CHAPTER 6

Intermediate PIC18 Projects

Chapter Outline

PIC Microcontroller Projects in C. http://dx.doi.org/10.1016/B978-0-08-099924-1.00006-X

In this chapter, we will be developing more complex projects using various peripheral devices. As in the previous chapter, the project description, hardware design, PDL, full program listing, and description of the program for each project will be given in detail.

Project 6.1—Four-Digit Multiplexed Seven-Segment Light Emitting Diode Event Counter Using an External Interrupt

Project Description

This project is similar to Project 5.10, but here, the timer interrupt of the microcontroller is used to refresh the displays. In Project 5.10, the microcontroller was busy updating the displays continuously, and thus, it could not perform any other tasks. For example, if we wish to make a counter with a 1 s delay between each count, the program given in Project 5.10 could not be used as the displays cannot be updated while the program waits for 1 s.

In this project, an external interrupt-based event counter will be designed to count up by one and display on the seven-segment displays every time an external event is detected. In this project, external interrupt input INT0 (RB0) is used as the event input. An event is said to

occur whenever the INT0 input goes from logic 0 to logic 1. The event count will be incremented inside an interrupt service routine (ISR). The displays will be refreshed inside a timer ISR so that the processor is free to do other tasks while the displays are refreshed.

In this project, Timer0 is used in the 8-bit mode (since the required delay is only several milliseconds) to refresh the displays. The time for a timer interrupt is given by the following:

$$\text{Time} = (4 \times \text{clock period}) \times \text{Prescaler} \times (256 - \text{TMR0L})$$

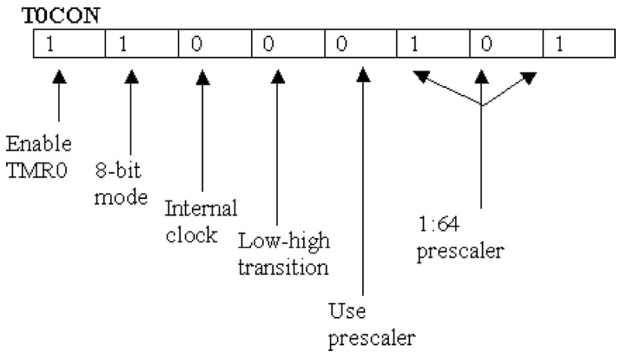
Where Prescaler is the selected prescaler value, and TMR0L is the value loaded into timer register TMR0L to generate timer interrupts every Time period. In our application, the clock frequency is 8 MHz, that is, the clock period = 0.125 μs, and Time = 5 ms. Selecting a prescaler value of 64, the number to be loaded into TMR0L can be calculated as follows:

$$\text{TMR0L} = 256 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

or

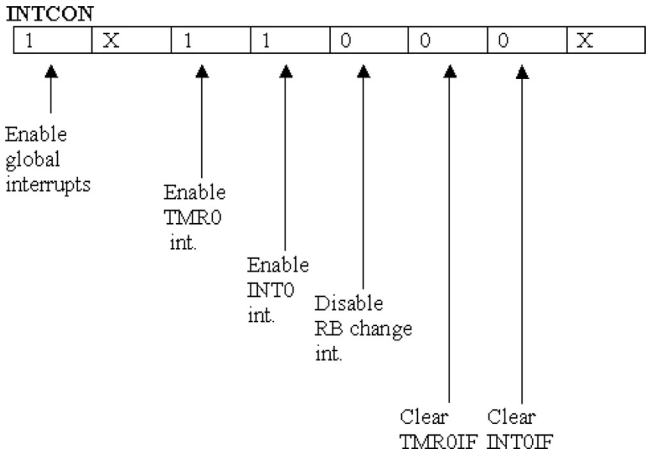
$$\text{TMR0L} = 256 - \frac{5000}{4 * 0.125 * 64} = 100$$

Thus, TMR0L should be loaded with 100. The value to be loaded into the TMR0 control register T0CON can then be found as follows:



Thus, the T0CON register should be loaded with hexadecimal 0xC5. The next register to be configured is the interrupt control register INTCON where we will disable

priority-based interrupts and enable the global interrupts and TMR0 and INT0 interrupts:



Taking the do not care entries (X) as 0, the hexadecimal value to be loaded into register INTCON is thus 0xB0.

In addition, we have to set bit 6 of register INTCON2 so that external interrupts are recognized on the low to high transition of the INT0 pin. Looking at the data sheet, we see that this bit is automatically set by default after power up (or reset).

In this project, the source of interrupt can either be from the timer or from an external event. Inside the ISR, we should check the interrupt flags of each source to determine the actual cause of the interrupt.

