), answer the questions:

g.1 This produces a periodic waveform; draw it starting at time 0 and continue for a couple of cycles.

**The falling edge on RB1 caused by RB5 going low in the while(1) loop triggers the ISR, which sets RB5 back high again. After the ISR exits, execution is picked back up in the while(1) loop, causing RB1 to go low again, re-triggering the interrupt. The code bounces back and forth between the while(1) loop and the ISR.**

g.2 What determines the HIGH PULSE WIDTH time?

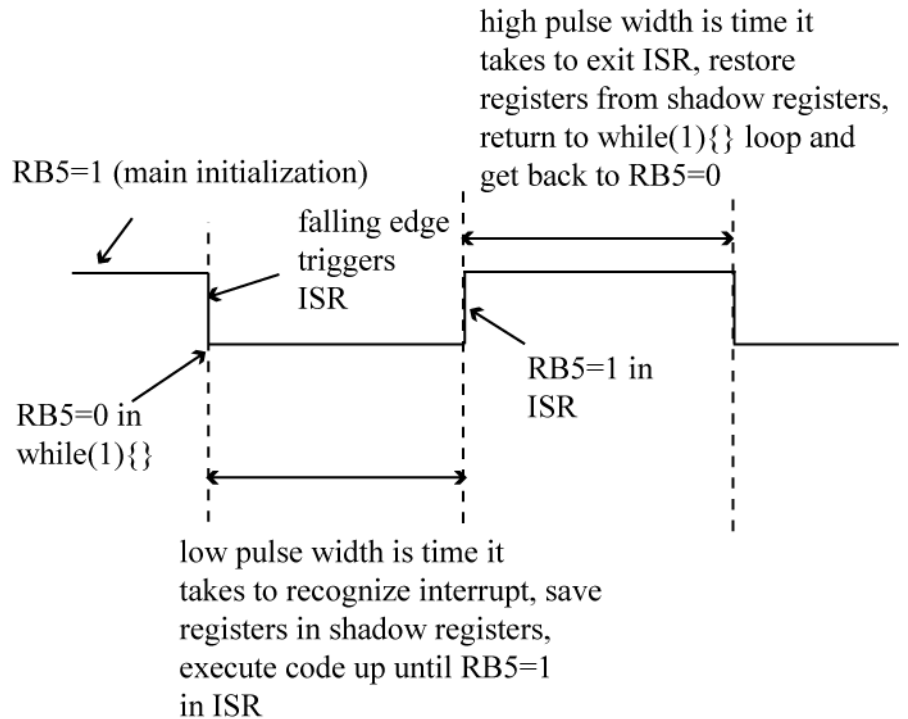
g.3 What determines the LOW PULSE WIDTH time?

```

interrupt my_isr() {
    if (INT1IF) {
        INT1IF = 0;
        RB5 = 1;
    }
} //end my_isr()

main() {
    TRISB5 = 0; TRISB1 = 1;
    RB5 = 1;
    INTEDG1 = 0; IPEN = 0; INT1IF = 0; INT1IE = 1; GIE = 1;
    while (1) {
        RB5 = 0;
    } // end while()
} //end main()

```



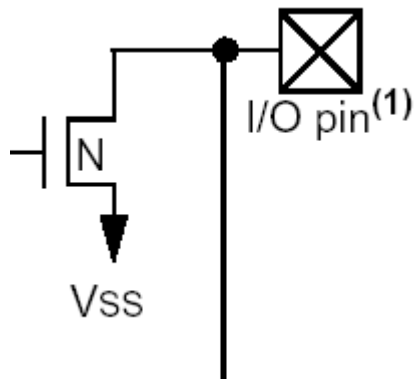
Part II: (28 pts) Answer 7 out of the next 9 questions. Cross out the 2 questions that you do not want graded. Each question is worth 4 pts.

1. Given a voltage of 3 V, a clock freq of 10 MHz, and a current consumption of 5 mA, what is the new current consumption predicted by theory if the voltage is increased to 4 V and the clock frequency to 20 MHz?

$$5 \text{ mA} (4 * 4 * 20) / (3 * 3 * 10) = 17.8 \text{ mA}$$

voltage and current increased, so current draw increases

2. Draw a picture that shows how an open-drain output differs from a normal CMOS output.



open drain output lacks a P pullup to VDD.

