Laboratory 13

Programming a PIC Microcontroller –Part II. *(modified from lab text by Alciatore)*

Required Components:

- 1x PIC16F88 microcontroller
- 1x 0.1 μF capacitor
- 3x NO button switches
- 4x lkΩ resistor
- 1x 7 segment display
- 1x 330 Ω resistor

Required Special Equipment and Software:

- MPLab X, microchip technology's IDE
- XC8, opensource compiler for PICs
- PICkit 2 software
- CanaKit USB PIC Programmer
- Demonstration hexadecimal counter circuit board with hardware switch debouncer (555 timer circuit and a D flip-flop latch IC).

Prelab

See steps 2 and 3 of Procedure section.

Objective

This laboratory exercise builds upon the introduction to the PIC microcontroller started in the previous Lab. Here, input polling is introduced as an alternative to interrupts. You will learn how to configure and control the inputs and outputs of the PIC using the TRIS registers. You will also learn how to perform logic in your programs. You will first observe and describe the operation of a pre-built hexadecimal counter project provided by me. You will then create and test an alternative design using different hardware and software.

Demonstration Circuit for hexadecimal counting using polling

The first part of the laboratory involves the demonstration of an existing counting circuit using a PIC to activate the 7 segments of a digital LED display**. You will not be building a circuit or writing code for this demonstration design**; although, the alternative design you will implement has some similarities. At power-up of the demo circuit, a zero is displayed, and three separate buttons are used to increment by one, decrement by one, or reset the display to zero. The display is hexadecimal so the displayed count can vary from 0 to F. An input monitoring technique called polling is used in this example. With polling, the program includes a loop that continually checks the values of specific inputs. The output display is updated based on the values of these inputs. Polling is different from interrupts in that all processing takes place within the main program loop, and there is not a separate interrupt service routine. Polling has a disadvantage that if the program loop takes a long time to execute (e.g., if it performs complex control calculations), changes in the input values may be missed. The main advantage of polling is that it is very easy to program.

TRIS Registers

At power-up, all bits of PORTA and PORTB are initialized as inputs. However, in this example we require 7 output pins. Here, pins to be used as outputs are designated using special registers called the TRISA and TRISB. These registers let us define each individual bit of the PORT as an input or an output. Setting a TRIS register bit to 0 designates an output and setting the bit to 1 designates an input. For example,

TRISA = 0b00000000;

designates all bits of PORTA as outputs and

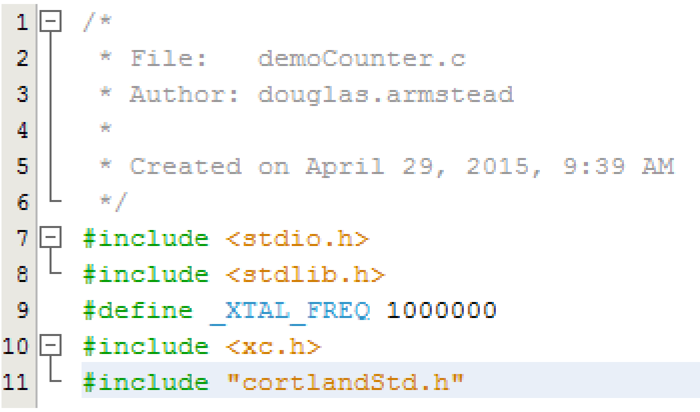
TRISB = 0b01110000;

designates bits 4, 5, and 6 of PORTB as inputs and the others as outputs. Note that by default PORTA has only 5 usable bits (bits 0 through 4 are available, 5 is typically a memory clear pin, 6 and 7 are typically external clock pins), the three most significant bits of PORTA are ignored and have no effect. At power-up all TRIS register bits are set to 1, so all pins are treated as inputs by default (i.e., TRISA=0xFF and TRISB=0xFF).

demoCounter.C Code

The code "demoCounter.c" for the up/down hex counter in listed in three pieces below. NOTE - You will need to modify this code, and the circuit, for this Laboratory. See Lab 12 for more information.

The program has pre-processor directives (e.g., #DEFINE and #include), note that you will need cortlandStd.h, (see lab website) followed by the updatePins(int) function.



