Chapter 9
PICF18 Interrupt I/O, LCD, and Keyboard Interfacing

Chien-Chung Ho (何建忠)

Basics of Polled I/O vs. Interrupt I/O

- Two ways programmed I/O can be utilized: unconditional I/O and conditional I/O.
- The microcontroller can send data to an external device at any time using unconditional I/O. The external device must always be ready for data transfer. A typical example is when the microcontroller outputs a 7-bit code through an I/O port to drive a seven-segment display connected to this port.

Basics of Polled I/O vs. Interrupt I/O

- In conditional I/O, the microcontroller outputs data to an external device via handshaking. This means that data transfer occurs via the exchanging of control signals between the microcontroller and an external device. The microcontroller inputs the status of the external device to determine whether the device is ready for data transfer. Data transfer takes place when the device is ready.

Basics of Polled I/O vs. Interrupt I/O

- The concept of polled I/O will now be demonstrated by means of data transfer between a generic microcontroller and an analog-to-digital (A/D) converter or ADC. Assume that this microcontroller does not have any on-chip A/D converter. Next, consider the A/D converter chip shown in Figure 9.2.

Basics of Polled I/O vs. Interrupt I/O

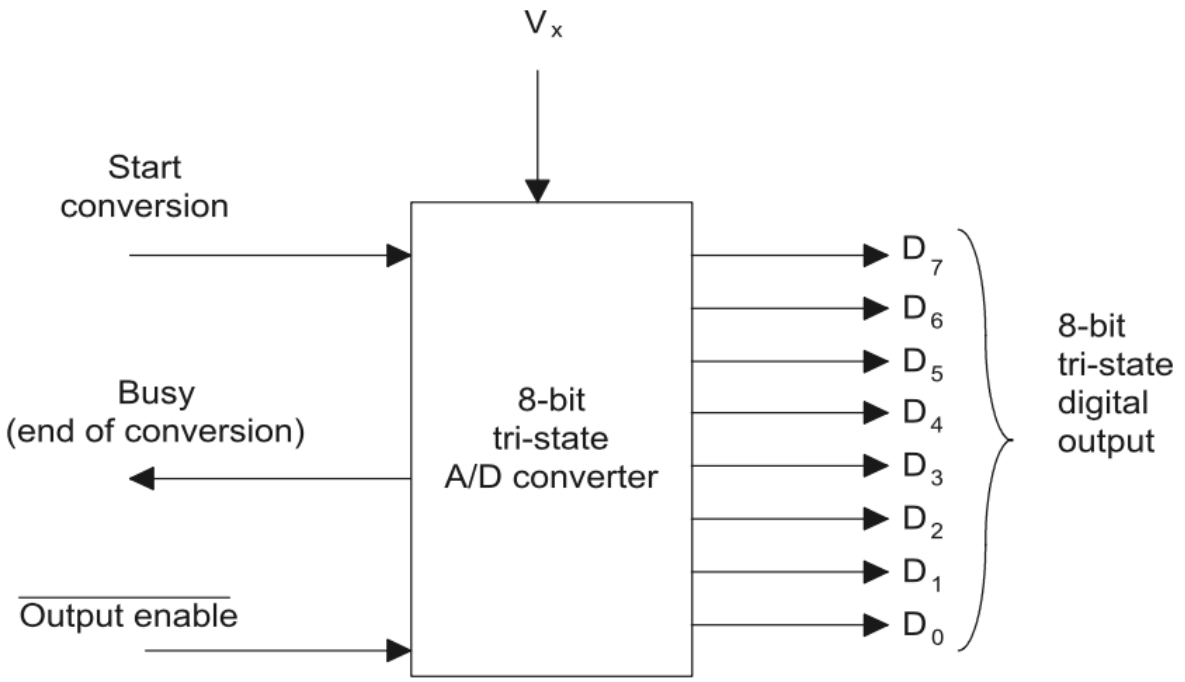


FIGURE 9.2 A/D converter

Basics of Polled I/O vs. Interrupt I/O

- The A/D converter transforms an analog voltage Vx into an 8-bit binary output at pins D7-D0. A pulse at the START conversion pin initiates the conversion. This drives the BUSY signal to LOW. The signal stays LOW during the conversion process. The BUSY signal goes to HIGH as soon as the conversion ends. Note that a LOW on the $\overline{\text{OUTPUT ENABLE}}$ transfers the A/D converter's outputs.

Basics of Polled I/O vs. Interrupt I/O

- The concept of polled I/O can be demonstrated by interfacing an external A/D converter chip to the microcontroller chip. Figure 9.3 shows such an interfacing example. The user writes a program to carry out the conversion process. When this program is executed, the microcontroller sends a pulse to the START pin of the converter via bit 2 of port A.

Basics of Polled I/O vs. Interrupt I/O

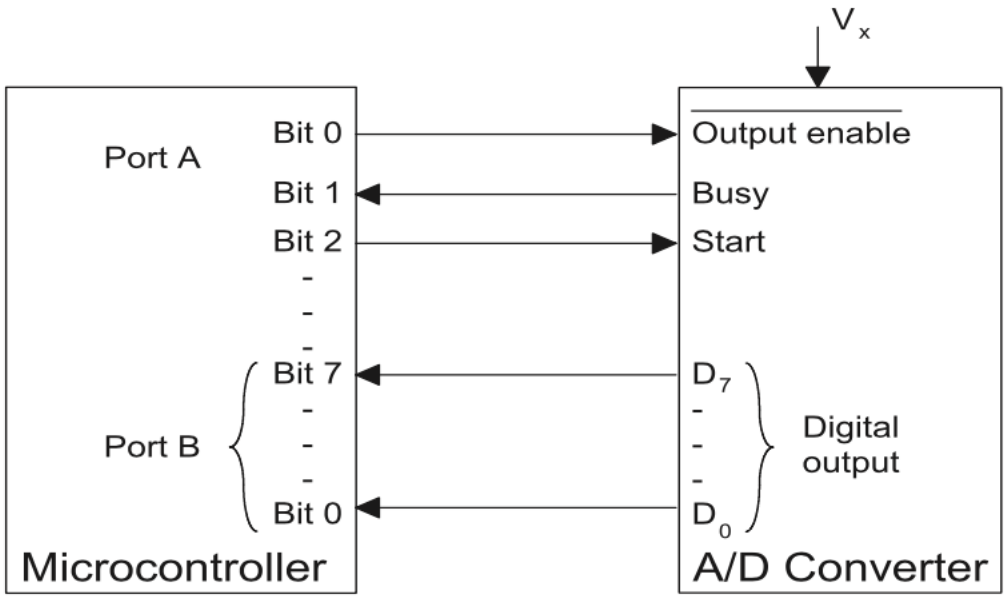


FIGURE 9.3 Interfacing an A/D converter to a generic microcontroller

Basics of Polled I/O vs. Interrupt I/O

- The microcontroller then checks the BUSY signal by inputting bit 1 of port A to determine if the conversion is completed. If the BUSY signal is HIGH (indicating the end of conversion), the microcontroller sends a LOW to the $\overline{\text{OUTPUT ENABLE}}$ pin of the A/D converter.

Basics of Polled I/O vs. Interrupt I/O

- The microcontroller then inputs the converter's D0-D7 outputs via port B. If the conversion is not completed, the microcontroller waits in a loop checking for the BUSY signal to go to HIGH. This is called "Conditional or Polled I/O". A disadvantage of polled I/O is that the microcontroller needs to check the status bit (BUSY signal for the A/D converter) by waiting in a loop.

Basics of Polled I/O vs. Interrupt I/O

- This type of I/O transfer is dependent on the speed of the external device (A/D Converter in this case). For a slow device, this waiting may slow down the microcontroller's capability of processing other data in the main program. The interrupt I/O technique is efficient in this type of situation.

Basics of Polled I/O vs. Interrupt I/O

- Interrupt I/O is a device-initiated I/O transfer. The external device is connected to a pin called the “interrupt (INT) pin” on the microcontroller chip as shown in Figure 9.4. When the microcontroller needs an I/O transfer with the A/D converter, a user-written program starts the A/D conversion via bit 7 of port A. The A/D converter chip performs conversion independent of the microcontroller.

Basics of Polled I/O vs. Interrupt I/O

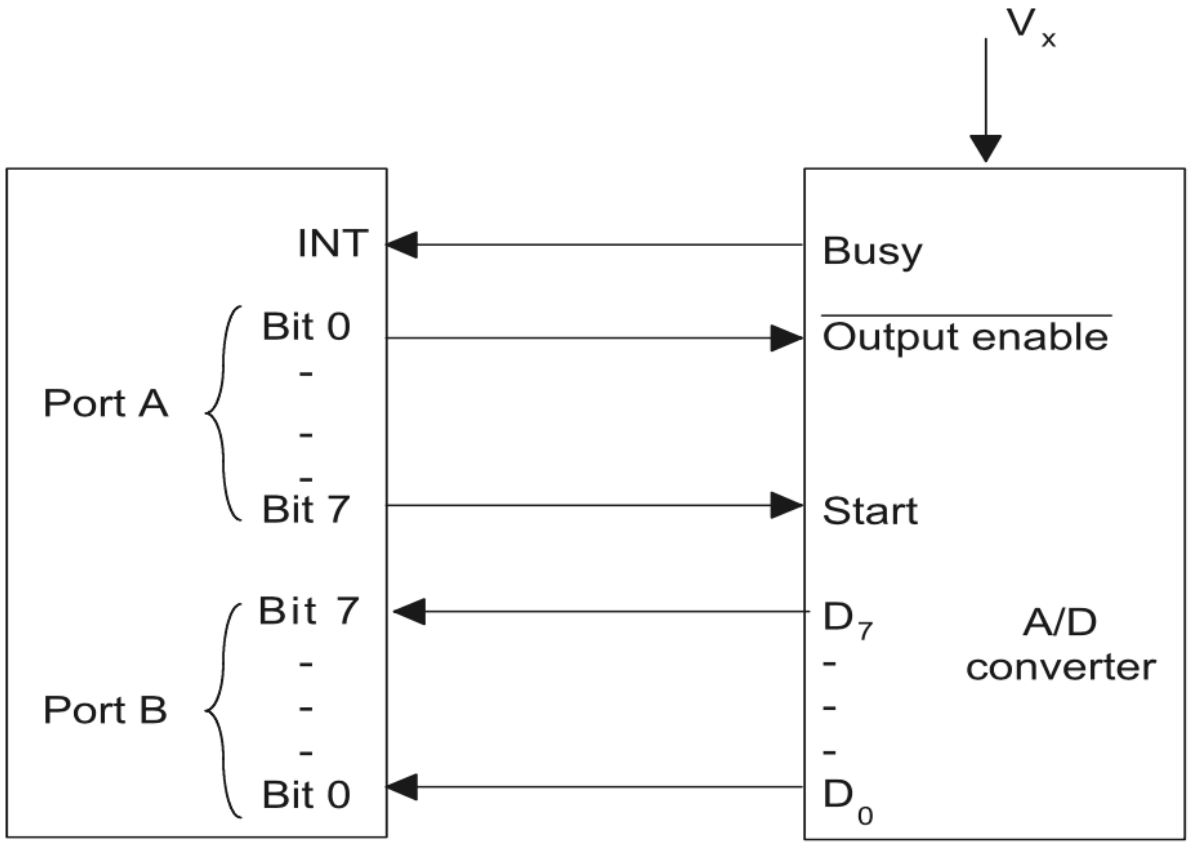


FIGURE 9.4 Microcomputer A/D converter interface via interrupt I/O

Basics of Polled I/O vs. Interrupt I/O

- When the conversion is completed, the A/D converter outputs a HIGH on its BUSY pin. This will activate the interrupt pin of the microcontroller. The microcontroller completes the current instruction and normally saves the contents of the current program counter (RETURN address) and the status register onto the stack.

Basics of Polled I/O vs. Interrupt I/O

- The microcontroller then automatically loads an address into the program counter to branch to a subroutine-type program called the “interrupt-service routine.” This program is written by the user. The A/D converter chip wants the microcontroller to execute this program to input data. The last instruction of the service routine is a “RETURN from interrupt” which is typically similar in concept to the RETURN instruction used at the end of a subroutine.

Basics of Polled I/O vs. Interrupt I/O

- Handling interrupt I/O is very similar in concept to subroutine CALL and RETURN instructions. The basic differences:
  - Execution of a subroutine program is initiated using software by a microcontroller's 'CALL to subroutine' instruction in the main program. The interrupt, on the other hand, is initiated using hardware by activating an interrupt pin by an external device.

Basics of Polled I/O vs. Interrupt I/O

- Handling interrupt I/O is very similar in concept to subroutine CALL and RETURN instructions. The basic differences:
  - The instructions ‘RETURN from subroutine’ and ‘RETURN from interrupt’ are different. The PIC18F ‘RETURN from subroutine’ will pop the program counter while the PIC18F ‘RETURN from interrupt’ will pop the program counter and also enable the interrupts.

Basics of Polled I/O vs. Interrupt I/O

Example 9.1 Assume that the PIC18F4321 microcontroller shown in Figure 9.5 is required to perform the following:

If Vx > Vy , turn the LED ON if the switch is open; otherwise, turn the LED OFF.

(a) Write a PIC18F assembly language program to accomplish the above by polling the comparator output via bit 0 of PORTB.

(b) Write a C program to accomplish the above by inputting the comparator output via bit 0 of PORTB.

Basics of Polled I/O vs. Interrupt I/O

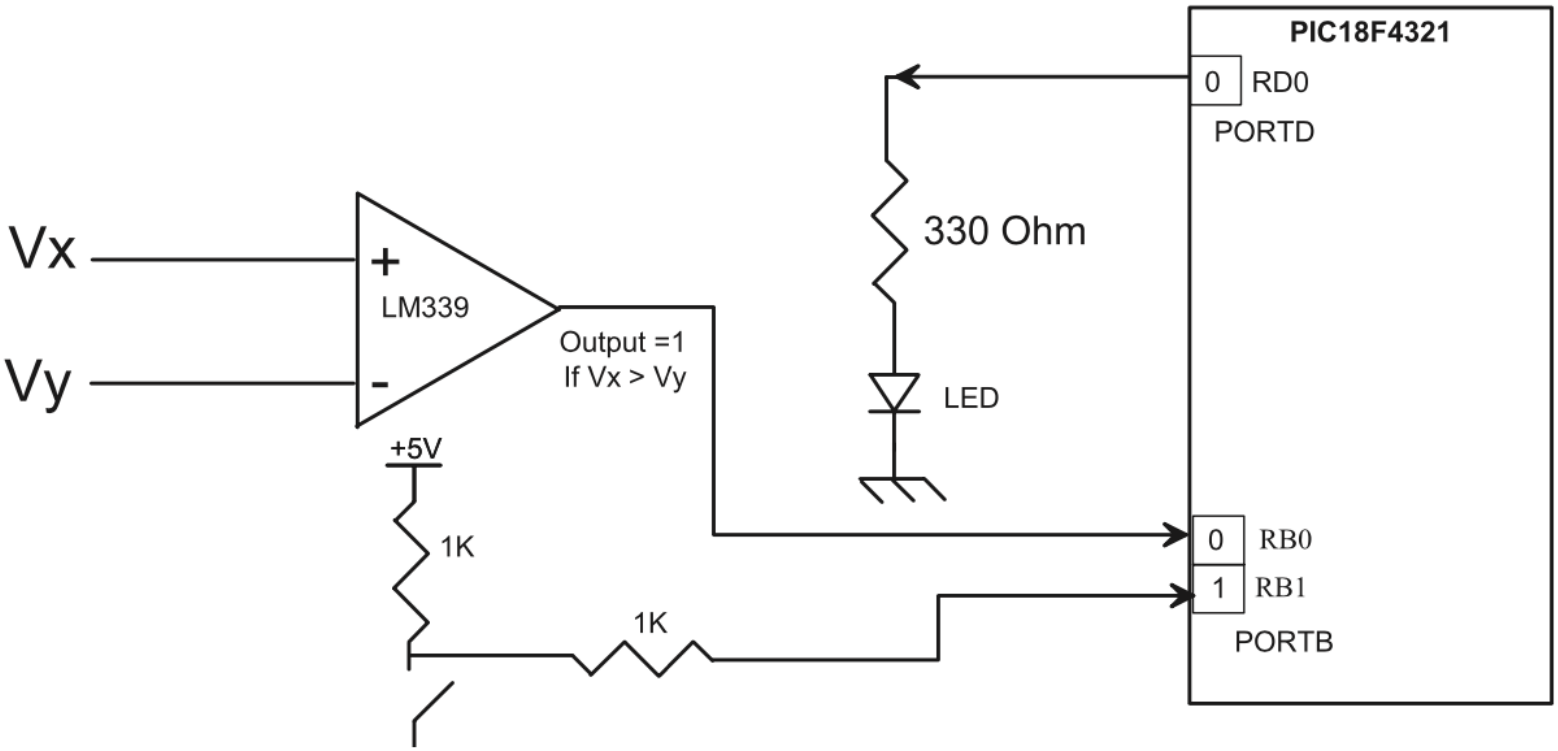
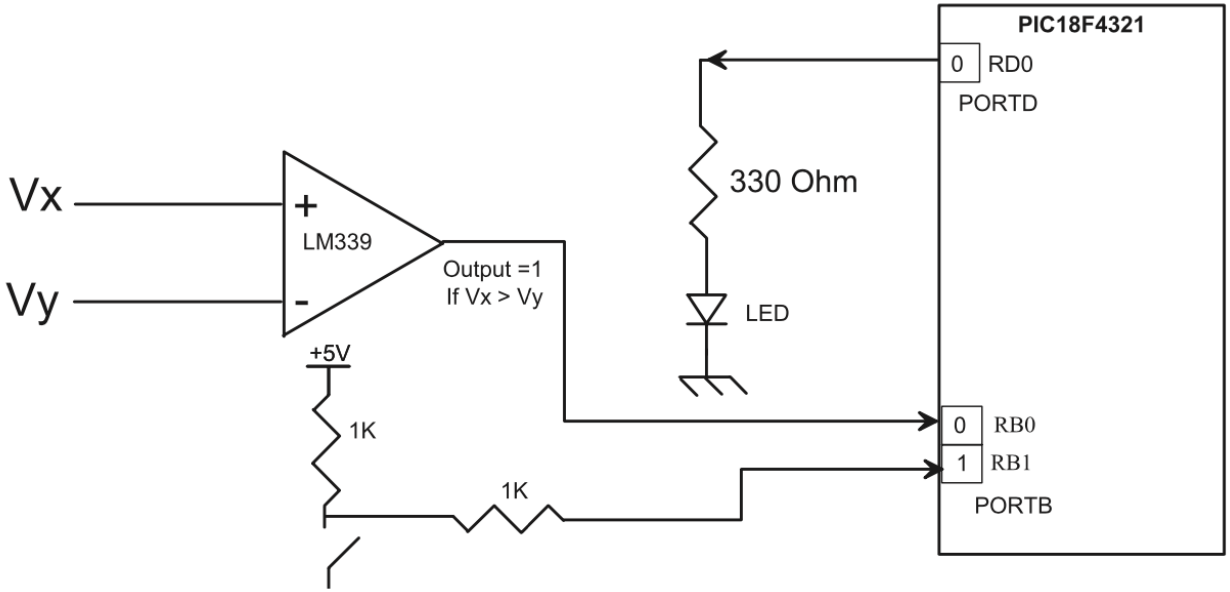