3. In the code below, a student is trying to detect a power-on-reset, but a mistake has been made. Correct the code.

```
main() {  
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz  
    if(!POR) {  
        printf("Power-on-reset detected!");pcrlf();  
        POR = 1; //MUST SET POR to 1 to avoid false detection later  
    }  
    //rest of code
```

4. What is the SPBRG value for a baud rate of 38400 assuming an FOSC of 15 MHz and high speed mode?

**SPBRG = 15 MHz / (16 \* 38400) - 1 = 23.4, round to 23**

5. In the code below, what will happen assuming the standard PIC18 setup that you have been using in lab?

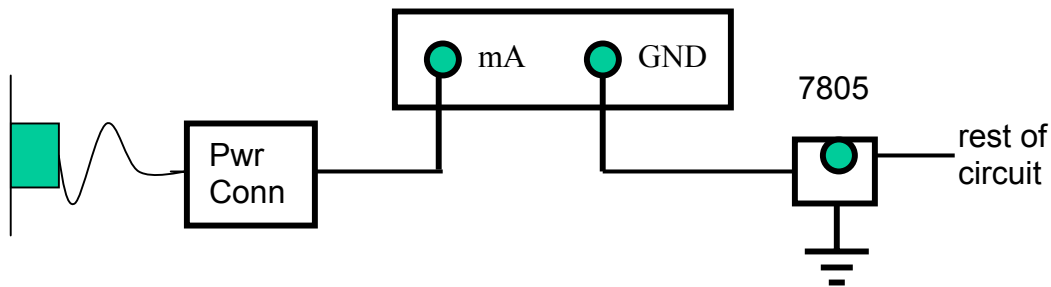
```
main() {  
  
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz  
    printf("Daylight Savings Time is Evil!");pcrlf();  
    SWDTEN = 1;  
    while (1) {  
        // do nothing  
    } //end while()  
} //end main()
```

The message about 'daylight..' is printed, and the WDT is enabled by SWDTEN=1 . The while(1) loop is entered; after some time, the WDT expires, resetting the PIC. The main() code is re-executed, and it all happens again. So you continually see the message printed to the screen.

6. Can I hook the PIC TX and RX pins directly to pins 2, 3 of the DB9 to implement my serial port connection to the PC? If NO, why not? What is needed?

No, RS232 levels are -3V to -25V for logic 1, and +3V to +25V for logic 0. You need an interface IC like the MAX232 to convert from CMOS levels to RS232 levels and vice-versa.

7. Draw a diagram that shows how to measure the TOTAL current flowing into your PIC18F242 breadboard setup, including the current used by the voltage regulator.



8. For the 7-bit value 0x4E, what is the value of the parity bit if EVEN parity is used. What types of errors is a parity bit guaranteed to detect? A change in one bit value? A change in two bit values? A change in more than two bits?

The value 0x4E = 1001110, has an even number of '1' bits already so an even parity bit is '0' for this value.

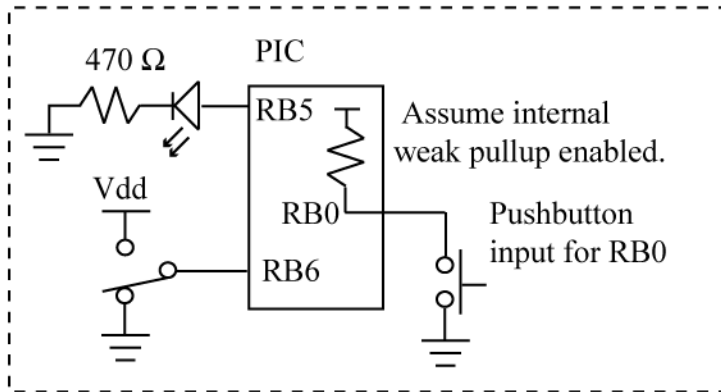
A parity bit is guaranteed to detect a change in a single bit value. It may or may not detect a change in multiple bits depending on how the bits are changed.

9. The PIC18 automatically saves the registers BSR, W, and STATUS into the shadow registers when an interrupt occurs for either a low or high priority interrupts. Assume that you had a system with both low and high priority interrupts. What is the first thing that your low priority interrupt service routine should do with the values saved in the BSR, W, STATUS shadow registers? Explain why. Is this also necessary for the high priority ISR? Why or Why not?