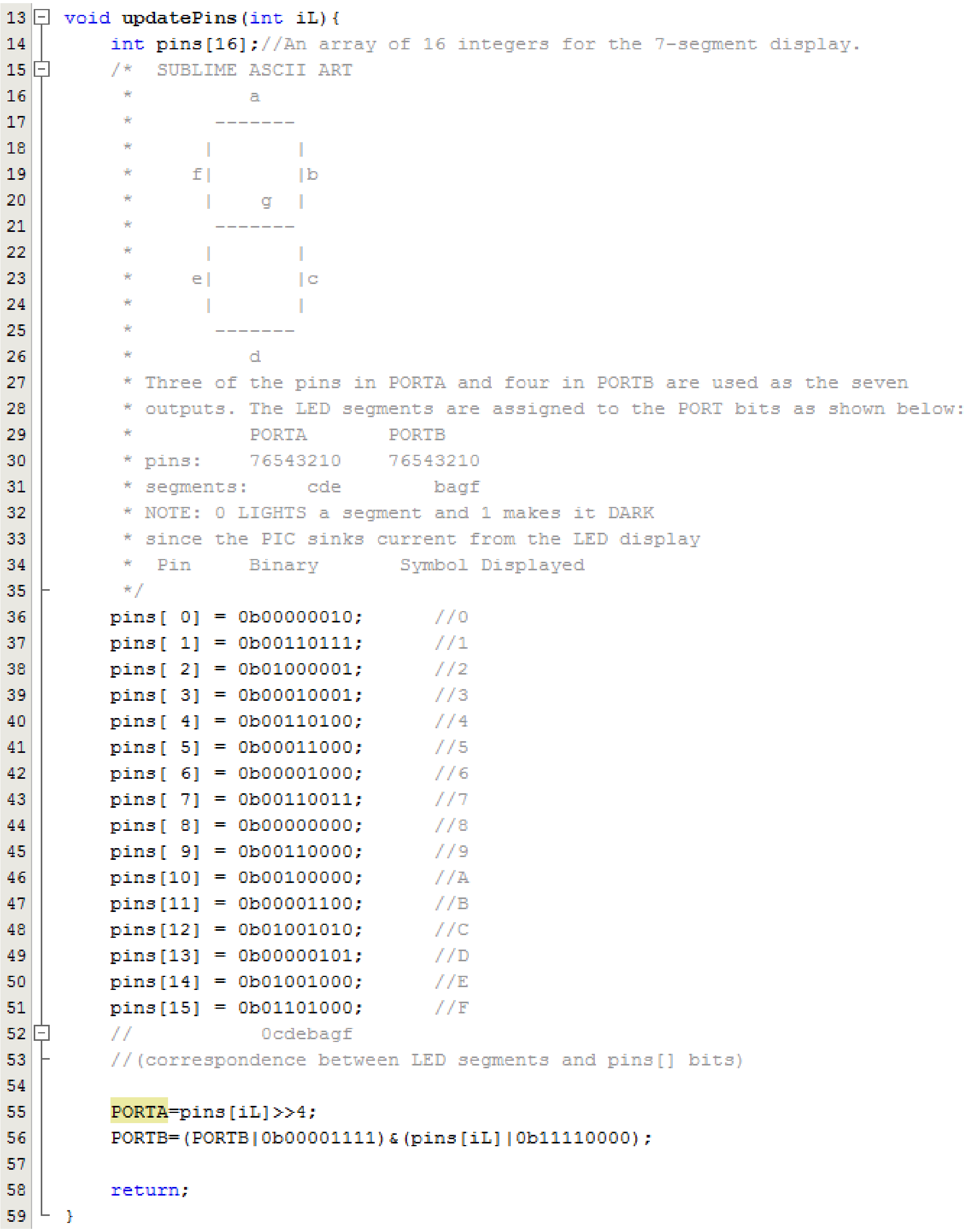
```c
/*
 * File:   demoCounter.c
 * Author: douglas.armstead
 *
 * Created on April 29, 2015, 9:39 AM
 */
#include <stdio.h>
#include <stdlib.h>
#define _XTAL_FREQ 1000000
#include <xc.h>
#include "cortlandStd.h"
```

updatePins(i) Function

This function sets the output pins to control the LED based on the integer it receives. It returns nothing to main() and so I have given it type void. The only variable in this function is pins. It has type integer and is an array which holds 16 separate numbers It maps out what value the pin s must have to display each symbol. The array entries are labeled from 0 to 15. The storage slots are accessed using pins[i], (e.g., pins[7] accesses the 8th integer in pins).