When a timer interrupt occurs, the TMR0L register should be reloaded inside the timer ISR. Interrupt flags of both interrupt sources must be cleared inside their ISR routines so that further interrupts can be accepted from these sources.

Figure 6.1 shows the block diagram of the project.

Project Hardware

The circuit diagram of the project is shown in Figure 6.2. The circuit is basically the same as in Figure 5.56, but here, additionally the external interrupt input is used for event triggering.

If you are using the EasyPIC V7 development board, make sure that the following jumpers are set correctly on the board:

SW4: DIS0, DIS1, DIS2, DIS3 set to ON
J17: set to GND

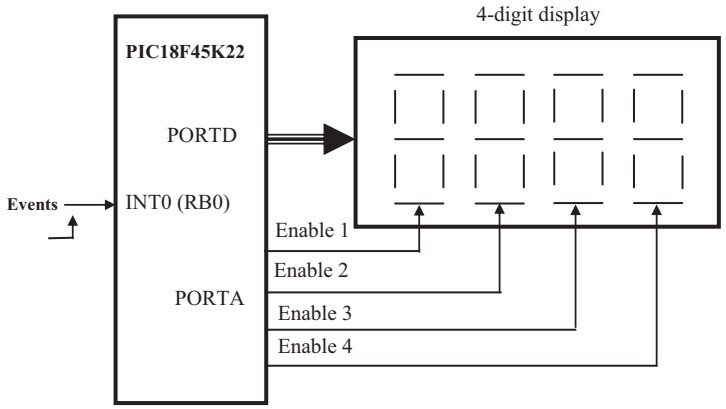


Figure 6.1: Block Diagram of the Project.

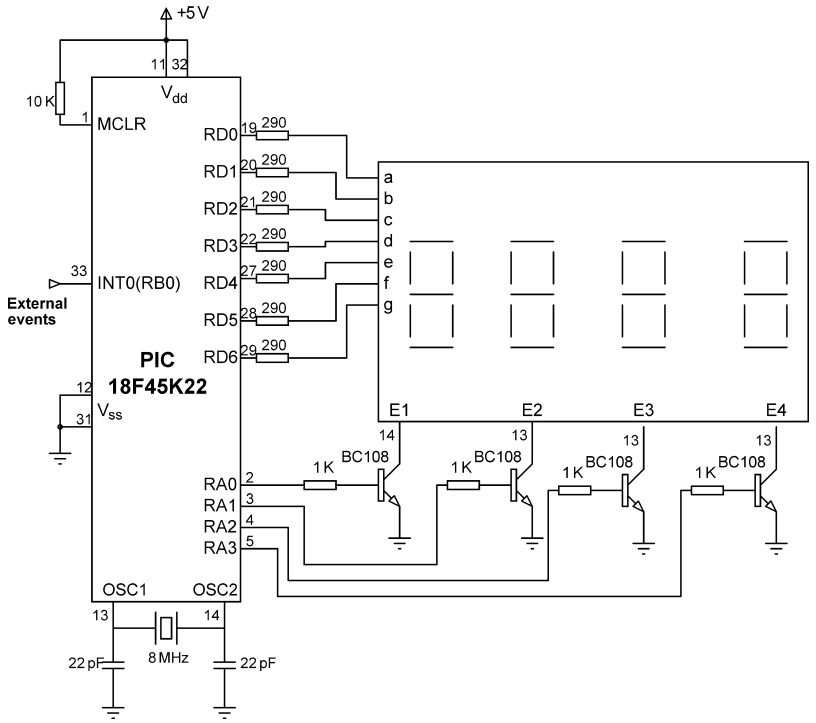


Figure 6.2: Circuit Diagram of the Project.

```
MAIN PROGRAM:

BEGIN
        Configure PORTD as digital and output
        Configure PORTA as digital and output
        Configure PORTB as digital and RB0 input
        Clear all digits
        Configure TIMER0
        Configure and enable external interrupts
        WAIT FOREVER
END


INTERRUPT SERVICE ROUTINE:

        BEGIN
                IF the interrupt source = Timer0
                        Reload timer register
                        Clear timer interrupt flag
                        CALL Display to display digit data
                        Enable appropriate display digit
                ELSE IF the interrupt source = INT0
                        Increment event count
                        Clear INT0 interrupt flag
                ENDIF
        END


BEGIN/DISPLAY
        Extract the bit pattern to send to the port to display a number
        Return the bit number to the calling programm
END/DISPLAY
```

Figure 6.3: PDL of the Project.

Project PDL

The PDL of the project is shown in Figure 6.3. The program is in two sections: the main program and the ISR. Inside the main program, TMR0L is configured to generate interrupts at every 5 ms. Also, input INT0 (RB0) is configured to accept external interrupts. Whenever an interrupt occurs, the source of the interrupt is detected. If the source is timer interrupt, then the display is refreshed. If the source of the interrupt is INT0, then it is assumed that an external event has occurred, and a counter is incremented by 1.