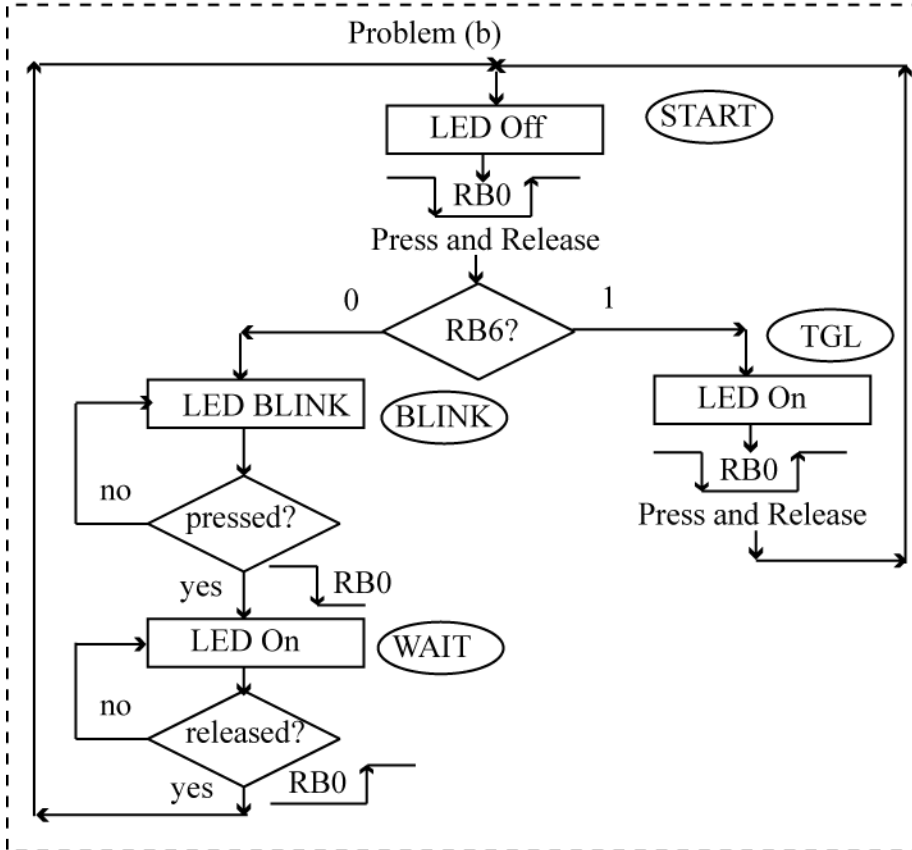
The low priority ISR should immediately disable the high priority interrupts (GIE=0), and save the BSR, W, and STATUS in some temporary locations. After this, the low priority ISR can enable the high priority interrupt and continue execution. If a high priority interrupt occurs, the shadow registers are used to save the BSR/W/Status registers. This is not needed by the high priority interrupt because the high priority ISR will not be interrupted.

Figures

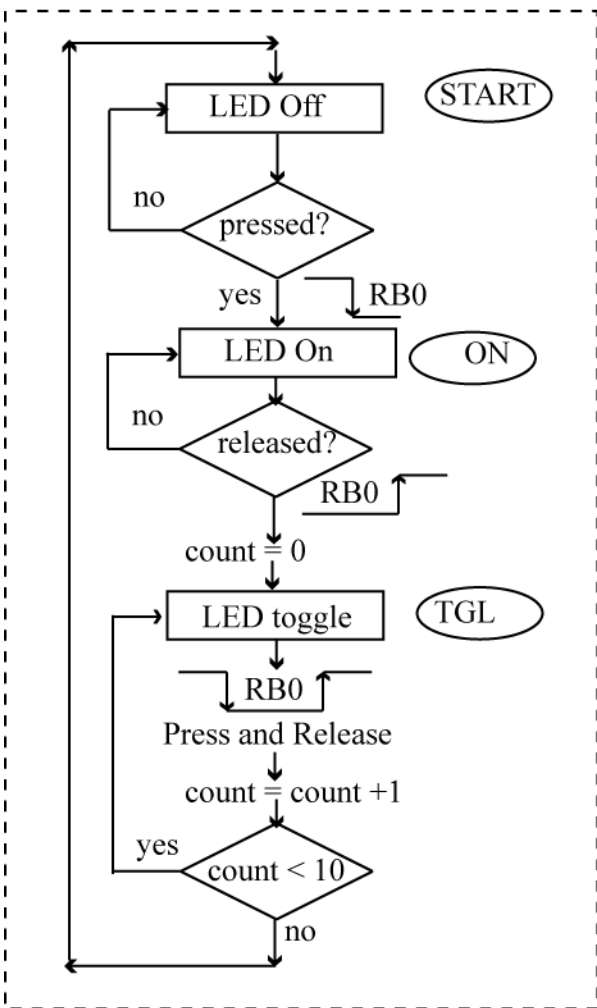
Problem (a)



Problem (b)



Problem (c)



LED toggle means that if it is ON, then turn it OFF, and vice-versa

Problem (g)