The 8 bits in each integer are grouped into 2 sets of 4 bits: the left 4 bits (most significant bits: MSB's) assign values to the pins set by PORTA, and the right 4 bits (least significant bits: LSB's) assign values to the pins set by PORTB. These depend on the function of the circuit attached to the PIC. Bits 4,5, and 6 of the pins[i] variable are output through PORTA pins to segments e, d, and c of the display, respectively. Bits 0, 1,2, and 3 of the pins[i] variable are output through PORTB pins to segments f, g, a, and b of the display, respectively. Note that bit 7 is set to 0 for each element in the pins[i] variable. Also, note that bit values assume negative logic where a 0 turns the segment on and a 1 turns the segment off.

Now for the most direct action in displaying the output. A simple assignment statement (a statement containing an equal sign =) performs the write to the pins that change the segments in the display. Recall that only three of the five available output pins are used on PORTA (the LSB's of PORTA). Since the pins[i] variable stores values for PORTA in bits 4 through 7 (the MSB's of pins[i]), these bits must be extracted and shifted. The right shift operator (») is used to shift the four MSB's four places to the right to become the four LSB bits 0 through 3. The four MSB's are replaced with 0's as a result of the shift. The result is written to PORTA by the assignment statement. Output to PORTB is more complex since the procedure seeks to retain the existing values for bits 4 through 7 (the MSB's of PORTB) since these bits are reserved for the button inputs, and yet changing bits 0 through 3 (the LSB's of PORTB). Boolean logic operators for OR (|) and AND (&) are used to carry out this process. The OR in the left set of parentheses maintains the four MSB's of PORTB, and sets the four LSB's to 1. The OR in the right set of parentheses maintains the four LSB's of the pins[I] variable, and sets the four MSB's to 1. When the two results are ANDed, the four MSB's of PORTB are retained, while the four LSB's are changed to the values found in the currently indexed pins[i] element. The assignment to PORTB effectively writes the LSB's to the pins, which turns the respective segments on or off. The display is updated every time a call is made to the updatePins(i) function.



```
void updatePins(int iL){
    int pins[16];//An array of 16 integers for the 7-segment display.
    /*  SUBLIME ASCII ART
     *          a
     *       -------
     *      |       |
     *     f|       |b
     *      |    g  |
     *       -------
     *      |       |
     *     e|       |c
     *      |       |
     *       -------
     *          d
     * Three of the pins in PORTA and four in PORTB are used as the seven
     * outputs. The LED segments are assigned to the PORT bits as shown below:
     *          PORTA       PORTB
     * pins:    76543210    76543210
     * segments:     cde        bagf
     * NOTE: 0 LIGHTS a segment and 1 makes it DARK
     * since the PIC sinks current from the LED display
     *  Pin    Binary       Symbol Displayed
     */
    pins[ 0] = 0b00000010;      //0
    pins[ 1] = 0b00110111;      //1
    pins[ 2] = 0b01000001;      //2
    pins[ 3] = 0b00010001;      //3
    pins[ 4] = 0b00110100;      //4
    pins[ 5] = 0b00011000;      //5
    pins[ 6] = 0b00001000;      //6
    pins[ 7] = 0b00110011;      //7
    pins[ 8] = 0b00000000;      //8
    pins[ 9] = 0b00110000;      //9
    pins[10] = 0b00100000;      //A
    pins[11] = 0b00001100;      //B
    pins[12] = 0b01001010;      //C
    pins[13] = 0b00000101;      //D
    pins[14] = 0b01001000;      //E
    pins[15] = 0b01101000;      //F
    //           0cdebagf
    //(correspondence between LED segments and pins[] bits)

    PORTA=pins[iL]>>4;
    PORTB=(PORTB|0b00001111)&(pins[iL]|0b11110000);

    return;
}
```

main() Function

The main() function is where all of the control in the program resides and the registers are set. The TRIS registers are set to determine the I/O status of the pins in PORTA and PORTB. Since all bits in TRISA are 0, all pins corresponding to PORTA are set as outputs. Note that PORTA bits 5, 6, and 7 have no function since no pins are configured on the PIC to correspond to these values. The TRISB register value is set so that PORTB bits 4, 5, and 6 are inputs (each of these 3 bits is set to 1), while the other 5 pins of PORTB are set as outputs. Each of these three input pins for PORTB is attached to a separate button: Depending upon which button is pressed, the counter will either increment by one, decrement by one, or reset to zero using the hexadecimal counting sequence.