FIGURE 9. 5 Figure for Example 9.1

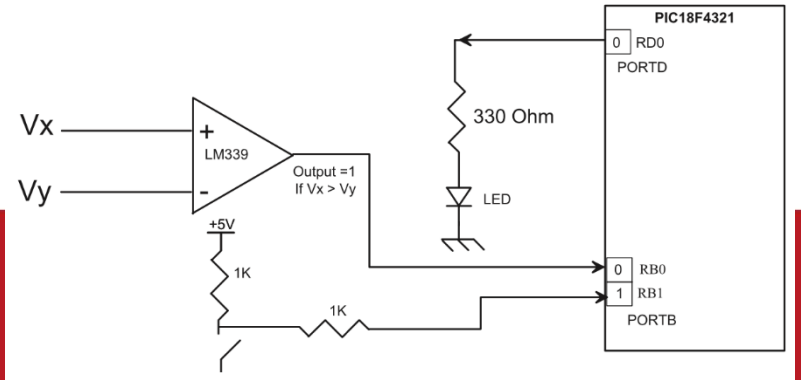
Basics of Polled I/O vs. Interrupt I/O

- An LM339 comparator is connected to control when the LED will be turned ON or OFF. In Figure 9.5, when Vx > Vy the comparator will output a one, and the PIC18F4321 will turn the LED ON or OFF depending on the switch status.
- If Vx < Vy, then the comparator will output a zero and the LED will be turned OFF. In the program, the ADCON1 register is used to configure RB0 and RB1 as inputs. The TRISD is used to make RD0 of PORTD an output.



Basics of Polled I/O vs. Interrupt I/O

```
        INCLUDE <P18F4321.INC>
        ORG     0               ; RESET VECTOR
        BRA     MAIN            ; JUMP TO MAIN
        ORG     0x100
MAIN    BCF     TRISD, TRISD0   ; CONFIGURE BIT 0 OF PORTD AS OUTPUT
        MOVLW   0x0F            ; CONFIGURE BITS 0 AND 1 OF PORTB
        MOVWF   ADCON1          ; AS DIGITAL INPUTS
BEGIN   BCF     PORTD, RD0      ; TURN LED OFF
CHECK   BTFSS   PORTB, RB0      ; CHECK IF COMPARATOR OUTPUT IS ONE
        BRA     BEGIN           ; WAIT IN LOOP UNTIL ONE
        BTFSS   PORTB, RB1      ; CHECK IF SWITCH IS OPEN
        BRA     BEGIN
        BSF     PORTD, RD0      ; IF SWITCH IS OPEN, TURN LED ON
        BRA     CHECK
        END
```



Basics of Polled I/O vs. Interrupt I/O

- PORTB and PORTD are configured. The instruction BCF PORTD, RD0 turns the LED OFF. In order to check whether the comparator is outputting a one, the instruction BTFSS PORTB, RB0 is used in the program. After execution of this instruction, if bit 0 of PORTB (comparator output) is 0, the next instruction, BRA BEGIN continues looping until the comparator outputs a one. However, if the comparator output is 1, the BTFSS PORTB, RB0 will skip the next instruction (BRA BEGIN).

Basics of Polled I/O vs. Interrupt I/O

(b) The C program is provided below:

```
#include <p18f4321.h>

void main (void)
{
TRISDbits.TRISD0 = 0;   // PORD is output
ADCON1 = 0x0F;          // Configure for PORTB to be digital input
PORTDbits.RD0 = 0;      // Turn LED OFF

while(1)
{
        PORTDbits.RD0 = 0;              // Turn LED OFF
        while(PORTBbits.RB0 = =1)       // While Vx > Vy
        {
                if(PORTBbits.RB1= =1)
                        PORTDbits.RD0 = 1;      // Turn LED ON
                else if (PORTBbits.RB1= = 0)
                        PORTDbits.RD0 = 0;      // Turn LED OFF
        }
}
}
```

PIC18F Interrupts

- PIC18F interrupts are basically divided into two types, namely external and internal interrupts.
- External interrupts are initiated by external devices via the PIC18F's interrupt pins (INT0-INT2). Internal interrupts are activated internally by on-chip peripheral devices (ADC, Hardware timers) for conditions such as interrupts due to "completion of on-chip A/D conversion" and "hardware timers reaching the programmed limits".

PIC18F Interrupt Types

- These interrupts are activated by the leading edge (LOW to HIGH) pulses in default mode. The PIC18F has the flexibility of changing the triggering levels to a trailing edge (HIGH to LOW) via programming a register (to be discussed later).

PIC18F Interrupt Types

- For edge-triggered interrupts, in order for the PIC18F to recognize an interrupt such as INT0, the INT0 pin must be held LOW for at least two instruction cycles and HIGH for at least two instruction cycles. This means that if the internal clock of 1MHz is used, then each instruction cycle time is 4 microseconds. Hence, the PIC18F will recognize an interrupt if it is LOW for at least 8 microseconds (two instruction cycles) and HIGH for at least 8 microseconds (two instruction cycles).

Programming the PIC18F Ext. Interrupts

- Upon power-on reset, the PIC18F handles three external interrupts (INT0, INT1, INT2) in “default mode”. The starting address of the service routine for all three interrupts is the same address 0x00008 in the program memory. Each of the three external interrupts has an individual interrupt enable bit along with a corresponding flag bit located in a register. For example, the INT0 interrupt enable bit (INT0IE) and the INT0 interrupt flag bit (INT0IF) are located in the INTCON register.

Programming the PIC18F Ext. Interrupts

- Each of the other two external interrupts (INT1 and INT2), on the other hand, have individual interrupt enable bits (INT1IE, INT2IE) along with the corresponding flag bits (INT1IF, INT2IF) in another register called the INTCON3 register. Figure 9.7 shows the INTCON and INTCON3 registers along with the associated interrupt bits.

Programming the PIC18F Ext. Interrupts

(a) INTCON Register

7	6	5	4	3	2	1	0	
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0I F	INT0I F	RBIF	INTCON

bit 7 **GIE/GIEH:** Global Interrupt Enable bit
When IPEN = 0: 1 = Enables all unmasked interrupts 0 = Disables all interrupts
When IPEN = 1: 1 = Enables all high priority interrupts 0 = Disables all interrupts
bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit
When IPEN = 0: 1 = Enables all unmasked peripheral interrupts 0 = Disables all peripheral interrupts
When IPEN = 1: 1 = Enables all low priority peripheral interrupts 0 = Disables all low priority peripheral interrupts
bit 5 **TMR0IE:** TMR0 Overflow Interrupt Enable bit,
1 = Enables the TMR0 overflow interrupt 0 = Disables the TMR0 overflow interrupt
bit 4 **INT0IE:** INT0 External Interrupt Enable bit
1 = Enables the INT0 external interrupt, 0 = Disables the INT0 external interrupt
bit 3 **RBIE:** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt, 0 = Disables the RB port change interrupt
bit 2 **TMR0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software), 0 = TMR0 register did not overflow
bit 1 **INT0IF:** INT0 External Interrupt Flag bit
1 = The INT0 external interrupt occurred (must be cleared in software), 0 = The INT0 external interrupt did not occur
bit 0 **RBIF:** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Programming the PIC18F Ext. Interrupts

(b) **INTCON3 Register**

7	6	5	4	3	2	1	0	
INT2I P	INT1I P	---	INT2IE	INT1IE	---	INT2I F	INT1I F	INTCON3

bit 7 **INT2IP:** INT2 External Interrupt Priority bit, 1 = High priority 0 = Low priority
bit 6 **INT1IP:** INT1 External Interrupt Priority bit, 1 = High priority 0 = Low priority
bit 5 **Unimplemented:** Read as '0'
bit 4 **INT2IE:** INT2 External Interrupt Enable bit, 1 = Enables the INT2 interrupt
0 = Disables the INT2 interrupt
bit 3 **INT1IE:** INT1 External Interrupt Enable bit, 1 = Enables the INT1 interrupt
0 = Disables the INT1 interrupt
bit 2 **Unimplemented:** Read as '0'
bit 1 **INT2IF:** INT2 External Interrupt Flag bit, 1 = The INT2 interrupt occurred (must be cleared in software)
0 = The INT2 external interrupt did not occur
bit 0 **INT1IF:** INT1 External Interrupt Flag bit, 1 = The INT1 external interrupt occurred (must be cleared in software)
0 = The INT1 external interrupt did not occur

FIGURE 9. 7 INTCON (Interrupt Control) and INTCON3 (Interrupt Control 3) registers

Programming the PIC18F Ext. Interrupts

- Figure 9.8 depicts a simplified diagram showing the PIC18F external interrupt structure in default mode (branches to a single interrupt address vector 0x08). All PIC18F interrupts are disabled upon reset. However, before the PIC18F can accept external interrupts, certain initializations are necessary as follows (see Figure 9.8):

Programming the PIC18F Ext. Interrupts

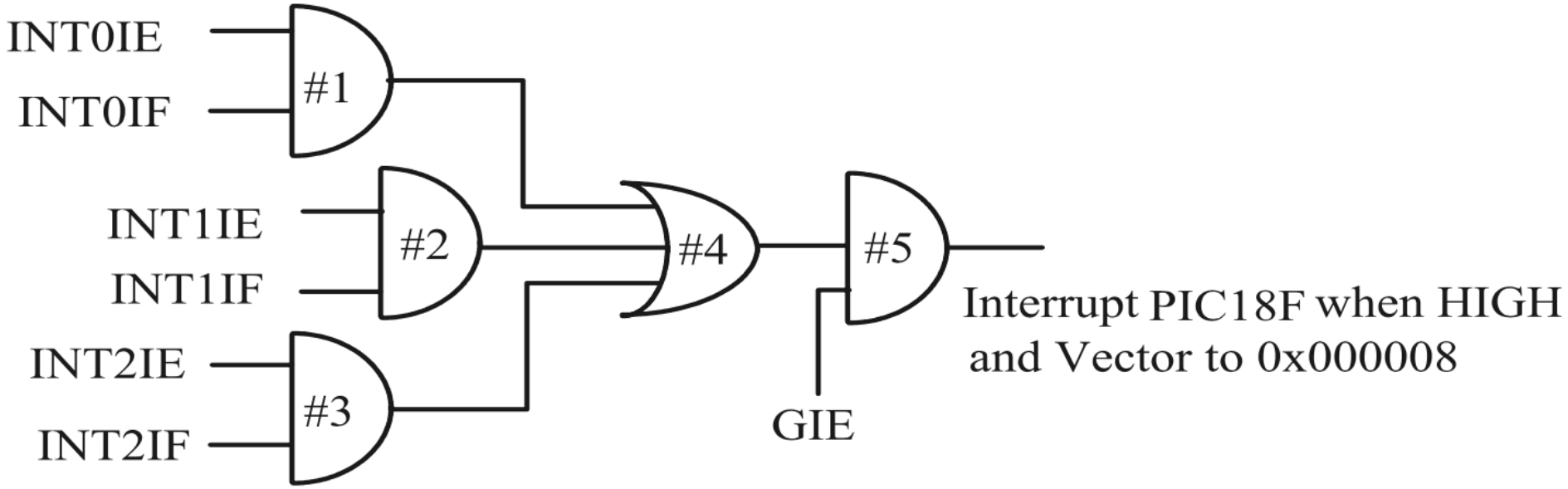


FIGURE 9.8 Simplified schematic for the PIC18F external interrupts for power-on reset

Programming the PIC18F Ext. Interrupts

1. Clear INTxIF for each interrupt to 0. These flags will be set to ones automatically after occurance of interrupts. Note that INT0IF is located in the INTCON register while INT1IF and INT2IF are located in INTCON3 register.
2. Set INTxIE for each interrupt to 1. Note that INT0IE is located in the INTCON register while the INT1IE and INT2IE are contained in the INTCON3 register.
3. Set the GIE (Global Interrupt Enable) bit in the INTCON register to 1.
4. Finally, since the external interrupts (INT0- INT2) are multiplexed with analog inputs (AN8, AN10, and AN12), the ADCON1 register must be configured as digital input.

Programming the PIC18F Ext. Interrupts

Next, consider INT0 in Figure 9.8. According to the steps listed above, the following initializations in PIC18F assembly language are necessary before the PIC18F will be able to accept external interrupt via its INT0 pin:

```
BCF     INTCON, INT0IF  ; Clear INT0IF to 0
BSF     INTCON, INT0IE  ; Set INT0IE to 1
BSF     INTCON, GIE     ; Set GIE to 1
MOVLW   0x0F            ; Configure INT0-INT2 as digital inputs
MOVWF   ADCON1
```

The C-equivalent of the above PIC18F assembly language will be:

```
INTCONbits.INT0IF = 0;      // Clear INT0IF to 0
INTCONbits. INT0IE = 1;     // Set INT0IE to 1
INTCONbits. GIE = 1;        // Set GIE to 1
ADCON1 = 0x0F;              // Configure INT0-INT2 as digital inputs
```

Programming the PIC18F Ext. Interrupts

- The above PIC18F instruction sequence will send a ‘0’ to the output of AND gate #1. This is because INT0IF = 0, INT0IE = 1, and GIE = 1. This will make the output of AND gate #5 as 0. As soon as the PIC18F is interrupted via the INT0 pin, the PIC18F completes the current instruction, and then automatically sets the INT0IF to 1. Hence, the output of AND gate #5 will be ‘1’ in Figure 9.8, and the PIC18F will branch to the interrupt service routine.

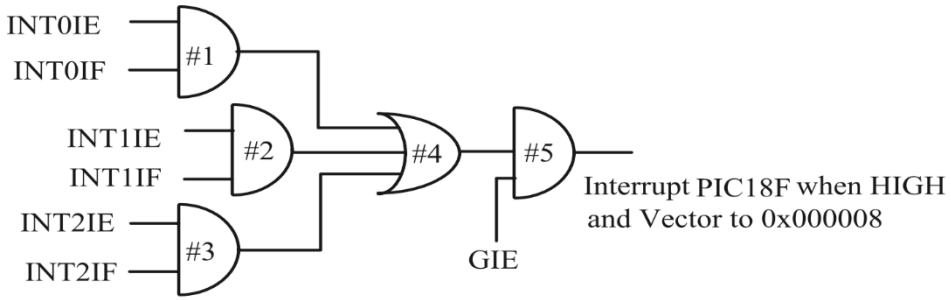


FIGURE 9.8 Simplified schematic for the PIC18F external interrupts for power-on reset

Programming the PIC18F Ext. Interrupts

The following PIC18F assembly instruction sequence will enable all external interrupts:

```
BCF     INTCON, INT0IF    ; INT0IF = 0
BCF     INTCON3, INT1IF   ; INT1IF = 0
BCF     INTCON3, INT2IF   ; INT2IF = 0
BSF     INTCON, INT0IE    ; Enable INT0
BSF     INTCON3, INT1IE   ; Enable INT1
BSF     INTCON3, INT2IE   ; Enable INT2
BSF     INTCON, GIE       ; Enable GIE
MOVLW   0X0F              ; Configure INT0-INT2
MOVWF   ADCON1            ; as digital inputs
```

Programming the PIC18F Ext. Interrupts

- The following C-language statements will enable all external interrupts:

```
INTCONbits.INT0IF=0;          // INT0IF = 0
INTCON3bits.INT1IF=0;         // INT1IF = 0
INTCON3bits.INT2IF=0;          // INT2IF = 0
INTCONbits.INT0IE=1;           // Enable INT0
INTCON3bits.INT1IE=1;          // Enable INT1
INTCON3bits.INT2IE=1;          // Enable INT2
INTCONbits. GIE = 1;           // Enable all inputs
ADCON1=0x0F;                   // Configure INT0-INT2 as digital inputs
```

Programming the PIC18F Ext. Interrupts

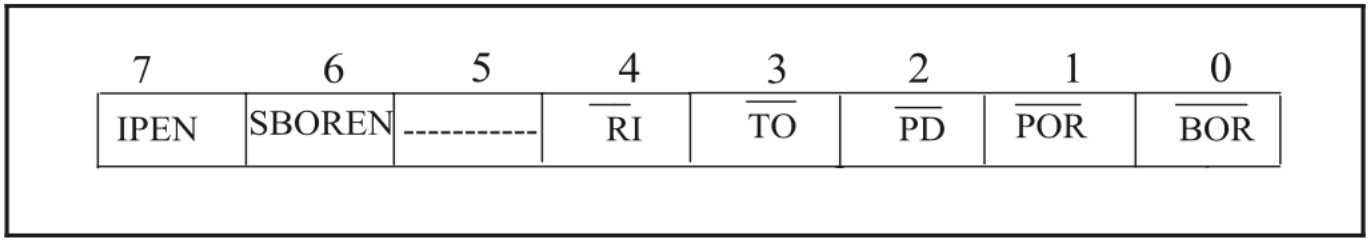
- Note that before executing the last instruction of the interrupt service routine, the user must clear the INT0IF bit using BCF INTCON, INT0IF. The RETFIE instruction at the end of the service routine will automatically set the GIE bit to one enabling all interrupts, and will load the program counter from the hardware stack with the previously pushed value. Note that external interrupts INT1 and INT2 can be explained in a similar manner.

