Laboratory 10

Programming a PIC Microcontroller - Part II

Required Components:

- 1 PIC16F88 18P-DIP microcontroller
- 1 0.1 μF capacitor
- 3 SPST microswitches or NO buttons
- 4 1kΩ resistors
- 1 MAN 6910 or LTD-482EC seven-segment LED digital display
- 1 330Ω DIP resistor array

Required Special Equipment and Software:

- Mecanique's Microcode Studio integrated development environment software
- MicroEngineering Labs' PicBasic Pro compiler
- MicroEngineering Labs' U2 USB Programmer
- Demonstration hexadecimal counter circuit board containing a 555 timer circuit and a D flip-flop latch IC

10.1 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 hexadecimal counter project demonstrated in the video on the Lab website. You will then create and test an alternative design using different hardware and software.

10.2 Hexadecimal Counter 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. In the previous examples that also required an output, this was not necessary since the High and Low commands set the TRIS registers automatically. In this example we will use assignment statements to set the PORT output values directly (e.g., PORTB = %00011001), requiring setting of the TRIS registers. Most PICBasic commands that use pins as outputs or inputs automatically set the TRIS register bits to appropriate values.

Setting a TRIS register bit to 0 designates an output and setting the bit to 1 designates an input. For example,

TRISA = %00000000

designates all bits of PORTA as outputs and

TRISB = %01110000

designates bits 4, 5, and 6 of PORTB as inputs and the others as outputs.

Note that since PORTA has only 5 usable bits (bits 0 through 4), 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=$FF and TRISB=$FF).

The code "counter.bas" for the up/down hex counter in listed below. **NOTE - You will need to modify this code, and the circuit, for this Laboratory. See Sections 10.3 and 10.5 for more information.**

```
' counter.bas
' PicBasic hex up/down counter

' Identify and set the internal oscillator clock speed (required for the PIC16F88)
DEFINE OSC 8
OSCCON.4 = 1
OSCCON.5 = 1
OSCCON.6 = 1

' Turn off the A/D converter (required for the PIC16F88)
ANSEL = 0

' Declare variables
pins        var     byte[16]    ' an array of 16 bytes used to store the 7-segment display codes
I           var     byte        ' counter variable
```

```
' Initialize I/O pins
TRISA = %00000000          ' all PORTA pins initialized as outputs
                           '   (although only pins 0, 1, and 2 are used)
TRISB = %01110000          ' PORTB.4,5,6 pins initialized as inputs
                           '   (RB4: reset,  RB5: increment,  RB6: decrement)
                           ' all other PORTB pins initialized as outputs


' Initialize the pin values for the 7-segment LED digit display with segments as illustrated below
'                            a
'                        -------
'                       |       |
'                   f   |       |   b
'                       |   g   |
'                        -------
'                       |       |
'                   e   |       |   c
'                       |       |
'                        -------
'                            d


'  Three of the pins in PORTA and four of the pins in PORTB are used as the seven outputs.
'  The LED segments are assigned to the PORT bits as shown below:
'                    PORTA                    PORTB
' bit number:     76543210                 76543210
' segment:        -----cde                 ----bagf

' The PORT pin numbers (on the PIC pin-out) correspond to the PORT bit numbers as follows:
' LED segment:          e       d       c
' PORTA bit:            0       1       2
' PORTA pin name:     RA0     RA1     RA2
' PORTA pin number:    17      18       1

' LED segment:          f       g       a       b
' PORTB bit:            0       1       2       3
' PORTB pin name:     RB0     RB1     RB2     RB3
' PORTB pin number:     6       7       8       9

'  NOTE: 0 turns a segment ON and 1 turns it OFF since the PIC sinks current from the LED display
'                  binary           hex      display
'              %0cdebagf  (correspondence between LED segments and pins array bits)
pins[ 0] = %00000010    ' 02       0
pins[ 1] = %00110111    ' 37       1
pins[ 2] = %01000001    ' 41       2
pins[ 3] = %00010001    ' 11       3
pins[ 4] = %00110100    ' 34       4
pins[ 5] = %00011000    ' 18       5
pins[ 6] = %00001000    ' 08       6
```

```
pins[ 7] = %00110011   ' 33      7
pins[ 8] = %00000000   ' 00      8
pins[ 9] = %00110000   ' 30      9
pins[10] = %00100000   ' 20      A
pins[11] = %00001100   ' 0C      b
pins[12] = %01001010   ' 4A      C
pins[13] = %00000101   ' 05      d
pins[14] = %01001000   ' 48      E
pins[15] = %01101000   ' 68      F
'           %0cdebagf  (correspondence between LED segments and pins array bits)


' Initialize the display to zero
I = 0
Gosub Updatepins

' Main loop
myloop:

  If (PORTB.4 == 1) Then    ' reset
    I = 0
    Gosub Updatepins
    Pause 100      ' 0.1 sec delay
  Endif

  If (PORTB.5 == 1) Then    ' increment
     If (I == 15) Then
        I = 0
     Else
        I = I + 1
     Endif

     Gosub Updatepins
     Pause 100      ' 0.1 sec delay
  Endif

  If (PORTB.6 == 1) Then    ' decrement
     If (I == 0) Then
         I = 15
     Else
         I = I - 1
     Endif

     Gosub Updatepins
     Pause 100      ' 0.1 sec delay
  Endif

Goto myloop       ' go back to the beginning of the loop and continue to poll the inputs
```

```
' Updatepins Subroutine
'   sends new output values to pins

Updatepins:
   ' Use the right shift operator to move the top MSB's of pins[I] to the 4 LSB's of PORTA
   '    padding the 4 MSBs of PORTA with 0's
   PORTA = pins[I] >> 4
   ' Use logic to retain the 4 MSB's of PORTB and replace the 4 LSB's of PORTB by
   '    by the 4 LSB's of pins[I]
   PORTB = (PORTB | %00001111) & (pins[I] | %11110000)
   Return

End               ' End of program
```