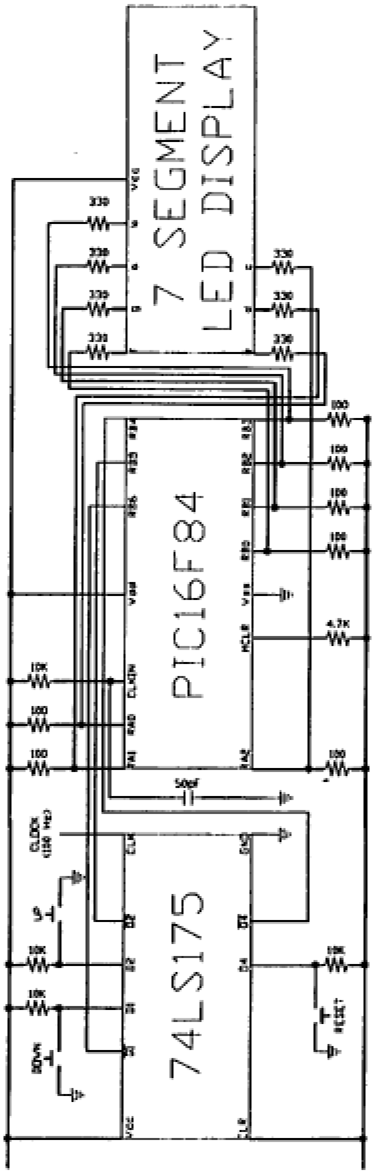
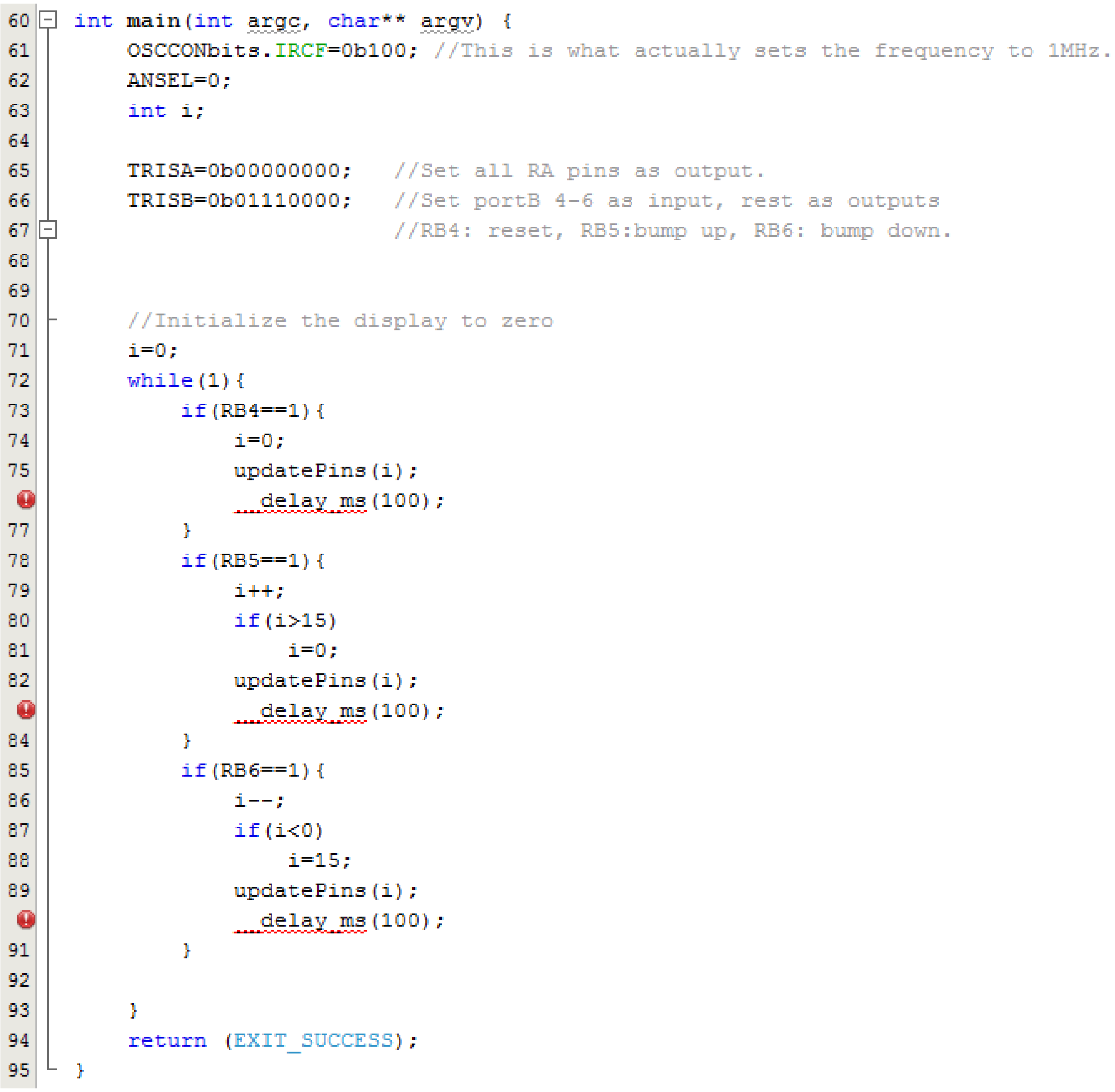