Programming the PIC18F Ext. Interrupts

The PIC18F4321 contains ten registers which are used to control interrupt operation. These registers are:

- RCON (Figure 8.6)
- INTCON (Figure 9.7(a))
- INTCON2 (9.9)
- INTCON3 (Figure 9.7(b))
- PIR1, PIR2 (to be discussed in Chapter 10, Figures 10.8, 10.12)
- PIE1, PIE2 (to be discussed in Chapter 10, Figures 10.9, 10.13)

Programming the PIC18F Ext. Interrupts



Bit 7 = IPEN (Interrupt Pending, 1 = enable, 0 = disable; to be discussed later) **Bit 6** = SBOREN (BOR software enable bit, 1 = enable, 0 = disable) **Bit 5** = Unimplemented (read as '0') **Bit 4** = $\overline{\text{RI}}$ (RESET instruction bit, 1 = RESET instruction executed, 0 = RESET instruction not executed) **Bit 3** = $\overline{\text{TO}}$ (Watchdog Timeout bit, 1 = upon power-up or execution of CLRWDT or SLEEP instruction, 0 = a watchdog timer timeout occurred) **Bit 2** = $\overline{\text{PD}}$ (Power- Down detection bit, 1 = upon power-up or execution of CLRWDT instruction, 0 = execution of SLEEP instruction. **Bit 1** = A $\overline{\text{POR}}$ (Power-on Reset status bit, 1 = A Power-on reset has not occurred, 0 = A Power-on reset has occurred) **Bit 0** = $\overline{\text{BOR}}$ (Brown-out Reset status bit, 1 = A Brown-out reset has not occurred, 0 = A Brown-out reset has occurred)

FIGURE 8. 6 RCON (RESET CONTROL) Register

Programming the PIC18F Ext. Interrupts

7	6	5	4	3	2	1	0	
$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG0	-------	TMR0IP	-----	RBIP	INTCON2

bit 7 **RBPU:** PORTB Pull-up Enable bit
1 = All PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values
bit 6 **INTEDG0:** External Interrupt 0 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 5 **INTEDG1:** External Interrupt 1 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 4 **INTEDG2:** External Interrupt 2 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 3 **Unimplemented:** Read as '0'
bit 2 **TMR0IP:** TMR0 Overflow Interrupt Priority bit, 1 = High priority, 0 = Low priority
bit 1 **Unimplemented:** Read as '0'
bit 0 **RBIP:** RB Port Change Interrupt Priority bit, 1 = High priority, 0 = Low priority

FIGURE 9. 9 INTCON2 Register

Programming the PIC18F Ext. Interrupts

- Registers RCON, INTCON, INTCON2, and INTCON3 are associated with external and port change interrupts. Registers PIR1, PIR2, PIE1 and PIE2 are used by peripheral interrupts.
- The PIC18F4321 interrupts can be classified into **high priority** interrupt levels and **low priority** interrupt levels. The high priority interrupt vector is at address 0x000008 and the low priority interrupt vector is at address 0x000018 in the program memory.

Programming the PIC18F Ext. Interrupts

- Upon power-on reset, the interrupt address vector is 0x000008, and no interrupt priorities are available. Also, IPEN is automatically cleared to 0, and the PIC18F operates as a high-priority interrupt (single interrupt) system. The IPEN bit (bit 7 of the RCON register) of the RCON register can be programmed to assign interrupt priorities. During normal operation, the IPEN bit can be set to one by executing the “BSF RCON, IPEN” in PIC18F assembly or “RCONbits.IPEN = 1;” in C-language.

Programming the PIC18F Ext. Interrupts

- When IPEN = 1, setting the GIEH bit (bit 7 of INTCON register) enables all interrupts that have the priority bit set (high priority), and setting the GIEL bit (bit 6 of INTCON register) enables all low interrupts. When the interrupt flag, interrupt enable bit, and appropriate global interrupt enable bit are set, the interrupt will vector immediately to address 0x000008 or 0x000018, depending on the priority bit setting.

(a) INTCON Register

7	6	5	4	3	2	1	0	
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0I F	INT0I F	RBIF	INTCON

bit 7 **GIE/GIEH:** Global Interrupt Enable bit
When IPEN = 0: 1 = Enables all unmasked interrupts 0 = Disables all interrupts
When IPEN = 1: 1 = Enables all high priority interrupts 0 = Disables all interrupts
bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit
When IPEN = 0: 1 = Enables all unmasked peripheral interrupts 0 = Disables all peripheral interrupts
When IPEN = 1: 1 = Enables all low priority peripheral interrupts 0 = Disables all low priority peripheral interrupts

Programming the PIC18F Ext. Interrupts

- From Figure 9.7(a), when IPEN of RCON register is 0, bits 6 and 7 of the INTCON register become GIE (Global Interrupt Enable) bit and PEIE (Peripheral Interrupt Enable) bit.
- Setting or cleaning the GIE bit via programming will enable or disable all interrupts (INT0, INT1, INT2).
- On the other hand, setting or clearing the PEIE bit will enable or disable all peripheral interrupts.

Programming the PIC18F Ext. Interrupts

(a) INTCON Register

7	6	5	4	3	2	1	0	
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0I F	INT0I F	RBIF	INTCON

bit 7 **GIE/GIEH:** Global Interrupt Enable bit
When IPEN = 0: 1 = Enables all unmasked interrupts 0 = Disables all interrupts
When IPEN = 1: 1 = Enables all high priority interrupts 0 = Disables all interrupts
bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit
When IPEN = 0: 1 = Enables all unmasked peripheral interrupts 0 = Disables all peripheral interrupts
When IPEN = 1: 1 = Enables all low priority peripheral interrupts 0 = Disables all low priority peripheral interrupts
bit 5 **TMR0IE:** TMR0 Overflow Interrupt Enable bit,
1 = Enables the TMR0 overflow interrupt 0 = Disables the TMR0 overflow interrupt
bit 4 **INT0IE:** INT0 External Interrupt Enable bit
1 = Enables the INT0 external interrupt, 0 = Disables the INT0 external interrupt
bit 3 **RBIE:** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt, 0 = Disables the RB port change interrupt
bit 2 **TMR0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software), 0 = TMR0 register did not overflow
bit 1 **INT0IF:** INT0 External Interrupt Flag bit
1 = The INT0 external interrupt occurred (must be cleared in software), 0 = The INT0 external interrupt did not occur
bit 0 **RBIF:** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Programming the PIC18F Ext. Interrupts

(b) **INTCON3 Register**

7	6	5	4	3	2	1	0	
INT2I P	INT1I P	---	INT2IE	INT1IE	---	INT2I F	INT1I F	INTCON3

bit 7 **INT2IP:** INT2 External Interrupt Priority bit, 1 = High priority 0 = Low priority
bit 6 **INT1IP:** INT1 External Interrupt Priority bit, 1 = High priority 0 = Low priority
bit 5 **Unimplemented:** Read as '0'
bit 4 **INT2IE:** INT2 External Interrupt Enable bit, 1 = Enables the INT2 interrupt
0 = Disables the INT2 interrupt
bit 3 **INT1IE:** INT1 External Interrupt Enable bit, 1 = Enables the INT1 interrupt
0 = Disables the INT1 interrupt
bit 2 **Unimplemented:** Read as '0'
bit 1 **INT2IF:** INT2 External Interrupt Flag bit, 1 = The INT2 interrupt occurred (must be cleared in software)
0 = The INT2 external interrupt did not occur
bit 0 **INT1IF:** INT1 External Interrupt Flag bit, 1 = The INT1 external interrupt occurred (must be cleared in software)
0 = The INT1 external interrupt did not occur

FIGURE 9. 7 INTCON (Interrupt Control) and INTCON3 (Interrupt Control 3) registers

Programming the PIC18F Ext. Interrupts

- Table 9.1 shows the three external interrupts of the PIC18F4321 along with the corresponding Interrupt Priority (IP) bit.
- Since an IP (Interrupt Priority) is not assigned to INT0, it always has the high priority level. However, INT1 and INT2 can be programmed as high or low level priorities.

Programming the PIC18F Ext. Interrupts

TABLE 9.1 PIC18F4321 External interrupts along with Interrupt Priority (IP) bits

Interrupt Name	Interrupt Priority (IP) bit	Comment
INT0	Unassigned	Since no interrupt priority bit is assigned, INT0 always has the high priority level.
INT1	INT1IP	Bit 6 of INTCON3 register (Figure 9.7(b)); 1 = high priority, 0 = low priority
INT2	INT2IP	Bit 7 of INTCON3 register (Figure 9.7(b)); 1 = high priority, 0 = low priority

Scenario Analysis

1. **INT0** is High Priority by default.
2. **INT1** is configured as High Priority.
3. **INT2** is configured as Low Priority.

If **INT0**, **INT1**, and **INT2** are triggered simultaneously:

- Both **INT0** and **INT1** are High Priority, but the hardware gives **INT0** a higher priority because its interrupt number is lower.
- **INT2** is Low Priority and will only be serviced after all High Priority interrupts are completed.

Programming the PIC18F Ext. Interrupts

- For example, in order to program INT1 as a high priority interrupt and INT2 as a low priority interrupt, the following PIC18F assembly language instruction sequence can be used:

```
BSF     RCON, IPEN          ; Set IPEN to 1, enable interrupt level
BSF     INTCON, GIEL        ; Set low priority levels
BSF     INTCON, GIEH        ; Set high priority levels
BSF     INTCON3, INT1IP     ; INT1 has high level
BCF     INTCON3, INT2IP     ; INT2 has low level
```

The equivalent C-statements will be as follows:

```
RCONbits.IPEN = 1;          // Set IPEN to 1, enable interrupt level
INTCONbits.GIEL = 1;        // Set low priority levels
INTCONbits.GIEH = 1;        // Set high priority levels
INTCON3bits.INT1IP = 1;     // INT1 has high level
INTCON3bits.INT2IP = 0;     // INT2 has low level
```

Programming the PIC18F Ext. Interrupts

Note that INT0 along with either INT1 or INT2 can be used as two interrupts in an application. Suppose INT0 and INT1 are used in a two-interrupt systgem. Since INT0 has always the high priority, the INT1 must be configured as the low priority. The following PIC18F assembly language instruction sequence will accomplish this:

```
BSF     RCON, IPEN          ; Enable priority interrupt
BSF     INTCON, GIEL        ; Enable global low interrupt
BSF     INTCON, GIEH        ; Enable global high interrupt
BCF     INTCON3, INT1IP     ; Configure INT1 as low priority
```

The equivalent C-statements are as follows:

```
RCONbits.IPEN = 1;          // Enable priority interrupt
INTCONbits.GIEL = 1;        // Enable global low interrupt
INTCONbits.GIEIH = 1;       // Enable global high interrupt
INTCON3bits.INT1IP = 0;     // Configure INT1 as low priority
```

Programming the PIC18F Ext. Interrupts

- Finally, the “PORTB Interrupt-on-Change” is used in keyboard interfacing. The PIC18F4321 provides four interrupt-on-change pins (KB10 through KB13). These pins are multiplexed with bits RB4 through RB7 of PORTB. An input change (HIGH to LOW or LOW to HIGH) on one or more of these KB10-KB13 pins will set a single flag bit, RBIF (bit 0 of INTCON register) to one. Note that a single flag bit is assigned to all four interrupts.

Programming the PIC18F Ext. Interrupts

- The interrupt can be enabled/disabled by setting/clearing the single enable bit, RBIE (bit 3 of INTCON register). Interrupt priority for PORTB interrupt-on-change is determined by the value contained in the interrupt priority bit RBIP (bit 0 of INTCON2 register of Figure 9.9). Each bit associated with this interrupt can be set or cleared as appropriate via programming the corresponding register.

Programming the PIC18F Ext. Interrupts

7	6	5	4	3	2	1	0	
$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG0	-------	TMR0IP	-----	RBIP	INTCON2

bit 7 **RBPU:** PORTB Pull-up Enable bit
1 = All PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values
bit 6 **INTEDG0:** External Interrupt 0 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 5 **INTEDG1:** External Interrupt 1 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 4 **INTEDG2:** External Interrupt 2 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 3 **Unimplemented:** Read as '0'
bit 2 **TMR0IP:** TMR0 Overflow Interrupt Priority bit, 1 = High priority, 0 = Low priority
bit 1 **Unimplemented:** Read as '0'
bit 0 **RBIP:** RB Port Change Interrupt Priority bit, 1 = High priority, 0 = Low priority

FIGURE 9. 9 INTCON2 Register

Ext. Interrupts Using Assembly

- **Setting the triggering levels of INTn Pin Interrupts**
- External interrupts on the INT0, INT1, and INT2 pins are edge-triggered. Upon power-on reset, each of these external interrupts (INT0, INT1, INT2) are activated by a rising edge pulse (LOW to HIGH). The PIC18F has the flexibility of changing the triggering levels for these interrupts to falling edge pulses (HIGH to LOW).

Ext. Interrupts Using Assembly

- This can be accomplished by programming bits 4 through 6 of the INTCON2 register (Figure 9.9). For example, the instruction BCF INTCON2, INTEDG0 will change INT0 from a rising-edge triggered to falling-edge triggered interrupt.

7	6	5	4	3	2	1	0	
$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG0	-------	TMR0IP	-----	RBIP	INTCON2

bit 7 **RBPU:** PORTB Pull-up Enable bit
1 = All PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values
bit 6 **INTEDG0:** External Interrupt 0 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 5 **INTEDG1:** External Interrupt 1 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge
bit 4 **INTEDG2:** External Interrupt 2 Edge Select bit, 1 = Interrupt on rising edge, 0 = Interrupt on falling edge

Ext. Interrupts Using Assembly

- **Return from Interrupt Instruction**
- The “RETFIE s” instruction is normally used at the end of the service routine. ‘s’ can be 0 or 1. When s = 0, the PIC18F RETFIE 0 or simply RETFIE instruction pops the contents of the program counter previously pushed before going to the service routine, enables the global interrupt enable bit, and returns control to the appropriate place in the main program.

Ext. Interrupts Using Assembly

- **Return from Interrupt Instruction**
- For high priority interrupts only eight 16-bit words from addresses 0x000008 through 0x000017 are available for the service routine since the address 0x000018 is assigned to the low priority interrupt. This means that a small service routine can be written starting at address 0x08. However, for large service routines, the programmer may have to write a service routine by jumping to another location using the GOTO instruction.

Ext. Interrupts Using C

- MPLAB C18 compiler does not automatically place an interrupt service routine at user-defined interrupt address vector. Hence, the GOTO instruction is normally placed at the interrupt address vector 0x08 for transferring control to the interrupt service routine. The C18 compiler does not have any feature to jump from one address to another. However, the C18 compiler has "_asm" and "_endasm" directives to place "GOTO" instruction at address 0x08 to jump to the interrupt service routine.

Ext. Interrupts Using C

- **Specifying starting address of the service routine using the C18 compiler**
- Using the C18 compiler, the interrupt address vectors, must be defined and placed at the appropriate vector addresses using the “#pragma code” directive. For example, the programmer can use the statement “#pragma code begin=0x08” to place begin at an address 0x08.
- Hence, the C statement “#pragma code int_vect = 0x08” will assign the address 0x08 to label “int_vect”.

Ext. Interrupts Using C

- **A typical structure for interrupt programs using C**

```
#include  <P18F4321.h>
void        ISR (void);
#pragma code Int=0x08     // At interrupt code jumps here
void  Int(void)
{
_asm                          // Using assembly language
GOTO ISR
_endasm
}
#pragma code                  //  Main program
void main( )
{
// Typically configure ports, enable INT0IE, clear INT0IF
while(1){                     // Wait in an infinite loop for the interrupt to occur
}

#pragma    interrupt  ISR
void ISR(void)                // Start of Interrupt service routine
          {
INTCONbits.INT0IF =0;  // clear INT0IF, and then write the service
          }
}
```

Ext. Interrupts Using C

- In the above, the “#pragma code” directive will place “Int” at the specific location 0x08 in program memory. The main program typically configures ports, enables interrupt, clears the interrupt flag bit and waits in an infinite loop for the interrupt to occur. When the interrupt occurs, the program will automatically jump to memory location 0x08.