After the initial comments labeling the program, the pins variable is defined as an array of 16 bytes. Elements in the array are accessed by the syntax pins[I], where I is the index having values from 0 through 15. Binary values for the pins array elements are set in the "Initialize the pin values ..." section. The mapping of these bits to individual segments on the 7-segment counter display, and the structure of the counter circuit dictates the assignment of each of these bits to a specific segment. The 8 bits in each byte 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. Again, 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.

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 actually exist 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.

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 index for the pins[I] variable is set to 0 so that the display will be zero. The Updatepins routine is called to update the value displayed as described in detail below. 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 index for the pins[I] variable. The internally nested IF statement checks if the index exceeds the allowed value of 15, and if it does the index 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 updates the display, and a 0.1 second pause occurs. The pause prevents the count from updating too quickly while the button is

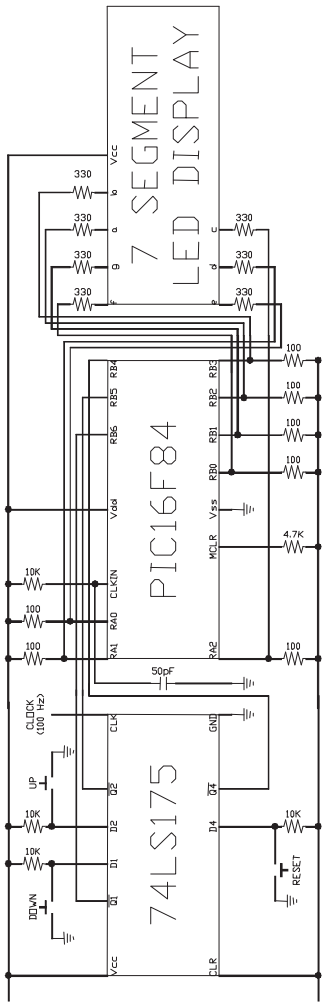
being held down before it is released. If the button is held down for more than 0.1 second the counter will increment every 0.1 second. Then PORTB.6 is checked and a value of 1 will cause a decrement in the index for the pins[I] variable. Once the display reaches a minimum value of zero, the routine will cycle back to F for the next hexadecimal value.