Figure 1 Circuit diagram for the Demonstration-Only hexadecimal counter. *NOTE - Do not build this circuit. You will build an alternative design instead* (see below).

The polling loop used to check for button input is in the "Main loop" of the program.

- The first IF statement checks whether the button attached to PORTB.4 is down. If it is, the variable i is set to 0 so that the display will be zero. The updatePins(i) routine is called to update the value displayed as described in detail already. Then a pause occurs for 100 milliseconds (0.1 sec).
- The polling then continues by checking PORTB.5, and if the value is high, then the hexadecimal count is incremented by incrementing the variable i. The internally nested IF statement then checks if i exceeds the allowed value of 15, and if it does i is reset to 0. The display effectively will count from 0 through 15 as the button is repeatedly depressed or held down, but will cycle back to 0 after an F has been displayed (the highest digit value in hexadecimal). Again a call to updatePins(i) updates the display, and a 0.1 second pause occurs. The pause prevents the count from updating too quickly while the button is being held down before it is released. If the button is held down for more than 0.1 second the counter will increment every 0.1 second.
- Then PORTB.6 is checked and a value of 1 will cause a decrement in i. Once the display reaches a minimum value of zero, the routine will cycle back to F for the next hexadecimal value.

As shown in Figure 1, latching of button values is accomplished by the use of D flip-flops (74LS175) in the circuit. Also key to the circuit, but not shown in the figure, is the use of a 555 timer circuit to create a 100 Hz clock signal. On each positive edge of clock pulse, the current states of the buttons are stored (latched) in the D flip-flips on the 74LS175 IC. Together, the timer and flip-flops perform a hardware debounce. The latched values are read by the PIC each time the program passes through the polling loop. Note that the buttons are shown wired in the figure with negative logic, where the button signal is normally high and goes low when it is pressed. The software above assumes positive logic (with the aide of the Q outputs on the D flip-flops) instead where the button signal is normally low and goes high when pressed. This is easily accomplished by using a pull-down resistor to ground instead of a pull-up resistor to 5 V.



```
int main(int argc, char** argv) {
    OSCCONbits.IRCF=0b100; //This is what actually sets the frequency to 1MHz.
    ANSEL=0;
    int i;

    TRISA=0b00000000;   //Set all RA pins as output.
    TRISB=0b01110000;   //Set portB 4-6 as input, rest as outputs
                        //RB4: reset, RB5:bump up, RB6: bump down.


    //Initialize the display to zero
    i=0;
    while(1){
        if(RB4==1){
            i=0;
            updatePins(i);
            __delay_ms(100);
        }
        if(RB5==1){
            i++;
            if(i>15)
                i=0;
            updatePins(i);
            __delay_ms(100);
        }
        if(RB6==1){
            i--;
            if(i<0)
                i=15;
            updatePins(i);
            __delay_ms(100);
        }

    }
    return (EXIT_SUCCESS);
}
```

Again, you will not be building the circuit shown in Figure 1. I have the working circuit that will be used for demonstration only.