Ext. Interrupts Using C

Example 9.2 ***Interrupt I/O*** Assume that the PIC18F4321 microcontroller shown in Figure 9.10 is required to perform the following:

If Vx > Vy , turn the LED ON if the switch is open; otherwise, turn the LED OFF.

(a) Write a PIC18F assembly language program to accomplish this using INT0 external interrupt. Use port A for the LED and switch as above. Write the main program at address 0x100 in PIC18F assembly language and the service routine at address 0x08. Connect INT0 pin to the output of the comparator.

The main program will initialize hardware stack pointer (STKPTR) to 0x10, configure PORTB and PORTD, clear INT0IF and enable GIE and INT0IE. Write the service routine in PIC18F assembly language which will clear the INT0F flag, input the switch, output to LED, and then return to the main program.

(b) Assume that the PIC18F4321 microcontroller shown in Figure 9.10 is required to perform the following:

If Vx > Vy , turn the LED ON if the switch is open; otherwise, turn the LED OFF. Write the main program in C language which will initialize PORTB and PORTD, and then wait for interrupt in an infinite loop. Also, write a service routine in C language.

Ext. Interrupts Using C

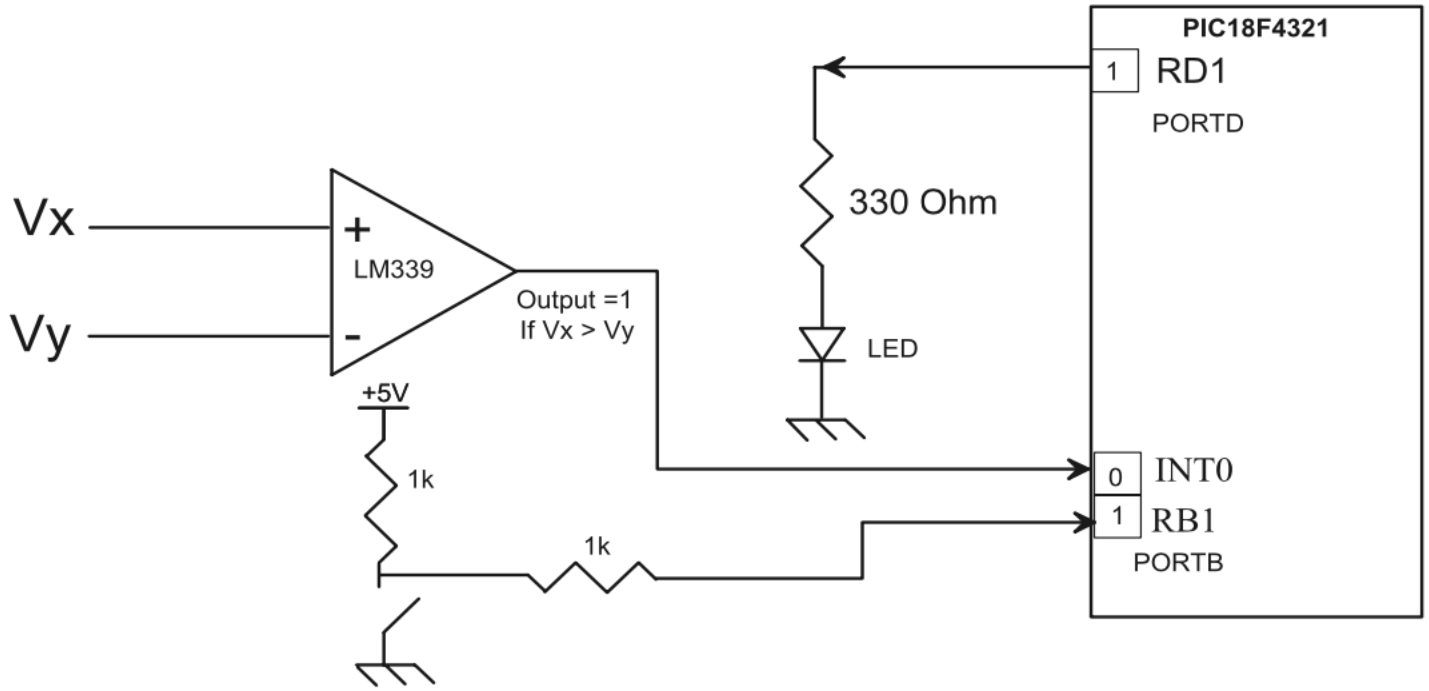


FIGURE 9.10 Figure for Example 9.2

Ext. Interrupts Using C

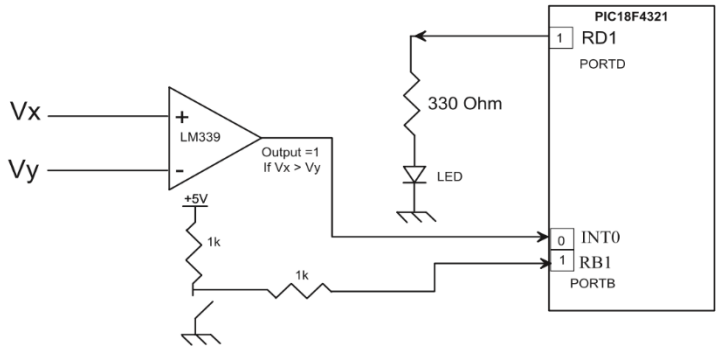
(a) Figure 9.10 shows the relevant connections of the comparator to the PIC18F4321 using interrupt I/O. Note that the comparator output is connected to bit 0 of PORTB to be used as an INT0 pin.

In this example, by using ADCON1 register, bit 0 of PORTB can be configured as digital input to accept interrupt via INT0. The INT0IE bit of the INTCON register must be set to one in order to enable the external interrupt along with GIE to enable global interrupts.

The PIC18F assembly language program using external interrupt INT0 is provided in the following:

```
INCLUDE <P18F4321.INC>
ORG     0                   ; RESET
BRA     MAIN_PROG
```

Ext. Interrupts Using C




```
; MAIN PROGRAM
                ORG     0x00100         ; MAIN PROGRAM
MAIN_PROG       MOVLW   0x10            ; Initialize STKPTR to 0x10
                MOVWF   STKPTR
                BCF     TRISD,TRISD1    ; Configure bit 1 OF PORTD as output
                MOVLW   0x0F            ; Configure bit 0 of PORTB as INT0
                MOVWF   ADCON1          ; and bit 1 as input
                BCF     INTCON, INT0IF  ; Clear INT0IF flag
                BSF     INTCON, GIE     ; Enable global interrupts
                BSF     INTCON, INT0IE  ; Enable the external interrupt
WAIT            BRA     WAIT            ; Wait for interrupt
                BRA     MAIN_PROG       ; Repeat
; INTERRUPT SERVICE ROUTINE
                ORG     0x000008        ; Interrupt Address Vector
INT_SERV        MOVFF   PORTB, PORTD    ; Output switch status to turn LED ON/OF
                BCF     INTCON, INT0IF  ; Clear the external interrupt flag to avoid
                                        ; double interrupt
                RETFIE                  ; Enable interrupt and return
                END
```

```
#include <P18F4321.h>
void COMP_ISR (void);
#pragma code COMP_Int=0x08  //At interrupt code jumps here
void COMP_Int(void)
{
_asm                // Using assembly language
GOTO  COMP_ISR
_endasm
}
#pragma code
void main( )                                    // start of the main program
{
TRISD=0x00;                                     // PORTD is output
ADCON1=0x0F;                                    // Configure RB1 and INT0 as digital input
INTCONbits.INT0IE=1;                            // Enable external interrupt
INTCONbits.INT0IF=0;                            // Clear the external interrupt flag
INTCONbits.GIE=1;                               // Enable global interrupts
PORTD=0;                                          // Turn off LED;
while(1){   // wait in an infinite loop for the interrupt to occur
        PORTD=0;                                // LED is off
        }
}
#   pragma  interrupt COMP_ISR
void COMP_ISR(void)  //start of the Comparator interrupt service routine
        {
        INTCONbits.INT0IF=0;                    // Clear external interrupt flag
                while(PORTBbits.RB0= =1){ // Check if comparator is high
                        PORTD = PORTB;          // Move PORTB into PORTD
        }
}
```

Ext. Interrupts Using C

- **Example 9.3** In Figure 9.11, if Vx > Vy, the PIC18F4321 is interrupted via INT0. On the other hand, opening the switch will interrupt the microcontroller via INT1. Note that in the PIC18F4321, INT0 has the higher priority than INT1.

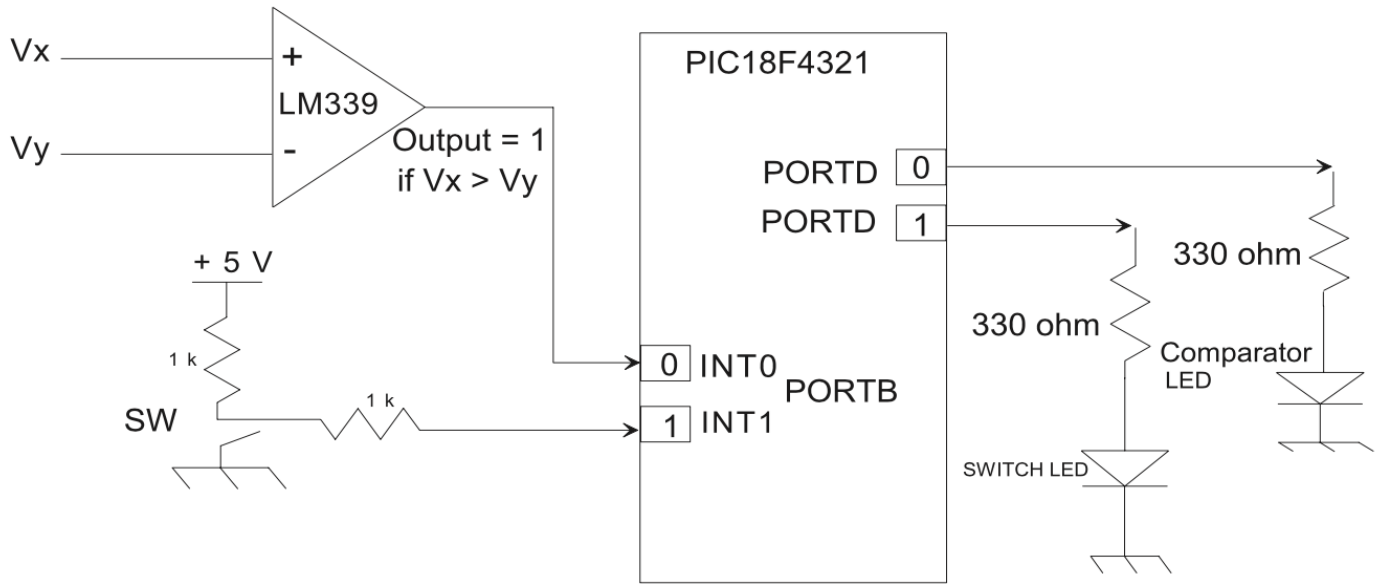


FIGURE 9.11 Figure for Example 9.3

Ext. Interrupts Using C

(a) Write the main program in PIC18F assembly language at address 0x100 that will perform the following:

- Initialize STKPTR to 0x12
- Configure PORTB as interrupt inputs.
- Clear interrupt flag bits of INT0IF in INTCON reg and INT1IF in INTCON3 reg
- Configure PORTD (bits 0, 1) as outputs for LEDs
- Set INT1 as a low priority interrupt by clearing INT1IP bit in INTCON3 reg
- Enable priority interrupts by setting IPEN bit in RCON reg
- Enable global HIGH and LOW interrupts by setting GIEH and GIEL bits in INTCON reg
- Turn both LEDs at PORTD OFF.
- Wait in an infinite loop, and wait for one or both interrupts to occur.
- Turn both LEDs at PORTD OFF (Comparator LED at bit 0 of PORTD and Switch LED at bit 1 of PORTD)
- Wait in an infinite loop for one or both interrupts to occur.

Also, write a service routine in PIC18F assembly language at address 0x200 for the high priority interrupt (INT0) that will perform the following:

- Clear interrupt flag for INT0 (INT0IF)
- Check to see if the comparator output is still 1. If it is, turn the LED ON at bit 0 of PORTD.

Finally, write a service routine in PIC18F assembly language at address 0x300 for the low priority interrupt (INT1) that will perform the following:

- Clear interrupt flag for INT1 (INT1IF)
- Turn LED ON at bit 1 of PORTD

Ext. Interrupts Using C

(b) In Figure 9.11, if Vx > Vy, the PIC18F4321 is interrupted via INT0. On the other hand, opening the switch will interrupt the microcontroller via INT1. Note that in the PIC18F4321, INT0 has the higher priority than INT1. Write the main program in C that will perform the following:

- Configure PORTB as interrupt inputs.
- Clear interrupt flag bits of INT0IF in INTCON reg and INT1IF in INTCON3 reg
- Configure PORTD (bits 0, 1) as outputs for LEDs
- Set INT1 as low priority interrupt by clearing INT1IP bit in INTCON3 reg
- Enable priority interrupts by setting IPEN bit in RCON reg
- Enable global HIGH and LOW interrupts by setting GIEH and GIEL bits in INTCON reg
- Turn both LEDs at PORTD OFF.
- Wait in an infinite loop, and wait for one or both interrupts to occur

Also, write a service routine in C-language for the high priority interrupt (INT0) that will check to see if the comparator output is still 1. If it is, turn LED at bit 0 of PORTD ON. If the comparator output is 0, return.

Finally, write a service routine in C language for the low priority interrupt (INT1) that will perform the following:

- Clear interrupt flag for INT1
- Check to see if the switch is still 1. If it is, turn LED at bit 1 of PORTD ON. If the switch input is 0, return.

Ext. Interrupts Using C

- Solution. With interrupt priority, the user has the option to have the interrupts declared as either low or high interrupts. If at anytime the low and high priority interrupts occur at the same time, the microcontroller will always service the high priority interrupt. In the above example, the comparator is set as the high priority and the switch is set as the low priority, so if both interrupts are triggered simultaneously, then only the LED associated with the comparator will be turned ON.

(a) PIC18F assembly language program:

```
INCLUDE <P18F4321.INC>
; RESET
                ORG     0               ; Reset vector
                BRA     MAIN            ; Jump to main program
; HIGH PRIORITY INTERRUPT ADDRESS VECTOR
                ORG     0x0008          ; High priority interrupt
                BRA     HIGH_INT_ISR    ; Jump to service routine for the
                                        ; comparator
; LOW PRIORITY INTERRUPT ADDRESS VECTOR
                ORG     0x0018          ; Low priority interrupt
                BRA     LOW_INT_ISR     ; Jump to service routine for the
                                        ; comparator
                                        ; MAIN PROGRAM
                ORG     0x0100
MAIN            MOVLW   0x12            ; Initialize STKPTR to 0x12
                MOVWF   STKPTR
                CLRF    TRISD           ; PORTD is output
                MOVLW   0x0F            ; Configure ADCON1 to set up
                MOVWF   ADCON1          ; INT0 and INT1 as digital inputs
                BSF     INTCON, INT0IE  ; Enable the external interrupt INT0
                BSF     INTCON3,INT1IE  ; Enable the external interrupt INT1
                BCF     INTCON,INT0IF   ; Clear the INT0IF flag
                BCF     INTCON3,INT1IF  ; Clear the INT1IF flag
                BCF     INTCON3, INT1IP ; Configure INT1 as low priority
                BSF     RCON, IPEN      ; Enable priority interrupt
```

```
                BSF     INTCON, GIEH    ; Enable global high interrupts
                BSF     INTCON, GIEL    ; Enable global low interrupts
OVER            CLRF    PORTD           ; Turn both LED's  off
                BRA     OVER            ; Wait for interrupt

; SERVICE ROUTINE FOR HIGH PRIORITY
                ORG     0x200
HIGH_INT_ISR    BCF     INTCON,INT0IF   ; Clear the interrupt flag
CHECK           BTFSS   PORTB,RB0       ; Check to see if comparator output is
                                        ; still one
                RETFIE
                MOVLW   0x01            ; Turn on LED at bit 0 of PORTD
                MOVWF   PORTD
                BRA     CHECK
; SERVICE ROUTINE FOR LOW PRIORITY
                ORG     0x300
LOW_INT_ISR     BCF     INTCON3, INT1IF ; Clear the interrupt flag
CHECK1          BTFSS   PORTB,RB1       ; Check to see if switch is still one
                RETFIE
                MOVLW   0x02            ; Turn on LED at bit 1 of PORTD
                MOVWF   PORTD
                BRA     CHECK1
                END
```

Ext. Interrupts Using C

- **C Program.**
- Note that the external interrupt INT0 can only be a high priority interrupt. Hence, INT0 is connected to the comparator output while the switch is connected to INT1 since it has the low priority. At the end of the code provided below, it can be seen that there are two interrupt service routines, HP_COMP_ISR and LP_SWITCH_ISR, which are the high priority and low priority service routines.