The most direct action in displaying the output occurs in the Updatepins routine. 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 maintain the existing values for bits 4 through 7 (the MSB's of PORTB) since these bits are reserved for the button inputs, while 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 maintained, 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 routine.

As shown in Figure 10.1, 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 $\overline{Q}$ 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 5V.

Again, **you will not be building the circuit shown in Figure 10.1**. You will just view a video demonstration of the working circuit on the Lab website.

NOTE:
These resistors
are on a DIP IC.



NOTE - Do not build this circuit.
You will build an alternative design instead (see Sections 10.3 and 10.5).

Figure 10.1 Circuit Diagram for the **Demonstration-Only** Hexadecimal Counter

10.3 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. This was not done for the example above since the hardware was originally designed and built by a graduate student assuming hardware interrupts would be used. Hardware interrupts are available only in PORTB, and if the three buttons were attached to PORTB, only five bits in PORTB would be available to be used as outputs. Furthermore, on the PIC16F84, PORTA only has five bits available. Because we need seven bits to drive the display, both PORTA and PORTB were used for the seven outputs.

An alternative design is outlined below using PORTB for all seven outputs and PORTA for the three inputs. This dramatically simplifies the Updatepins subroutine eliminating the need for the complex logic manipulations of the bits. The changes required to the hardware and software for the alternative design follow.

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

```
bit number:   %76543210
segment:      %-cdebagf
```

The TRIS registers are initialized as follows:

```
TRISA = %00001110          ' PORTA.1,2,3 pins are inputs
TRISB = %00000000          ' all PORTB pins are outputs (although, pin 7 is not used)
```

Then the simpler Updatepins subroutine is:

```
Updatepins:
   PORTB = pins[I]
   Return
```

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

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 10.2 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.**

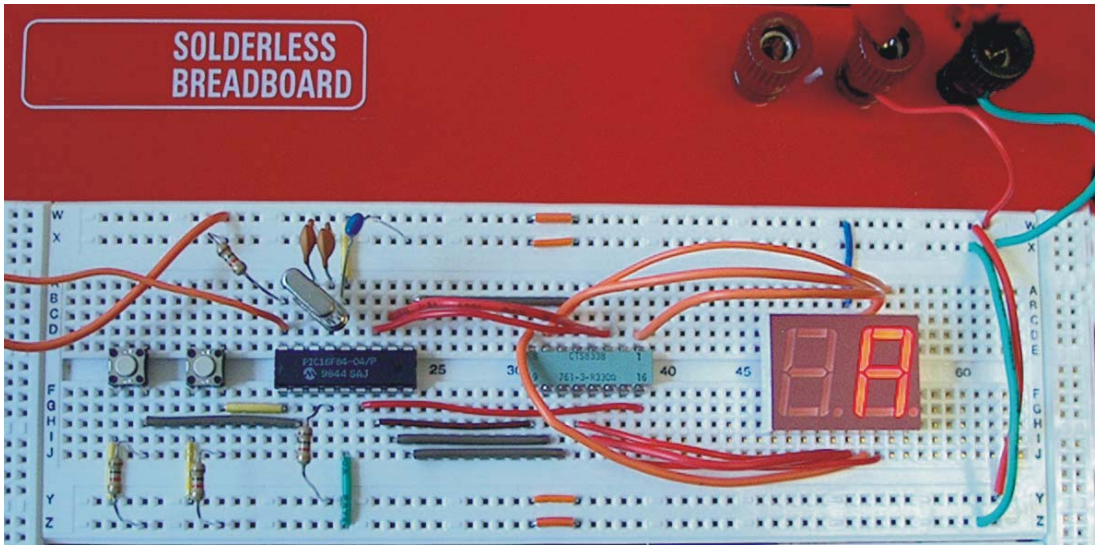


Figure 10.2 Alternative Design Hexadecimal Counter Circuit

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:

```
' Continue to increment every 0.2 sec while the increment button is being held down

Do While (PORTA.1 == 1)
        If (I == 15) Then
                I = 0
        Else
                I = I + 1
        Endif

        ' Update the display
        Gosub Updatepins

        ' Hold the current count on the display for 0.2 sec before continuing
        Pause 200
Loop


' Pause for 0.01 sec to allow any switch bounce to settle after button release
Pause 10
```

The decrement button would be 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 control with the pins array.