An Alternative Design

The hardware and software design in the previous section can be simplified if you use PORTA for the three button inputs and PORTB for all seven of the LED segment outputs. The original design allows for hardware interrupts which are available only in PORTB. PORTA has only has five bits available. Since we need seven bits to drive the display, the seven outputs are split between PORTA and PORTB.

An alternative design is outlined below using PORTB for all seven outputs and PORTA for the three inputs. This dramatically simplifies the updatePins(i) function obviating the need for the complex logic manipulations of the bits. The changes required to the hardware and software for the alternative design follow.

The pins in PORTB are assigned and connected to the LED segments as follows:

```
pins:        0b76543210
segments:    0b-cdebagf
```

The TRIS registers are initialized as follows:

```
TRISA = 0b00001110;    \\PORTA. 1,2,3 pins are inputs
TRISB = 0b00000000;    \\all PORTB pins are outputs (although the MSB, bit 7, is not used)
```

Then the simpler updatePins(iL) function retains the pins[] definition but the assignment becomes :

```
PORTB = pins[iL];
```

where the bit values in the pins[i] array element are written directly to the PORTB bits driving the LED segments.

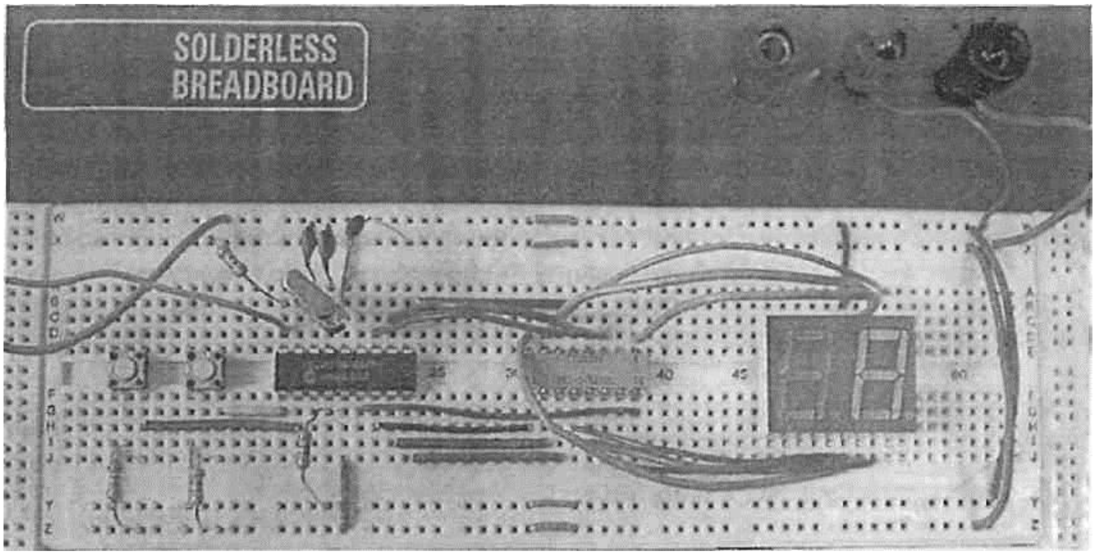


Figure 2 Alternative design hexadecimal counter circuit, you will not use a DIP resistor.

Switch debounce can be performed in software instead of hardware eliminating the need for the 555 timer and D flips-flop portions of the demonstration circuit. Figure 2 shows the hardware for the alternative design (using bits 1,2, and 3 on PORTA). Be sure to wire the switch with pull-down resistors for positive logic.