Ext. Interrupts Using C

```
#include <P18F4321.h>
void HP_COMP_ISR (void);
void LP_SWITCH_ISR(void);

#pragma code High_Priority_COMP_Int=0x08  //High interrupt code jumps here
void COMP_Int(void)
{
_asm  //Using assembly language
GOTO  HP_COMP_ISR  ( )
```

```
_endasm
}
#pragma code Low_Priority_SWITCH_Int=0x018  //Low interrupt code jumps here
void Switch_Int(void)
{
_asm  //Using assembly language
GOTO  LP_SWITCH_ISR ( )
_endasm
}
#pragma code
void main( )
{
TRISBbits.TRISB0=1;                    // Set bit 0 of PORTB as input
TRISBbits.TRISB1=1;                    // Set bit 1 of PORTB as input
TRISD=0x00;                            // PORTD is output
ADCON1=0x0F;                           // Configure PORTB to be digital input
INTCONbits.INT0IE=1;                   // Enable external interrupt INT0
INTCON3bits.INT1IE=1;                  // Enable external interrupt INT1
INTCONbits.INT0IF=0;                   // Clear INT0IF  interrupt flag bit for INT0
INTCON3bits.INT1IF=0;                  // Clear INT1IF interrupt flag bit for INT1
INTCON3bits.INT1IP=0;                  // Configure INT1 as low priority interrupt
RCONbits.IPEN=1;                       // Enable priority interrupts
INTCONbits.GIEH=1;                     // Enable global high priority interrupts
INTCONbits.GIEL=1;                     // Enable global low priority interrupts
PORTD=0;                               // Turn off LED;
```

```
while(1){
        PORTD=0;                        // LED is off
}

}                                       // End of main
#pragma  interrupt  HP_COMP_ISR
void HP_COMP_ISR(void){                 // High priority interrupt service
        INTCONbits.INT0IF=0;            // Clear external interrupt flag
        while(PORTBbits.RB0==1)         // Check if comparator is high
{
              PORTD=0x01;               // Turn on LED
        }
}                                       // End of HP_COMP_ISR

#pragma   interruptlow  LP_SWITCH_ISR
void LP_SWITCH_ISR(void){               // Low priority interrupt service
        INTCON3bits.INT1IF=0;           // Clear external interrupt flag
        while(PORTBbits.RB1==1)         // Check if switch is still on
        {
        PORTD=0x02;                     // Turn on LED
        }

}                                       // End of LP_SWITCH_ISR
```

Accessing On-chip peripheral devices

- There are several PIC18F on-chip peripheral I/O devices. Some of these on-chip peripherals include ADC (Analog to Digital) converter and Hardware timers. Each peripheral device is provided with a Peripheral Interrupt Flag such as the ADIF bit for the ADC. For polled I/O, this flag bit (ADIF for ADC) can be cleared to 0 first via programming. The PIC18F can then be programmed to start the ADC. Upon completion of the A/D conversion, the PIC18F automatically sets the ADIF bit to 1.

Accessing On-chip peripheral devices

- Using polled I/O, the PIC18F can be programmed to wait in a loop for the ADIF to go to HIGH. For interrupt I/O, the PIC18F is provided with a few more peripheral interrupt bits in addition to ADIF and GIE bits. Figure 9.12 shows a simplified diagram of a typical peripheral device such as the ADC along with these bits.

Accessing On-chip peripheral devices

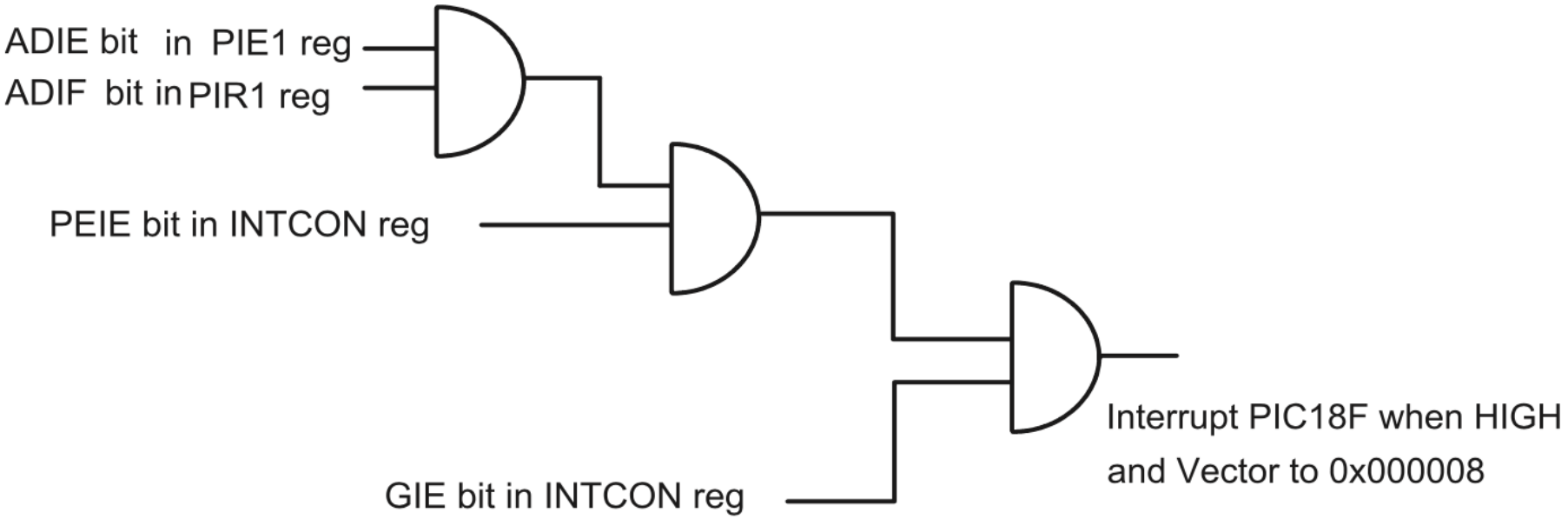


FIGURE 9.12 Interrupt-driven on-chip Peripheral device such as the ADC

Accessing On-chip peripheral devices

- As with polled I/O, the ADIF bit in the PIR1 in Figure 9.12 can be first cleared to 0. The ADIE, PEIE and GIE bits can then be set to ones by programming. Once the ADC is started by the PIC18F via programming, the ADIF is set to one indicating completion of A/D conversion. This will interrupt the PIC18F (Figure 9.12). The PIC18F will jump to interrupt service routine to input the ADC output.

PIC18F Interface to a typical LCD

- The seven-segment LEDs are easy to use, and can only display numbers and limited characters. An LCD is very useful for displaying numbers, and several ASCII characters along with graphics. Furthermore, the LCD consumes low power. Because of inexpensive price of the LCD these days, they have been becoming popular. The LCD's are widely used in copiers, fax machines, laser printers, networking equipment such as routers, and storage devices.

PIC18F Interface to a typical LCD

- Figure 9.13 shows a pictorial view of Hitachi HD44780 LCD along with the pins. The Hitachi HD44780 is a dot matrix LCD manufactoured in several different configurations. Common sizes can display up to one 8-character line or two 8-character lines on a display screen. The HD 44780 contains 16 pins (Figure 9.13).

PIC18F Interface to a typical LCD

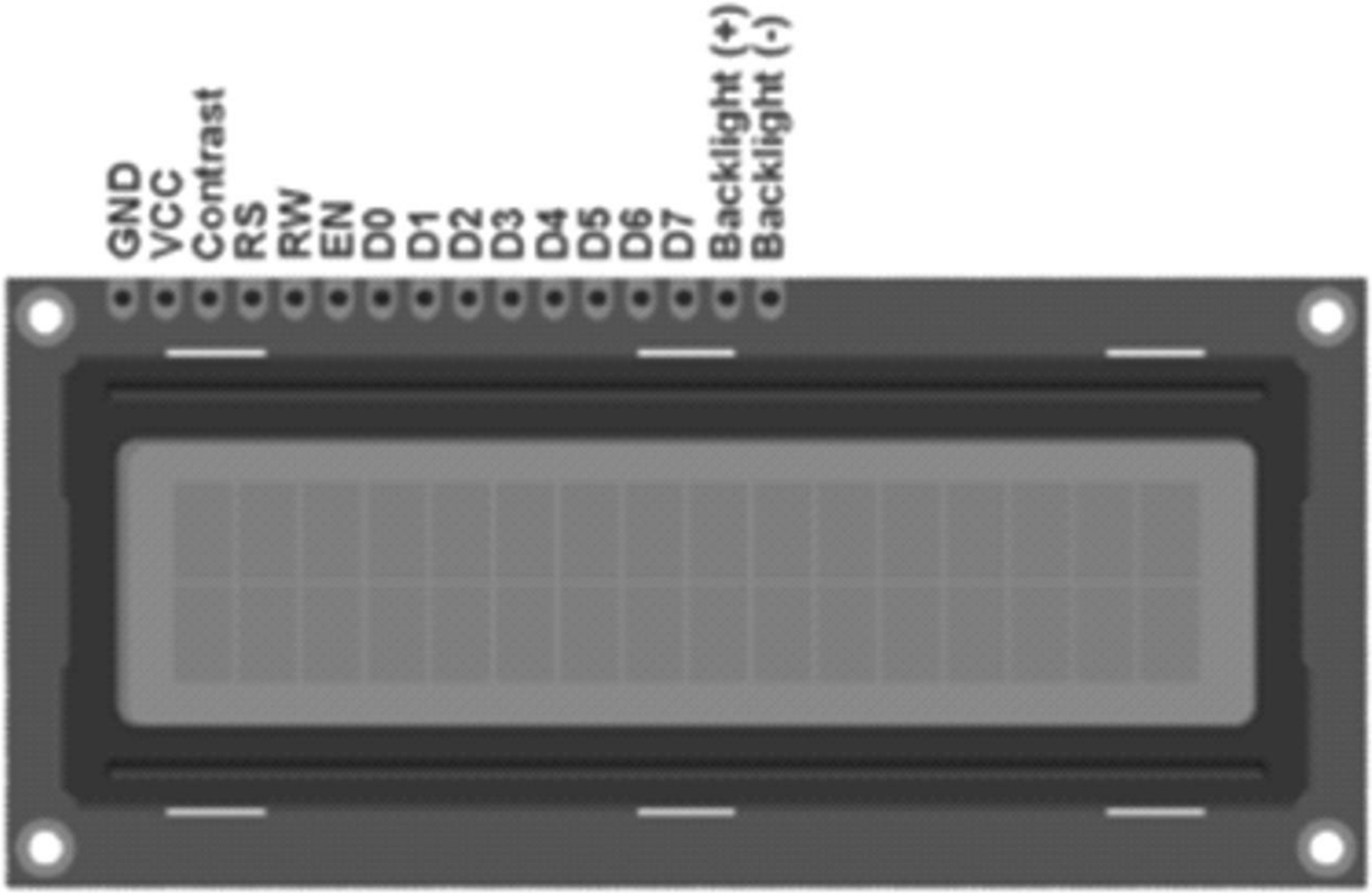


FIGURE 9.13 Hitachi HD44780 LCD and the pinout

PIC18F Interface to a typical LCD

- Pin 1 (GND): Connect to ground
- Pin 2 (VCC): Connect to +5V
- Pin 3 (Contrast): Connect to the middle terminal of a potentiometer (可變電阻); the other two terminals of the potentiometer should be connected to +5V and ground. The contrast of the LCD is changed by varying the value of the potentiometer.

PIC18F Interface to a typical LCD

- Pin 4 (RS- Register Select): RS = 0 means commands while RS = 1 indicates data to be displayed on the LCD screen.
- Pin 5 (RW-Read Write): RW = 1 for reading data from the LCD registers while RW = 0 for writing data into the LCD registers.
- Pin 6 (EN-Enable): 8-bit data or command is latched on the data pins (D0-D7) when EN goes from HIGH to LOW for at least 450 nsec.
- Pins 7-14 (D0-D7): Eight data pins

PIC18F Interface to a typical LCD

- Pin 15 (Backlight+), Pin 16 (Backlight-): Connect Backlight+ to 5V and Backlight- to the ground. The HD44780 is a character LCD, and displays texts only. This display is provided with LED backlight. The nominal operating voltage for LED backlights is 5V at full brightness.

PIC18F Interface to a typical LCD

- The HD44780 is an ASCII character-based alphanumeric LCD display and provides text displays along with LED backlight. It contains 16 pins. The VCC pin is connected to +5V and the VSS pin is connected to ground. The contrast pin is used to control the brightness of the display. The contrast pin is connected to a potentiometer with a value between 2k and 6k. The eight data pins (D0-D7) are used to input data and commands for displaying the desired message on the screen.

PIC18F Interface to a typical LCD

- The three control pins, EN, R/W, and RS, allow the user to let the display know what kind of data is sent. The EN pin latches the data from the D0-D7 pins into the LCD display. Data on D0-D7 pins will be latched on the trailing edge (high-to-low) of the EN pulse. The EN pulse must be at least 450 ns wide; no delay routine is needed because instructions to send a ‘1’ and then a ‘0’ will accomplish this. The R/W (read/write) pin, allows the user to either write to the LCD or read data from the LCD. In this example, the R/W pin will always be zero since only a string of data is written to the LCD. The R/W pin is set to one for reading data from the LCD.

PIC18F Interface to a typical LCD

- The command or data can be outputted to the LCD in two ways:
  1. One way is to provide time delays for a few milliseconds before outputting the next command or data.
  2. The second approach utilizes a busy flag in a register internal to the LCD to determine whether the LCD is free for the next data or command.
     - For example, in order to display characters one at a time, the LCD must be read by outputting a HIGH on the R/W pin.
     - The busy flag can be checked to ensure whether the LCD is busy or not before outputting another string of data. Note that the busy flag can thus be used instead of using time delays.

PIC18F Interface to a typical LCD

- Finally, the RS (Register Select) pin is used to determine whether the user is sending command or data. The LCD contains two 8-bit internal registers. They are command register and data register.
  - When RS = 0, the command register is accessed, and typical LCD commands such as clear cursor left (hex code 0x04) can be used. Table 9.2 shows a list of some of the commands.
  - Note that the busy flag is bit 7 of the LCD's command register. The busy bit can be read by outputting 0 to RS pin, 1 to R/W pin, and a leading edge (LOW to HIGH) pulse to the EN pin.

PIC18F Interface to a typical LCD

TABLE 9.2 Typical LCD commands along with 8-bit codes in hex

Hex	Command
0x01	Clear the screen
0x02	Return home
0x04	Shift cursor to left
0x06	Shift cursor to right
0x05	Shift display to right
0x07	Shift display to left
0x08	Display off, cursor off
0x0A	Display off, cursor on
0x0C	Display on, cursor off
0x0E	Display on, cursor blinking
0x10	Shift cursor position to the left
0x14	Shift cursor position to the right
0x80	Move cursor to the start of the first line

PIC18F Interface to a typical LCD

- When attempting to send data or commands to the LCD, the user must make sure that the values of EN, R/W, and RS are correct along with appropriate timing. For example, in order to send the 8-bit command code to the LCD, a PIC18F assembly or a C-language program can be written to perform the following steps:

PIC18F Interface to a typical LCD

- For sending the 8-bit command code to the LCD, a PIC18F assembly or a C- language program can be written to perform the following steps:
  - Output the command value to the PIC18F4321 I/O port that is connected to the LCD's D0- D7 pins.
  - Send a '0' to RS pin and a '0' to R/W pin.
  - Send a '1' and then a '0' to the EN pin to latch the LCD's D0-D7 code.
- For sending data to the LCD, the PIC18F assembly or C-program can be written to perform the following steps:
  - Output data from PORTD of the PIC18F4321 to LCD's D0-D7 pins.
  - Send a '1' to RS pin and a '0' to the R/W pin.
  - Send a '1' and then a '0' to the EN pin to latch the data.

PIC18F Interface to a typical LCD

EXAMPLE 9.4 Figure 9.14 shows the PIC18F4321's interface to the Hitachi HD44780 LCD display. It is desired to display the phrase "Switch Value:" along with the numeric BCD value (0 through 9) of the four switch inputs. Four switches are connected to bits 0 through 3 of PORTC. The D0-D7 pins of the LCD are connected to bits 0 through 7 of PORTD. The RS, R/$\overline{W}$, and EN pins of the LCD are connected to bits 0, 1, and 2 of PORTB of PIC18F4321.

Use time delay rather than the busy bit before outputting the command or data to the LCD. Also, assume 1-MHz default crystal frequency for the PIC18F4321.

(a) Write a PIC18F assembly language program to accomplish this.