10.4 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 PIC16F88 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 MicroCode Studio or 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 9 to ensure your PIC is functioning properly. Then incrementally add and test portions of your code one functional component at a time.

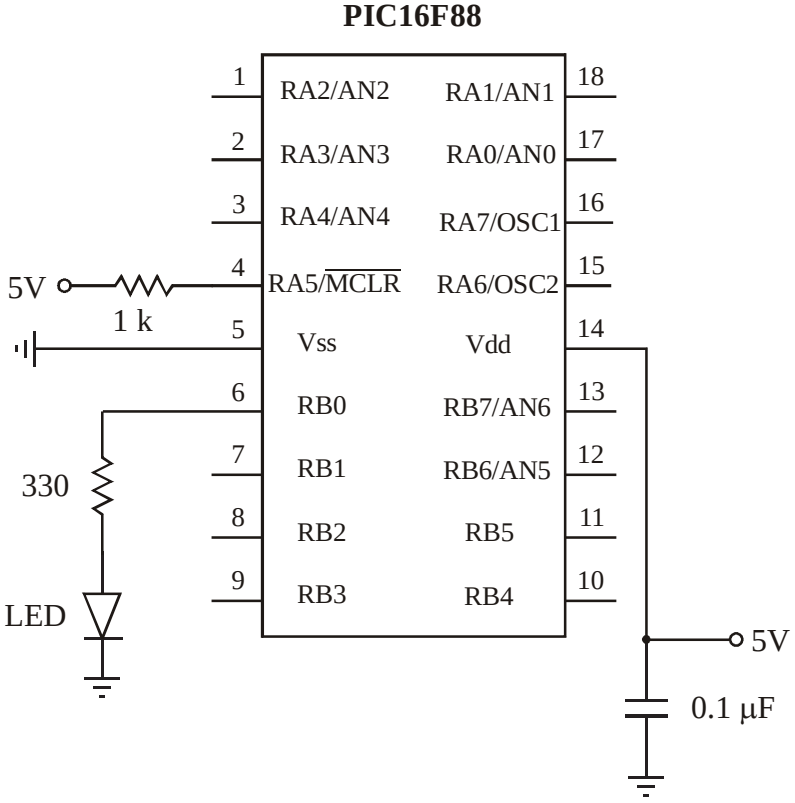
(8) Use a "chip puller" (small tool) to remove PICs and other ICs from the breadboard to prevent damage (i.e., bent or broken pins).

(9) Always use the PIC programming procedure in Section 9.4 of the previous Lab to ensure you don’t miss any important steps or details.

10.5 Procedure

(1) Watch the video demonstration on the Lab website of the original hexadecimal counter circuit described in Section 10.2. **Do not build this circuit.** The code and circuit in Section 10.2 is for demonstration only, and it serves as an additional example. Study the program listed in Section 10.2 and observe the functionality in the video, including the effects of holding down buttons.

(2) Using the figure below as a starting point, draw a complete and detailed wiring diagram required to implement the alternative counter design described in Section 10.3. Figure 10.3 shows useful information from the MAN6910 datasheet. Have your TA check your diagram before you continue. **PLEASE COMPLETE THIS BEFORE COMING TO LAB. NOTE: Your diagram will be very different from the one shown in Figure 10.1.**