The switch bounce that can occur when the button is pressed is not a problem in the original software presented above because a 0.1 sec pause gives the button signals more than enough time to settle. However, if a button if held down for more than 0.1 sec and then released, any bouncing that occurs upon release could cause additional increments or decrements. One approach to perform debouncing for the button release is to use a delay in software that waits for the bounce to settle before continuing with the remainder of the program. Here is how the code could be changed for the increment button:

```
while(PORTA.1==1){
        i++;
        if(i==16)
                i=0;
        updatePins(i);
        __delay_ms(200);
}
__delay_ms(10);                 \\For any switch bounce on release.
```

The decrement button is handled in a similar fashion. No debounce is required for the reset button because multiple resets in a short period of time (e.g., the few thousandths of a second when bouncing occurs) do not result in undesirable behavior.

Yet another alternative design that would further simplify the software would be to use a 7447 IC for BCD-to-7-segment decoding. This would eliminate the need for the pins array that does the decoding in software, but it would add an additional IC to the hardware design. Also, the 7447 displays non-alphanumeric symbols for digits above 9, instead of the hexadecimal characters (A, b, C, d, E) that we achieved by control with the pins array.

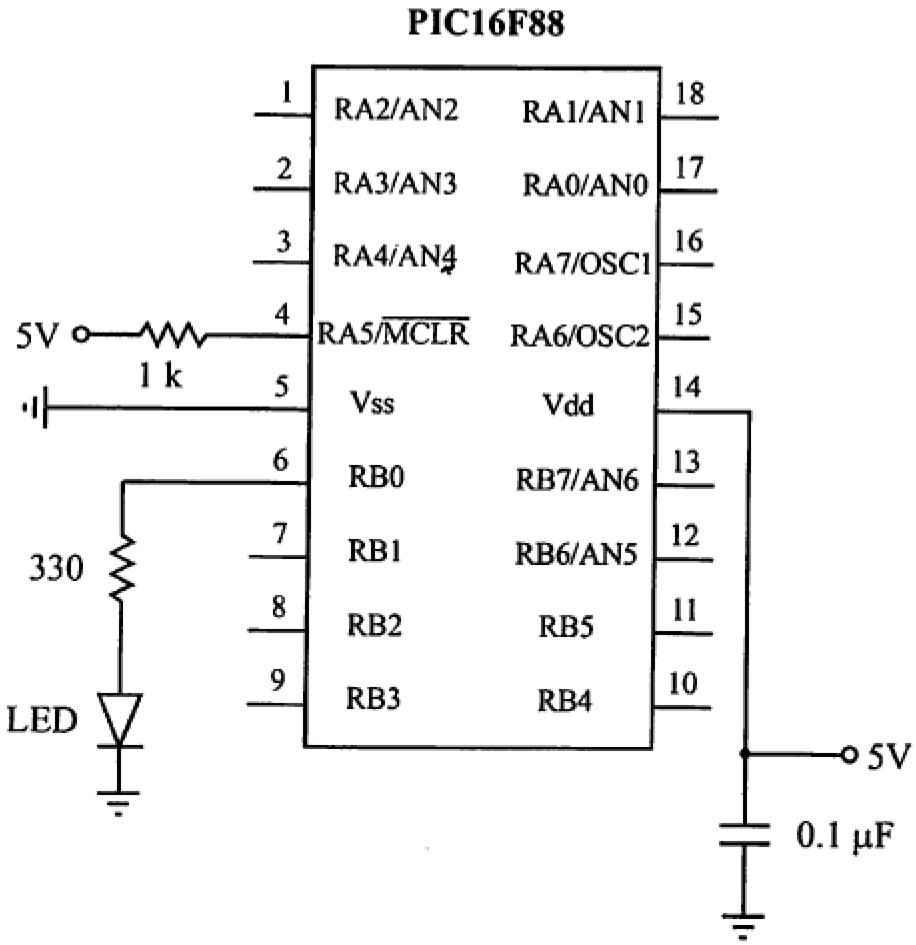
PIC Circuit Debugging Recommendations

It is rare that a wired PIC circuit works the first time it is tested. Often there are "bugs" with the software or the wiring. Here are some recommendations that can help you when trying to get a PIC circuit to function (e.g., with your project):