(b) Write a C language prograr

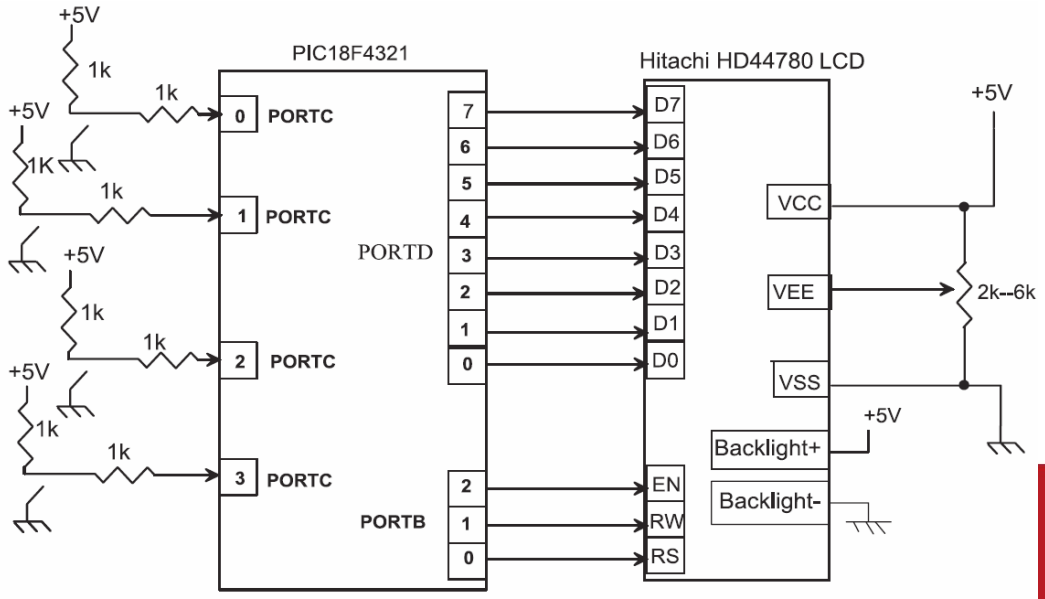


FIGURE 9.14 Figure for Example 9.4

PIC18F Interf

Hex	Command
0x01	Clear the screen
0x02	Return home
0x04	Shift cursor to left
0x06	Shift cursor to right
0x05	Shift display to right
0x07	Shift display to left
0x08	Display off, cursor off
0x0A	Display off, cursor on
0x0C	Display on, cursor off
0x0E	Display on, cursor blinking
0x10	Shift cursor position to the left
0x14	Shift cursor position to the right
0x80	Move cursor to the start of the first line

(a) Three subroutines are use
one for LCD data. Since s
in the main program with
are configured as output
The PIC18F assembly la

```
        INCLUDE <P18F4321.INC>
        ORG     0x100       ; Start of the MAIN program

MAIN    MOVLW   0x10        ; Initialize STKPTR with arbitrary value of 0x10
        MOVWF   STKPTR
        CLRF    TRISD       ; PORTD is output
        CLRF    TRISB       ; PORTB is output
        SETF    TRISC       ; PORTC is input
        CLRF    PORTB       ; rs=0 rw=0 en=0
        MOVLW   D'5'        ; 10 ms delay
        CALL    DELAY
        MOVLW   0x0C        ; Display on, Cursor off
        CALL    CMD
        MOVLW   D'5'        ; 10 ms delay
        CALL    DELAY
```

```
MOVLW   0x01
CALL    CMD         ; Clear Display
MOVLW   D'5'        ; 10 ms delay
CALL    DELAY
MOVLW   0x06        ; Shift Cursor to the right
CALL    CMD
MOVLW   D'5'        ; 10 ms delay
CALL    DELAY
MOVLW   0x80        ; Move cursor to the start of the first line
CALL    CMD
MOVLW   D'5'        ; 10 ms delay
CALL    DELAY
MOVLW   A'S'        ; Send ASCII S
CALL    LCDDATA
MOVLW   A'w'        ; Send ASCII w
CALL    LCDDATA
MOVLW   A'i'        ; Send ASCII i
CALL    LCDDATA
MOVLW   A't'        ; Send ASCII t
CALL    LCDDATA
MOVLW   A'c'        ; Send ASCII c
CALL    LCDDATA
MOVLW   A'h'        ; Send ASCII h
CALL    LCDDATA
MOVLW   A' '        ; Send ASCII space
CALL    LCDDATA
MOVLW   A'V'        ; Send ASCII V
```

Hex	Command
0x01	Clear the screen
0x02	Return home
0x04	Shift cursor to left
0x06	Shift cursor to right
0x05	Shift display to right
0x07	Shift display to left
0x08	Display off, cursor off
0x0A	Display off, cursor on
0x0C	Display on, cursor off
0x0E	Display on, cursor blinking
0x10	Shift cursor position to the left
0x14	Shift cursor position to the right
0x80	Move cursor to the start of the first l

PIC18F Interface to a typical LCD

```
        CALL    LCDDATA
        MOVLW   A'a'        ; Send ASCII a
        CALL    LCDDATA
        MOVLW   A'l'        ; Send ASCII l
        CALL    LCDDATA
        MOVLW   A'u'        ; Send ASCII u
        CALL    LCDDATA
        MOVLW   A'e'        ; Send ASCII e
        CALL    LCDDATA
        MOVLW   A':'        ; Send ASCII :
        CALL    LCDDATA
AGAIN   MOVF    PORTC, W    ; Move switch value to WREG
        ANDLW   0x0F        ; Mask lower 4 bits
        IORLW   0x30        ; Convert to ASCII data by ORing with 0x30
        CALL    LCDDATA     ; Display switch value on screen
        MOVLW   0x10        ; Shift cursor to left
        CALL    CMD
```

PIC18F Interface to a typical LCD

```
            BRA     AGAIN
CMD         MOVWF   PORTD     ; Command is sent to PORTD
            MOVLW   0x04
            MOVWF   PORTB     ; rs=0 rw=0 en=1
            MOVLW   D'5'      ; 10 ms delay
            CALL    DELAY
            CLRF    PORTB     ; rs=0 rw=0 en=0
            RETURN
LCDDATA     MOVWF   PORTD     ; Data sent to PORTD
            MOVLW   0x05      ; rs=1 rw=0 en=1
            MOVWF   PORTB
            MOVLW   D'5'      ; 10 ms delay
            CALL    DELAY
            MOVLW   0x01
            MOVWF   PORTB     ; rs=1 rw=0 en=0
            RETURN
DELAY       MOVWF   0x20
LOOP1       MOVLW   D'255'    ; LOOP2 provides 2 ms delay with a count of 255
            MOVWF   0x21
LOOP2       DECFSZ  0X21
            GOTO    LOOP2
            DECFSZ  0x20
            GOTO    LOOP1
            RETURN
            END
```

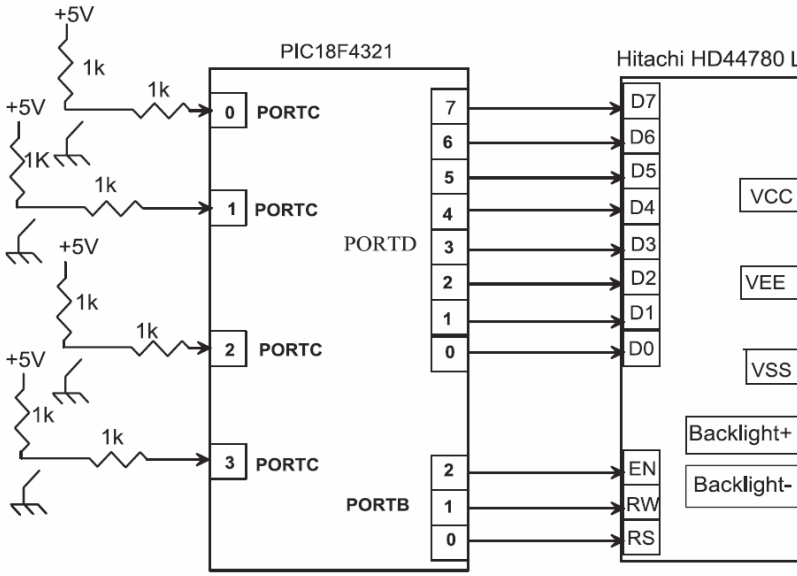


FIGURE 9.14 Figure for Example 9.4

PIC18F Interface to a typical LCD

In the above, the following loop for the 2 ms delay is used:

```
LOOP1   MOVLW   D'255'      ; LOOP2 provides 2 ms delay with a count of 255
        MOVWF    0x21
LOOP2   DECFSZ  0X21
        GOTO    LOOP2
```

PIC18F Interface to a typical LCD

- For a 1-MHz default crystal frequency, the PIC18F clock period will be 1 μsec. Hence, each instruction cycle will be 4 microseconds. For 2 ms delay, total cycles = (2 ms)/(4 μsec)= 500. The DECFSZ in the loop will require 2 cycles for (COUNT - 1) times when Z = 0 and the last iteration will take 1 cycle when skip is taken (Z = 1). Hence, the value of count will approximately be 255 (decimal) discarding execution times of certain instructions. Therefore, register 0x21 should be loaded with an integer value of 255 for an approximate delay of 2 ms.

PIC18F Interface to a typical LCD

- Let us consider the code for outputting a command code such as the command “move cursor to the beginning of the first line” to the LCD. From Table 8.3, the command code for this is 0x80. In the above program, the code MOVLW 0x80 moves 0x80 into WREG. The CALL CMD calls the subroutine called CMD. The CMD subroutine first outputs the command code 0x80 to PORTD using MOVWF PORTD.

PIC18F Interface to a typical LCD

- Since PORTD is connected to LCD's D0-D7 pins, this data will be available to be latched by the LCD. The following few lines of the code of the CMD subroutine are for outputting 0's to RS and R/W pins, and a trailing edge (1 to 0) pulse to EN pin along with a delay of 10 ms. Hence, the LCD will latch 0x80, and the cursor will move to the start of the first line. Note that an external counter of 5 is loaded into a register 0x21 with a 2 ms inner loop for LOOP2 is used for the 10 ms delay. Typical delays should be 10 to 30 milliseconds. Also, 1-MHz default crystal frequency for the PIC18F4321 is assumed. The program then returns to the main program.

PIC18F Interface to a typical LCD

- In order to display ‘S’, the MOVLW D’5’ moves 5 (decimal) into WREG, and CALL DELAY provides a 10 ms delay using this value in the routine. After executing the DELAY routine, MOVLW A’S’ moves the 8-bit ASCII code for S into WREG. The instruction CALL LCDDATA calls the subroutine called LCDDATA. The MOVWF PORTD instruction in this subroutine outputs the ASCII code for S into the D0-D7 pins of the LCD via PORTD. The next few instructions in the LCDDATA subroutine outputs 1 to the RS pin, 0 to the R/W pin, and a trailing edge (1 to 0) pulse to EN pin along with delay so that the LCD will latch ASCII code for S, and will display S on the screen.

(b) The C-program for the LCD is provided below:

```
#include <P18F4321.h>
#define DELAYTIME 5
void cmd(unsigned char);
void data(unsigned char);
void delay(unsigned int);
void main(void)
{
   unsigned char input, output,i;
    unsigned char tstr[13] = { 'S','w',
   'i','t','c','h',' ', 'V','a','l','u','e',':' };        // Switch input
   TRISD = 0x00;                                          // configure PORTD as output
   ADCON1 = 0x0F;                                         // PORTB as digital I/O
   TRISB = 0;                                             // configure PORTB as output
   TRISC = 0xFF;                                          // configure PORTC as input
   PORTB = 0x00;                                          // rs = 0 rw=0 en = 0
   delay(DELAYTIME);
   cmd(0x0C);                                             // display on, cursor off
   delay(DELAYTIME);
   cmd(0x01);                                             // clear display
```

PIC18F Interface to a typical LCD

```
delay(DELAYTIME);
cmd(0x06);                          // shift cursor to the right
delay(DELAYTIME);
cmd(0x80);                          // move the cursor to the start of the first line
delay(DELAYTIME);
for(i = 0; i < 13;i++)
    data(tstr[i]);                  // output "switch input:"
while(1)                            // Wait
{
   input = PORTC&0x0F;              // mask the four switch inputs
   output = 0x30|input;             // convert it to ASCII code by ORing with 30H
   data(output);                    // output the number
   delay(DELAYTIME);                // 10ms delay
   cmd(0x10);                       // shift cursor left one, goes back to the position
}
```

PIC18F Interface to a typical LCD

```
}                                                   // end main
void cmd(unsigned char value)
{
        PORTD = value;                              // command is sent to PORT D
        PORTB = 0x04;                               // rs= 0 rw=0 en=1
        delay(DELAYTIME);                           // delay 10ms
        PORTB = 0x00;                               // rs=0 rw = 0 en =0
}

void data(unsigned char value)
{
        PORTD = value;                              // data is sent to PORTD
        PORTB = 0x05;                               // rs=1 rw=0 en=1
        delay(DELAYTIME);                           // 10ms delay
        PORTB = 0x01;//rs=1 rw=0 en=0
}
```

PIC18F Interface to a typical LCD

```
void delay(unsigned int itime)
{
        unsigned int i,j;                    // 2 ms delay, Can be verified from the
        for(i=0; i<itime; i++);              // PIC18F assembly program for LCD
          for(j = 0; j<255;j++);
}
```

Basics of Keyboard and Display Interface

- A common method of entering programs into a microcontroller is via a keyboard. An inexpensive way of displaying microcontroller results is by using seven-segment displays. The main functions to be performed for interfacing a keyboard are:
  - Detect a key actuation
  - Debounce the key
  - Decode the key

Basics of Keyboard and Display Interface

- A keyboard is arranged in rows and columns. Figure 9.15 shows a 2 × 2 keyboard interfaced to a typical. In Figure 9.15, the columns are normally at a HIGH level. A key actuation is sensed by sending a LOW (closing the diode switch) to each row one at a time via RC0 and RC1 of port C. The two columns can then be inputted via RD2 and RD3 of PORTD to see whether any of the normally HIGH columns are pulled LOW by a key actuation. If so, the rows can be checked individually to determine the row in which the key is down. The row and column code for the key pressed can thus be found.

Basics of Keyboard and Display Interface

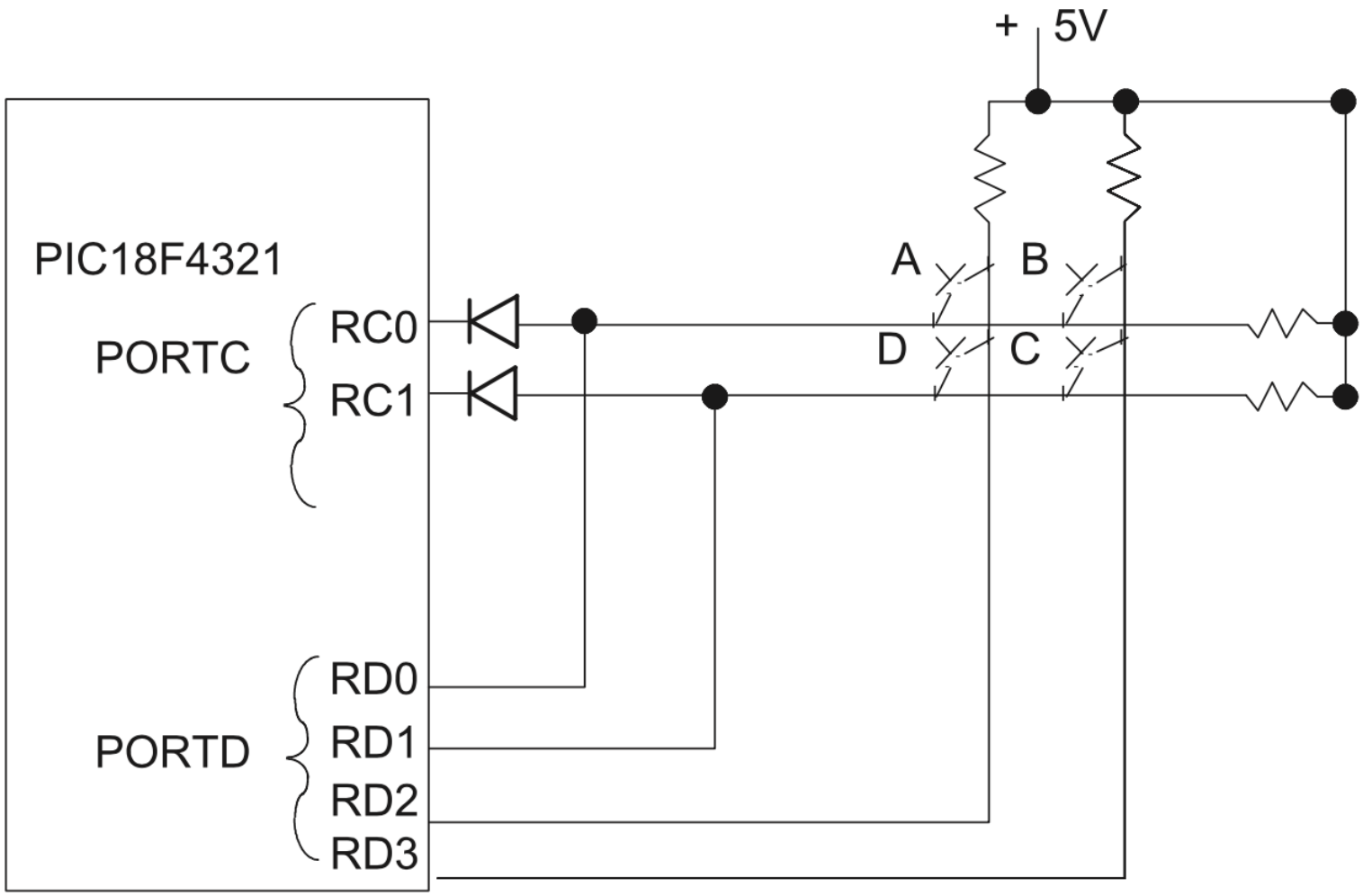


Figure 9.15 A 2x2 keyboard interfaced to the PIC18F4321

Basics of Keyboard and Display Interface

- Key bounce (抖動) occurs when a key is pressed or released; it bounces for a short time before making the contact. When the bounce occurs, **it may appear to the microcontroller that the same key has been actuated several times instead of just once.** This problem can be eliminated by reading the keyboard after about 20 ms and then verifying to see if it is still down. If it is, the key actuation is valid. The next step is to translate the row and column code into a more popular code.

Basics of Keyboard and Display Interface

- Translate the row and column code into a more popular code, such as hexadecimal or ASCII
- Certain characteristics associated with keyboard actuation must be considered
  - The **two-key lockout** ensures that only one key is pressed. An additional key depressed and released does not generate any codes. The system is simple to implement and most often used. However, it might slow down the typing because each key must be released fully before the next one is pressed down.
  - The **N-key rollover** will ignore all keys pressed until only one remains down.