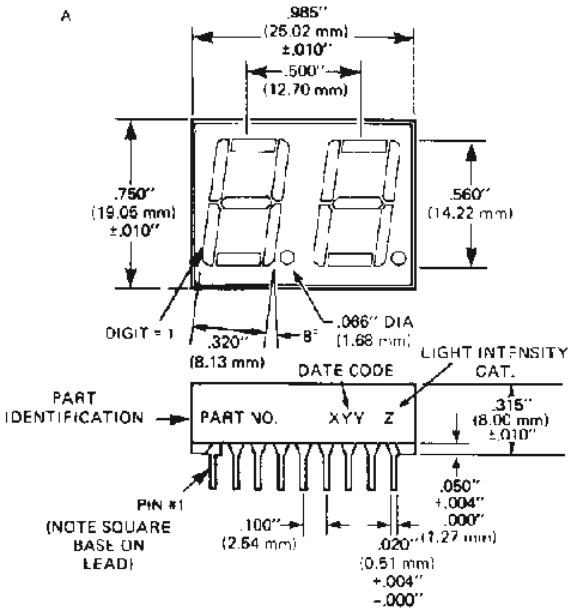


(3) Use an ASCII editor (e.g., Windows Notepad or MS Word - Text Only), or use Microcode Studio in Lab, to create the program necessary to control the alternative design. Name it "counter.bas". Save the file in a folder in your network file space named "counter." **PLEASE COMPLETE THIS BEFORE COMING 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. **NOTE: Make sure you use the power supply, and not the function generator, to power the circuit.**

Use the 330Ω DIP IC for the current-limiting resistors. If you are having problems, please refer to Section 10.4 for advice on how to get things working. When everything is working properly, demonstrate it to your TA for credit.



PIN NO.	A MAN6910
1	Cathode E 1
2	Cathode D 1
3	Cathode C 1
4	Cathode D.P. 1
5	Cathode E 2
6	Cathode D 2
7	Cathode G 2
8	Cathode C 2
9	Cathode D.P. 2
10	Cathode B 2
11	Cathode A 2
12	Cathode F 2
13	Anode Digit 2
14	Anode Digit 1
15	Cathode B 1
16	Cathode A 1
17	Cathode G 1
18	Cathode F 1

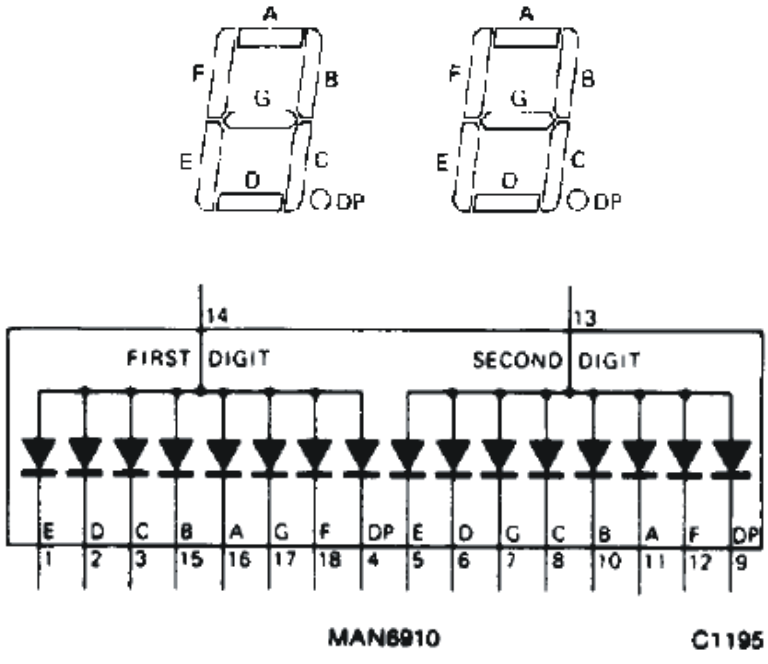


Figure 10.3 MAN6910 Datasheet Information

LAB 10 QUESTIONS

Group: ____ Names: _____ _____

_____ _____

(1) Rewrite "onint.bas" from the previous Lab using polling instead of interrupts.

(2) For the original counter design in Section 10.2, when the 'up' button is held down awhile the PIC will continue to count up. Explain why.

(3) For the original counter design in Section 10.2, explain what happens when the 'up' and 'down' buttons are held down together. Why does this happen?

(4) For the alternative counter design in Section 10.3, 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 alternative design in Section 10.3 if the 0.01 sec software pause were not included.

(6) Explain why debounce software is not required for the reset button in the alternative design in Section 10.3.

(7) For the original counter design in Section 10.2 that was demonstrated in the video (i.e., not the alternative design in Section 10.3 that you built), how would you create the functionality in the Updatepins subroutine 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.
Hint: The comments below the 7-segment-display illustration in the "counter.bas" program involving the "bit numbers" and "segments" can be helpful.