1. If you are using a PIC that requires an external oscillator (e.g, the PIC16F84x), make sure your circuit includes the necessary clock crystal and capacitor components. If you are using a PIC with an internal oscillator (e.g., the PIC16F88), make sure you include the necessary initialization code (e.g., see the code template for the PIC 16F88 available on the Lab website).
2. If you use MS Word or other word processor to edit your code, make sure you "Save As" a text file, or just copy and paste your code from the word processor into the MPLAB editor.
3. Make sure your wiring is very neat (i.e., not a "rats nest"), keep all of your wires as short as possible to minimize electrical magnetic interference (EMI) (and added resistance, inductance, and capacitance), and use appropriate lengths (about 1/4") for all exposed wire ends (to help prevent breadboard damage and shorting problems).
4. Follow all of the recommendations in Section 7.4 for prototyping IC circuits.
5. **Be very gentle with the breadboards**. Don't force wires into or out of the holes. If you do this, the breadboard might be damaged and you will no longer be able to create reliable connections in the damaged holes or rows.
6. Make sure all components and wires are firmly seated in the breadboard, establishing good connections (especially with larger PICs you might use in your projects). You can check all of your connection with the beep continuity feature on a multimeter.
7. Before writing and testing the entire code for your project, start with the BLINK program in Lab 12 to ensure your PIC is functioning properly. Then incrementally add and test portions of your code one functional component at a time.
8. Use a "chip puller" (small tool) to remove PICs and other ICs from the breadboard to prevent damage (i.e., bent or broken pins).
9. **Always use the PIC programming procedure from the previous Lab to ensure you don't miss any important steps or details.**

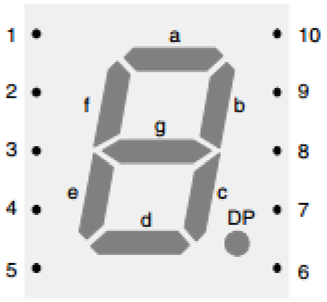
Procedure

1. Observe the demonstration of the original hexadecimal counter circuit. **Do not build this circuit**. The code and circuit are for demonstration only, and serve as an additional example. Study the program listed and fully test all functionality of the demo board. Examine the effect of holding down a button. Examine the effects of holding down multiple buttons at once.
2. Using the figure below as a starting point, draw a complete and detailed schematic required to implement the alternative counter design described in the Alternative Design section. Figure 3 shows useful information from the 7 segment datasheet. Show me your diagram before you continue. **PLEASE COMPLETE THIS BEFORE COMING TO LAB**. Note - your schematic will be very different from the one shown in Figure 1.



3. Use an ASCII editor (e.g., Windows Notepad or MS Word - Text Only) to create the program necessary to control the alternative design. Name it "counter.c". Bring the file with you on a flashdrive and call it “counter.c”. **PLEASE COMPLETE THIS BEFORE COMING TO LAB.**
4. Follow the procedure in the previous laboratory exercise to compile the program and load it onto a PIC.

5. Assemble and fully test your circuit with the programmed PIC. If you are having problems, please refer to the Debugging section for advice on how to get things working. When everything is working properly, demonstrate it to me for credit.



1	g
2	f
3	A (C)
4	e
5	d
6	DP
7	c
8	A (C)
9	b
10	a

96 11678

Figure 3 Seven segment display data.

Laboratory 13 Questions

Names___________________________________

1. Rewrite "offInterrupt.c" from the previous Lab using polling instead of interrupts.

2. For the original (demo) counter design when the 'up' button is held down awhile the PIC will continue to count up. Explain why.

3. For the original (demo) counter design, explain what happens when the 'up' and 'down' buttons are held down together. Why does this happen?

4. For the counter design you built, why is the 555 and D flip-flop hardware no longer required?

5. Explain how switch bounce could possibly have a negative impact with the design you built if the 0.01 sec software pause were not included.

6. Explain why debounce software is not required for the reset button in the design you built.

7. For the original (demo) counter design (i.e., not the design that you built), how would you create the functionality in the updatePins(i) function for updating the PORTA and PORTB registers using multiple individual bit references (e.g., PORTA.0 = pins[i].4, PORTA. 1 = pins[i].5, ...) instead of single-line assignment statements (e.g., PORTA = ... and PORTB = ...)? **Hint**: The comments above the assignment statements in the code explain what is being done.