Basics of Keyboard and Display Interface

- The following functions are typically performed for displays:
  - Output the appropriate display code
  - Output the code via right entry or left entry into the displays if there is more than one display
- A microcontroller program can easily realize these functions.

Basics of Keyboard and Display Interface

- If there is more than one display, the displays are typically arranged in rows, as shown in Figure 9.16.
  - One has the option of outputting the display code via right entry or left entry.
  - If the code is entered via **right entry**, **the code for the least significant digit of the four-digit display should be outputted first, then the next-digit code, and so on.** The program that outputs to the displays are so fast that visually all four digits will appear on the display simultaneously.
  - If the displays are entered via **left entry**, the most significant digit must be outputted first and the rest of the sequence is similar to that of the right entry

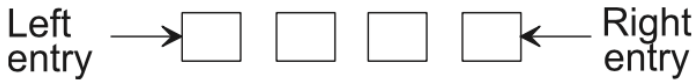


FIGURE 9.16 A row of four displays.

Basics of Keyboard and Display Interface

- Two techniques are typically used to interface a hexadecimal display to the microcontroller: **nonmultiplexed** and **multiplexed**.
- In nonmultiplexed methods, each hexadecimal display digit is interfaced to the microcontroller via an I/O port, as shown in Figure 9.17.
  - BCD-to-seven-segment conversion is done in software. The microcontroller can be programmed to output to the two display digits in sequence.
  - However, the microcontroller executes the display instruction sequence so fast that the displays appear to the human eye at the same time.

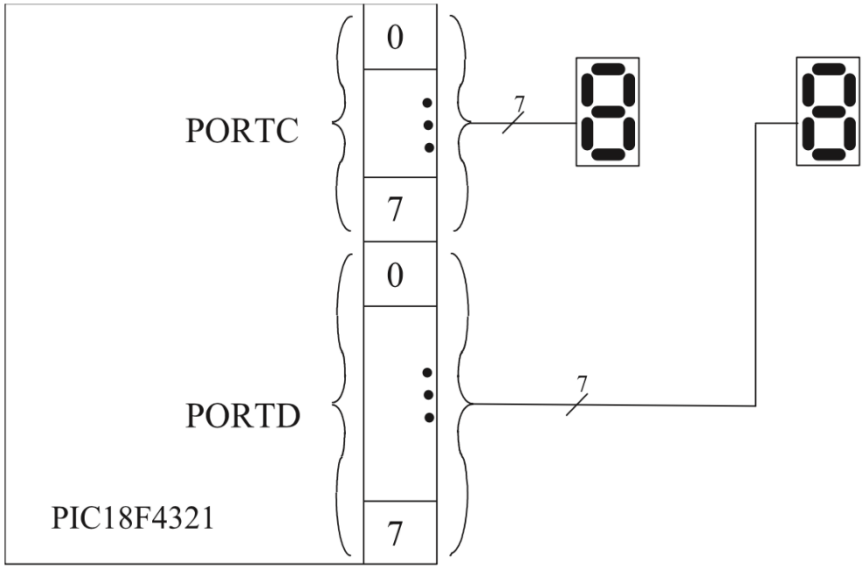


FIGURE 9.17 Nonmultiplexed hexadecimal displays.

Basics of Keyboard and Display Interface

- Figure 9.18 illustrates the multiplexing method of interfacing the two hexadecimal displays to the microcontroller.
  - In the multiplexing scheme, appropriate seven-segment code is sent to the desired displays on seven lines common to all displays. However, the display to be illuminated is grounded. Some displays, such as the Texas Instrument's TIL 311, have an on-chip decoder. In this case, the microcontroller is required to output 4 bits (decimal) to a display.

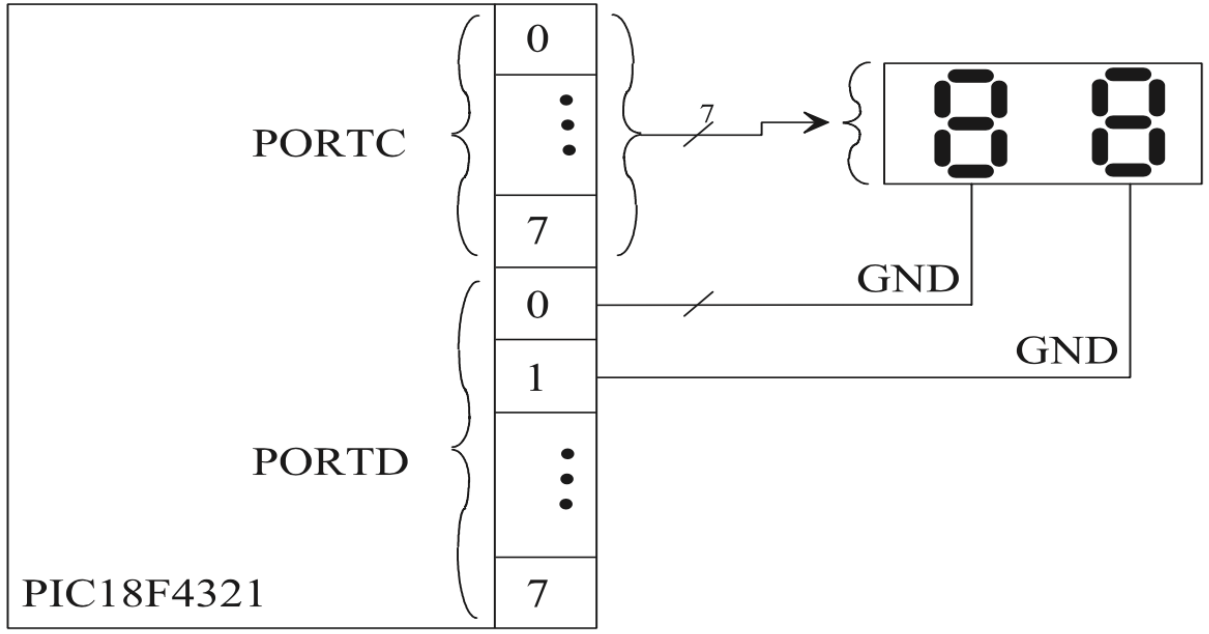


FIGURE 9.18 Multiplexed hexadecimal displays.

Basics of Keyboard and Display Interface

- The PIC18F4321 microcontroller is designed to display a hexadecimal digit (0-F) entered via a hexadecimal keypad (16 keys). The user will push one of the hex digits (0 to F) using the keys on the hexadecimal keyboard. The PIC18F4321 will input this data via PORTD, and output via PORTB to a seven-segment display.
- Figure 9.19 shows the hardware schematic.

Basics of Keyboard and Display Interface

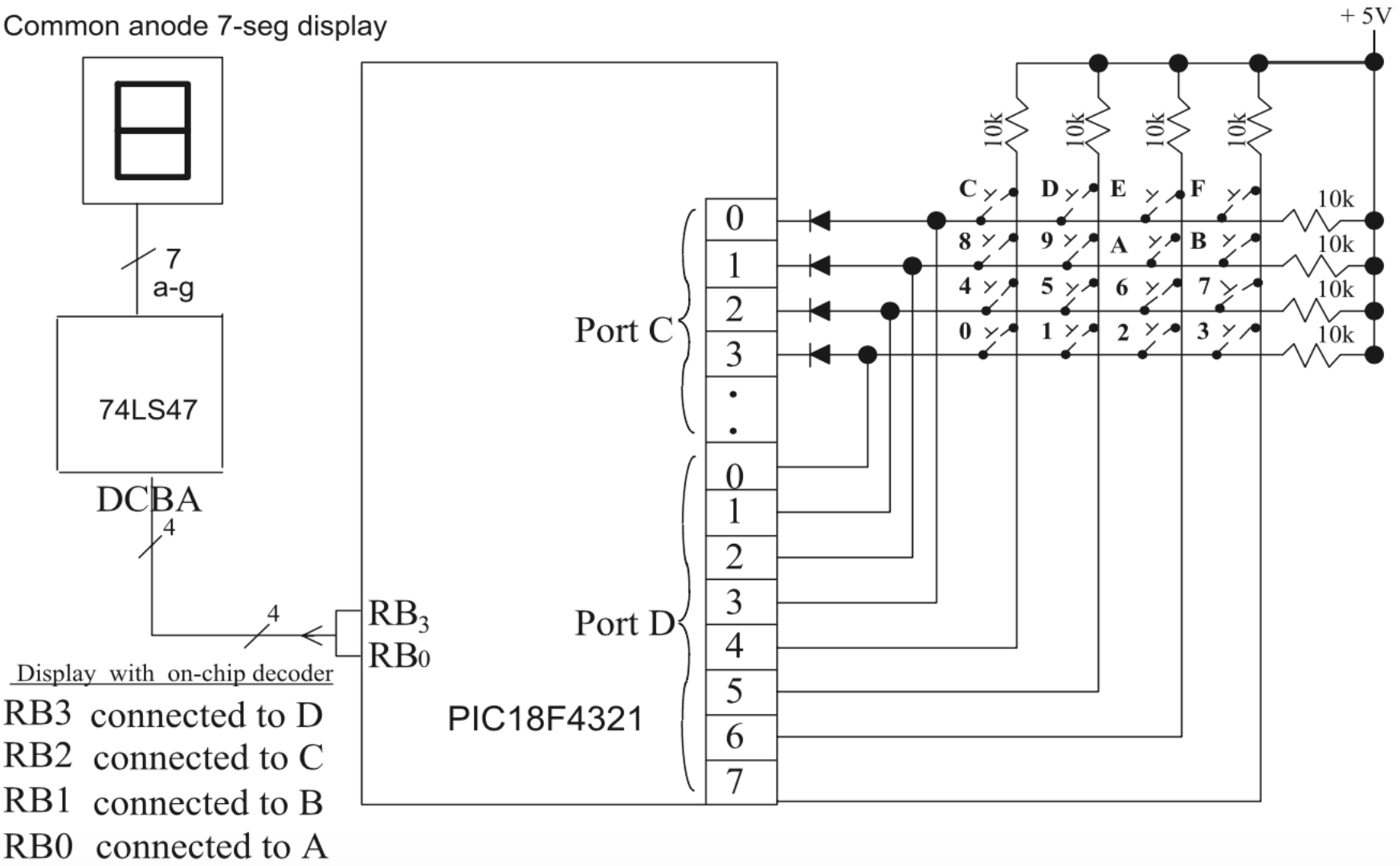


FIGURE 9. 19 PIC18F4321 interface to keyboard and display for the assembly program

Basics of Keyboard and Display Interface

- Three 8-bit I/O ports (PORTB, PORTC, PORTD) are used in the design. Ports B, C, and D are configured as follows:
  - PORTB is configured as an output port to display the key(s) pressed.
  - PORTC is configured as an output port to output zeros to the rows to detect a key actuation
  - PORTD is configured as an input port to receive the row–column code.

Basics of Keyboard and Display Interface

- The PIC18F4321 default crystal frequency of 1 MHz is assumed. Debouncing is provided to avoid unwanted oscillation caused by the opening and closing of the key contacts. To ensure stability of the input signal, a delay of 20 ms is used for debouncing the input.
- Using a 2 ms delay routine from the previous section as the inner loop, the following subroutine can be used for a delay routine of 20 ms:

Basics of Keyboard and Display Interface

```
DELAY   MOVLW   D'10'
        MOVWF   0x20
LOOP1   MOVLW   D'255'      ; LOOP2 provides a 2 ms delay with a count of 255
        MOVWF   0x21
LOOP2   DECFSZ  0X21
        GOTO    LOOP2
        DECFSZ  0x20
        GOTO    LOOP1
        RETURN
```

Assembly for keyboard/display interface

- A PIC18F assembly program is written for the keyboard/display interface. The program scans all 16 keys for key actuation. As soon as a key actuation is detected, the program will debounce the key using the DELAY routine, and then determine the key pressed using a decode table stored in memory. The decode table contains common anode seven-segment codes for the hexadecimal digits from 0 to F.

Assembly for keyboard/display interface

- The 74LS47 common-anode decoder is used for the display. It has four inputs (D, C, B, A with D as the most significant bit and A as the least significant bit), and seven outputs (a-g). The 7447 will display BCD numbers 0-9 and symbols for hex digits A-F as shown in Figure 9.20.

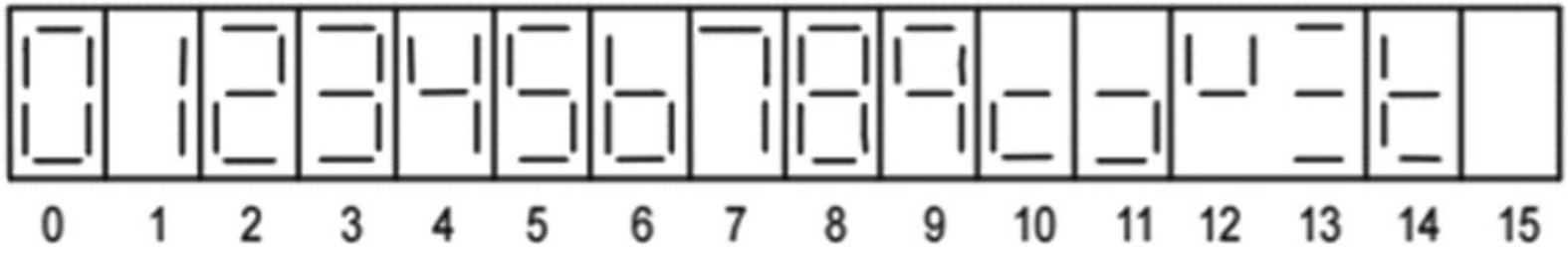


FIGURE 9. 20 74LS47 display for hex digits 0-F

Assembly for keyboard/display interface

- Seven-segment codes for 0-F are stored into data memory of the PIC18F4321 using the MOVLW instruction. Note that assembler directive DB could have been used to store the seven-segment codes in the program memory, and then transfer them to data memory using the TBLRD*+ instruction.

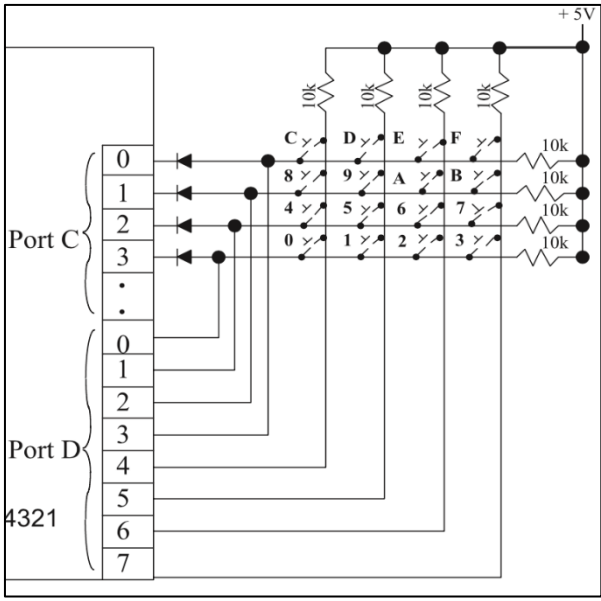
Assembly for keyboard/display interface

```
                INCLUDE<P18F4321.INC>
                ORG     0x100;          #1 Start of the program
OPEN            EQU     0xF0;           #2 Assign 0xF0 to OPEN
                MOVLW   0x77;           #3 Start of the Code
                MOVWF   0x50;           #4 Code for F
                MOVLW   0xB7;           Code for E
                MOVWF   0x51
                MOVLW   0xD7;           Code for D
                MOVWF   0x52
                MOVLW   0xE7;           Code for C
                MOVWF   0x53
                MOVLW   0x7B;           Code for B
                MOVWF   0x54
                MOVLW   0xBB;           Code for A
                MOVWF   0x55
                MOVLW   0xDB;           Code for 9
                MOVWF   0x56
                MOVLW    0xEB;          Code for 8
                MOVWF   0x57
                MOVLW   0x7D;           Code for 7
                MOVWF   0x58
                MOVLW   0xBD;           Code for 6
                MOVWF   0x59
                MOVLW   0xDD;           Code for 5
```

Assembly for keyboard/display interface

```
                MOVWF   0x5A
                MOVLW   0xED;           Code for 4
                MOVWF   0x5B
                MOVLW   0x7E;           Code for 3
                MOVWF   0x5C
                MOVLW   0xBE;           Code for 2
                MOVWF   0x5D
                MOVLW   0xDE;           Code for 1
                MOVWF   0x5E
                MOVLW   0xEE;           # 5 End of Code. Code for 0
                MOVWF   0x5F;
; Perform initializations
                CLRF    TRISB;          # 6 Configure PORTB as an output port
                CLRF    TRISC;          # 7 Configure PORTC as an output port
                SETF    TRISD;          # 8 Configure PORTD as an input port
                MOVLW   0x5;            # STKPTR is initialized with arbitrary value
                MOVWF   STKPTR;         since subroutine DELAY is used later
                MOVLW   0;               # 9 Initialize display with 0
                MOVWF   PORTB
```

```
; Detect a key actuation, debounce it, decode, and display
SCAN_KEY   MOVLW    0;          # 10 Output 0s to rows of the keyboard
           MOVWF    PORTC
           MOVLW    OPEN;       # 11 Move 0xF0 to 0x30
           MOVWF    0x30
KEY_OPEN   MOVF     PORD,W;     # 12 Read PORTD into WREG
           SUBWF    0x30,W;     # 13 Are keys opened?
           BNZ      KEY_OPEN;   # 14 Repeat if closed
           CALL     DELAY;      # 15 Debounce for 20 ms
KEY_CLOSE  MOVF     PORTD, W;   # 16 Read PORTD into WREG
           SUBWF    0x30,W;     # 17 Are all keys closed?
           BZ       KEY_CLOSE;  # 18 Repeat if opened
           CALL     DELAY;      # 19 Debounce again for 20 ms
           SETF     0x35;       # 20 Set 0x35 contents to all 1's
           BCF      STATUS,C;   # 21 Clear Carry flag
NEXT_ROW   RLCF     0x35,F;     # 22 Set up row mask
           MOVFF    0x35,0x36;  # 23 Save row mask in 0x36
           MOVFF    0x35,PORTC; # 24 Output 0 to a row
           MOVF     PORTD, W;   # 25 Read PORTD into WREG
           MOVWF    0x31;       # 26 Save row/column codes in 0x31
           MOVLW    0xF0;       Move data for masking
           ANDWF    0x31,W;     # 27 Mask row code
           CPFSEQ   0x30;       # 28 is column code affected?
           BRA      DECODE;     # 29 if affected, row found 0x31 has row
                                ; and column code
           MOVFF    0x36,0x35;  # 30 Restore row mask in 0x35
           BCF      STATUS,C;   # 31 Clear carry flag to 0
           GOTO     NEXT_ROW;   # 32 Check next row
DECODE     MOVLW    D'16';      # 33 Initialize 0x32 with 16 decimal since there
```



Assembly for keyboard/display interface

```
            MOVWF   0x32;           # 34 are 16 hex digits
            MOVWF   0x33;           Move 16 to 0x33
            DECF    0x33,F;         Decrement 0x33 by 1 to contain hex digits F to 0
            LFSR    0, 0x50;        # 35 Initialize FSR0 with 0x50
            MOVF    0x31,W;         # 36 Move row code to WREG
SEARCH      CPFSEQ  POSTINC0;       # 37 Compare and skip if equal
            BRA     SEARCH1;        # 38 Load if not found
            MOVFF   0x33, PORTB;    # 39 Get character along with LOW enable
            BRA     NEXT1;          # 40 Branch to NEXT1
SEARCH1     MOVFF   0x31, W
            DECF    0x33,F;         Decrement 0x33
            DECF    0x32,F;         Decrement 0x32
NEXT1       GOTO    SCAN_KEY;       #41 Branch to SEARCH if not 0
DELAY       MOVLW   D'10';          Return to scan another key
            MOVWF   0x20;           20 ms delay routine
LOOP1       MOVLW   D'255';         LOOP2 provides 2 ms delay
            MOVWF   0x21
LOOP2       DECFSZ  0x21;
            GOTO    LOOP2;
            DECFSZ  0x20;
            GOTO    LOOP1;
            RETURN;
            END
```

Assembly for keyboard/display interface

- In the program, a decode table for keys 0 through F are loaded into data memory using the MOVLW instruction starting at address 0x50 (chosen arbitrarily). The codes for the hexadecimal numbers 0 through F are obtained by inspecting Figure 9.19. For example, consider key 9. When key 9 is pressed and if a LOW is outputted by the program to bit 0 of PORTC, the second row and second column of the keyboard will be LOW. This will make the content of PORTD:

Bit Number:	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Data:	1	1	0	1	1	0	1	1	$= DB_{16}$

Assembly for keyboard/display interface

- Thus, a code of DB16 is obtained at PORTD when key 9 is pressed. Diodes are connected at the 4 bits (bits 0-3) of PORTC. This is done to make sure that when a 0 is output by the program to one of these bits (row of the keyboard), the diode switch will close and will generate a LOW on that row.

Assembly for keyboard/display interface

- Now, if a key is pressed on a particular row that is LOW, the column connected to this key will also be LOW. This will enable the programmer to obtain the appropriate key code for each key. Next, the assembly language program will be explained using some of the line numbers included in the comment field.

Assembly for keyboard/display interface

- Line #1 is the starting address of the program at 0x100. This address is chosen arbitrarily.
- Line #2 equates label OPEN to data 0xF0. This is because when all keys are up (no keys are pushed) and 0's are outputted to the rows via PORTC in Figure 9.19, data input at PORTD will be 11110000 (0xF0). Note that bits 0 -3 of PORTD are connected to rows and bits 4-7 of PORTD are connected to columns of the keyboard.

Assembly for keyboard/display interface

- Line #3 and #4 store code for F, and are the start of the codes while Line #5 is the end of codes.
- Line #’s 6 through 8 configure ports and initializes STKPTR . Line #9 initialize the seven-segment display by outputting 0.

Assembly for keyboard/display interface

- Line #’s 10 through 14 check to see if any key is pushed. This is done by outputting 0’s to all rows via PORTC, and then inputting PORTD. If all keys are open, data at PORTD will be 0xF0. Hence, 0xF0 stored in data memory address 0x30 is subtracted from data at PORTD in WREG. If Z = 0, the program waits in a loop with label KEY_OPEN until a key is pushed. When a key is closed, Z = 1, and the program comes out of the loop.
- Line #15 CALLs the DELAY routine to debounce the key by providing a delay of 20 ms.

Assembly for keyboard/display interface

- Line #’s 16 through 18 detect a key closure. The program inputs PORTD into WREG and, subtracts 0xF0 stored in 0x30 from [WREG]. If Z = 1, the program waits in a loop with label KEY_CLOSE until a key is closed. If Z = 0, the program leaves the loop.
- Line #19 CALLs the DELAY routine to debounce as soon as a key closure is detected. It is necessary to determine exactly which key is pressed. This is accomplished by outputting a ‘0’ to a row while outputting 1’s to the other three rows. Hence, a sequence of row-control codes (0xFE, 0xFD, 0xFB, and 0xF7, where the upper 4 bits ‘F’ are don’t cares in this case) are outputted via PORTC.

Assembly for keyboard/display interface

- Line #’s 20 through 22 initialize 0x35 to all 1’s, clear the C-bit to 0, and rotate [0x35] through carry once to the left to contain the appropriate row control code.
- For example, after the first RLCF in line #22, 0x35 will contain 11111110 (0xFE). Note that the low 4 bits are the row-control code (the upper 4 bits are don’t cares) for the first pass in the loop, labeled NEXT_ROW.
- Line #’s 23 and 24 output this data to PORTC to make the top row of the keyboard zero. The row–column code is inputted via PORTD to determine if the column code changes corresponding to each different row code.
- Line #’s 25 and 26 input PORTD into 0x31 via WREG.

Assembly for keyboard/display interface

- Lines 27 through 32 make the low 4 bits 0's and retain the upper 4 bits. If the column code is not 0xF0 (changed), the input key is identified. The program then goes through a lookup table to match the row–column code saved in 0x33. If the code is found, the corresponding index value, which equals the input key's value (0-9, and symbols for A-F), is displayed. However, if no key in the top row is pushed, a 0 is outputted to the second row, and the process continues. The program is written such that it will scan continuously for an input key and update the display for each new input.

Assembly for keyboard/display interface

- Suppose that key 9 is pushed when the program branches to DECODE at line #33.
- Line #’s 33 through 35 initialize data memory addresses 0x32, and 0x33 with 16 (total number of hex digits), decrement [0x33] by 1 which will initially hold 9, and will eventually contain the BCD digit to be displayed. Line #35 will load the starting address 0x50 of the decode table into FSR0 to be used as an indirect pointer.

Assembly for keyboard/display interface

- Line #36 moves the row code 0xDB (for 9) saved in data memory address 0x31 (Line #26) into WREG. The instruction “CMPSEQ POSTINC0” at line 37 will compare the code for 9 at address 0x50 (starting address of the table) with [WREG]. Since it is assumed that ‘9’ is pushed, there will be a match, and the instruction at line #40 will be skipped.
- The instruction “MOVFF 0x33, PORTB” at line #39 will be executed which will output 1001 to the DCBA inputs of the 7447 via PORTB, and the digit 9 will be displayed on the seven-segment display. The instruction “BRA NEXT1” at line #40 will branch to label NEXT1 (Line #41) which, in turn, will go back, and repeat the process.

Assembly for keyboard/display interface

- C language program for the keyboard/display interface Figure 9.21 shows the hex keyboard interface to the PIC18F4321. Since the 74LS47 is used for the display, digits 0-9 will be displayed properly while A-F will be displayed according to Figure 9.20. For example, if the key ‘F’ is pushed, then all LED segments will be turned OFF.

Assembly for keyboard/display interface

```
#include <P18F4321.h>
#define TRUE 1
#define FALSE 0
#define setbit(var,bit)((var) |= (1<<(bit)))          // #1 set a bit var = var | (1 <<bit)
#define clearbit (var,bit)((var) &= ~(1<<(bit)))   // #2 clear a bit var = var & ~(1<<bit)
#define testbit(var,bit)((var) & (1<<(bit)))        // #3 test a bit return var & (1<<bit)
unsigned char row=0x0F;                                  // #4 all rows are initially high
   int i=0, j=0;
unsigned char Matrix[4][4] =                          // #5 create a matrix to store the
                                                                 // corresponding Hex key numbers
{
{0x0F, 0x0E, 0x0D, 0x0C},
{0x0B, 0x0A, 0x09, 0x08},
{0x07, 0x06, 0x05, 0x04},
{0x03, 0x02, 0x01, 0x00}};
void delay_ms(unsigned int time);                  // #6 delay function to prived 2ms delay
```

```
void main()
{
OSCCON = 0x60;                          // 4MHz clock
ADCON1 = 0x0F;                          // #8 AN0-AN12 all digital input
TRISB = 0x00;                  // #9 configure PORTB as output for 7 seg decoder
TRISC = 0x00;                  // #10 configure PORTC as output for matrix keyboard
TRISD = 0xFF;                  // #11 configure PORTD as input for key pressed
PORTB = 0x00;                  // #12 output initially set to 0 at PORTB
while (1) {
PORTC = 0x00;                  // #13 output initially set to 0 at PORTC
while(PORTD! = 0x0F);          // #14 wait in while loop if a key is pressed
while(PORTD = = 0x0F);         // #15 wait in while loop if no keys are pressed
for(i = 0; i<4; i++)           // #16 Scan for the keyboard FOR COLUMN
{
for(j = 0; j<4;j++)            // #17 Scan FOR ROWS
 {
clearbit(row,j);               // #18 Clear a bit for corresponding row j
PORTC = row;                   // #19 Output a low for that row
if(testbit(PORTD,i)= = 0)      // #20 Detect if a key is pressed in column i
  {
  delay_ms(20);                // #21 Delay for 20ms
   if(testbit(PORTD,i) ==  0)  // #22 Check again for debouncing
   {
    PORTB = Matrix[j][i];      // #23 output result from lookup table to PORTB
   }
   }
   setbit(row,j);              // #24 Set the zero bits back to row variable
 }
```

Assembly for keyboard/display interface

```
 }
}
}
void delay_ms(unsigned int time)    // #25 2ms delay function
 {
 unsigned int i, j;
 for(i=0; i<time; i++)
 for(j=0; j<255; j++);
}
```

Assembly for keyboard/display interface

- In Figure 9.21, the four rows and four columns of the hex keyboard are connected to the PIC18F4321 via PORTC and PORTD respectively. The top row (row 0) of the keyboard is connected to RC0, next row (row 1) to RC1, next row (row 2) to RC2, and next row (row 3) to RC3. On the other hand, the rightmost column (column 0) is connected to RD3, next column to the left (column 1) to RD2, next column to the left (column 2) to RD1, and next column to the left (column 3) to RD0.

Assembly for keyboard/display interface

- In the figure, the columns of the keyboard connected to +5V, are normally HIGH. The key matrix at line #5 in the program is arranged differently than the Keyboard in Figure 9.21. For example, in the program Key ‘F’ is located at row 0 and column 0. Hence, key ‘F’ is located on the extreme right of the top row of the key matrix.

Assembly for keyboard/display interface

- In order to detect a key actuation, 0's are outputted on the four rows using "PORTC = 0x00;" (Line # 13). If a key is pushed, that key will place a '0' on the normally HIGH column connected to it. Thus, the pushed key will make one of the columns connected to it '0', and the other three columns will be HIGH.

Assembly for keyboard/display interface

- Line #18 clears a bit for row j, and line #19 outputs a LOW on that row.
- Line #20 detects if a key is pushed in column i connected to that row. Once the key is found, it is debounced (Line #21). With i and j, the key is obtained from the key matrix [j][i], and then outputted to PORTB on the seven segment display (Line #23).

Assembly for keyboard/display interface

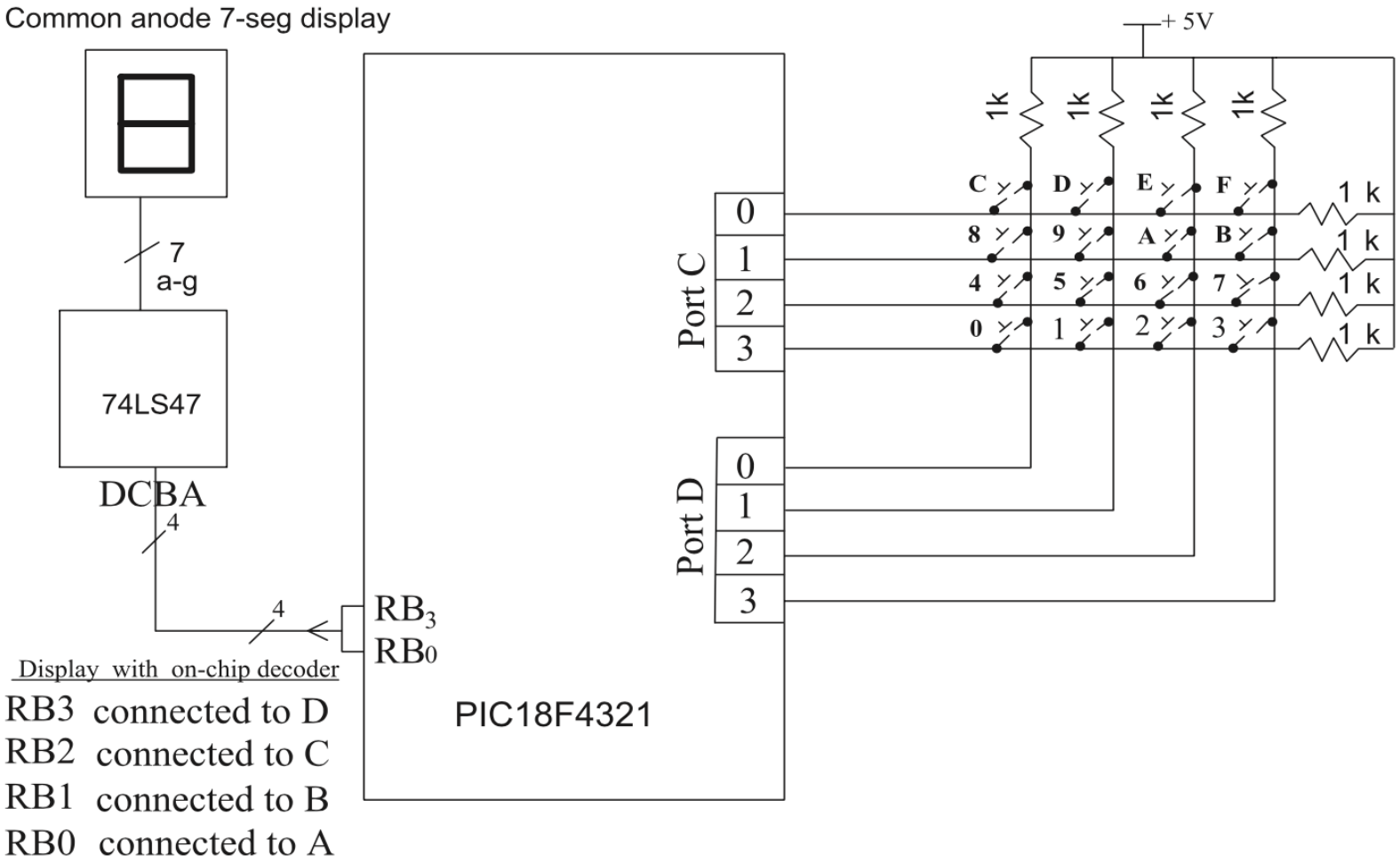


FIGURE 9.21 PIC18F4321 interface to keyboard and display for the C-program

Assembly for keyboard/display interface

- As an example, consider pressing key number 2. Note that ‘2’ is located at row 3 (j = 3) and at column 1 (i = 1). In the program, j = 0 and i = 0 initially. The “clearbit” function (Line #18) clears a bit, and then “PORTC = row;” (Line #19) outputs a ‘0’ on the row. This means that for j = 0 and i = 0, PORTC outputs 0xE (1110). The “testbit” function (Line #20) checks whether the input at PORTD is 0. If it is 0, the key is found. The key is debounced (Line #21) and then outputted to PORTB. The pushed key will thus be displayed on the seven-segment display. In case of pressed key ‘2’, the key is not found yet. Hence, the process continues until j = 3 and i =1.

Assembly for keyboard/display interface

- When j = 3 and i = 1, the “clearbit” function at line #18 clears a bit for row 3 (j = 3) and outputs 0x7 (0111) on low four bits of PORTC. When the “if” statement with the “testbit” function is executed (Line #20), column 1 (i = 1) is obtained. The coordinate with j = 3 (row 3) and i = 1 (column 1) provides the pressed key ‘2’. The key is then debounced and outputted to PORTB. Hence, the seven segment display will display ‘2’.