Project Program

mikroC Pro for PIC

The mikroC Pro for PIC program is called MIKROC-EVENT1.C and is shown in Figure 6.4. At the beginning of the main program, PORTD and PORTA are configured as

```
/*************************************************************************
                    4-DIGIT 7-SEGMENT DISPLAY EVENT COUNTER
                    ======================================

In this project four common cathode 7-segment LED displays are connected to PORTD of a
PIC18F45K22 microcontroller and the microcontroller is operated from an 8 MHz crystal.
Four PORTA pins are used to enable/disable the LEDs. In addition, external interrupt input
INT0 (RB0) is used to receive external events. An event is assumed to occur if pin INT0 goes
from logic 0 to logic 1.

The program uses two ISR routines: the timer routine is used to refresh the Display every 5 ms.
External interrupt ISR is used to increment the event count. The event count is displayed
continuously on the 7-segment displays.


Author:      Dogan Ibrahim
Date:        August 2013
File:        MIKROC-EVENT1.C
*************************************************************************/
#define DIGIT1 PORTA.RA0
#define DIGIT2 PORTA.RA1
#define DIGIT3 PORTA.RA2
#define DIGIT4 PORTA.RA3

unsigned int Cnt = 0;
unsigned char flag = 0;
unsigned int D1,D2,D3,D4,D5,D6;

//
// This function finds the bit pattern to be sent to the port to display a number
// on the 7-segment LED. The number is passed in the argument list of the function.
//
unsigned char Display(unsigned char no)
{
  unsigned char Pattern;
  unsigned char SEGMENT[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

  Pattern = SEGMENT[no];                          // Pattern to return
  return (Pattern);
}

//
// Interrupt Service Routine
//
void interrupt (void)
{
   if(INTCON.TMR0IF == 1)                         // If Timer interrupt occured
   {
    TMR0L = 100;                                  // Reload timer register
    INTCON.TMR0IF = 0;                            // Clear timer interrupt flag
    switch(flag)
```

Figure 6.4: mikroC Pro for PIC Program Listing.

```
    {
      case 0:
      {
        DIGIT1 = 0;                          // Disable digit 1
        D1 = Cnt/1000;                       // 1000s  digit
        PORTD = Display(D1);                 // Send to PORTD
        DIGIT4 = 1;                          // Enable digit 4
        flag = 1;
        break;
      }
      case 1:
      {
        DIGIT4 = 0;                          // Disable digit 4
        D2 = Cnt % 1000;
        D3 = D2/100;                         // 100s digit
        PORTD = Display(D3);                 // Send to PORTD
        DIGIT3 = 1;                          // Enable digit 3
        flag = 2;
        break;
      }
      case 2:
      {
        DIGIT3 = 0;                          // Disable digit 3
        D4 = D2 % 100;
        D5 = D4/10;                          // 10s digit
        PORTD = Display(D5);                 // Send to PORTD
        DIGIT2 = 1;                          // Enable digit 2
        flag = 3;
        break;
      }
      case 3:
      {
        DIGIT2 = 0;                          // Disable digit 2
        D6 = D4 % 10;
        PORTD = Display(D6);                 // Send to PORTD
        DIGIT1 = 1;                          // Enable digit 1
        flag = 0;
        break;
      }
    }
  }
  if(INTCON.INT0IF == 1)                     // If external interrupt occurred
  {
    Cnt++;                                   // Increment event count
    INTCON.INT0IF = 0;                       // Clear ext interrupt flag
  }
}

//
// Start of MAIN Program
//
void main()
{
```

Figure 6.4
cont'd

```
    ANSELA = 0;                                    // Configure PORTA as digital
    ANSELD = 0;                                    // Configure PORTD as digital
    ANSELB = 0;                                    // Configure PORTB as digital
    TRISA = 0;                                     // Configure PORTA as outputs
    TRISD = 0;                                     // Configure PORTD as outputs
    TRISB = 1;                                      // RB0 is event input

    DIGIT1 = 0;                                    // Disable digit 1
    DIGIT2 = 0;                                    // Disable digit 2
    DIGIT3 = 0;                                    // Disable digit 3
    DIGIT4 = 0;                                    // Disable digit 4
//
// Configure TIMER0 interrupts
//
    T0CON = 0xC5;                                  // TIMER0 in 8-bit mode
    TMR0L = 100;                                   // Load Timer register
//
// Configure External interrupts and enable interrupts
//
    INTCON = 0xB0;

    for(;;)                                        // Endless loop
    {                                              // Wait and process interrupts
    }
}
```

Figure 6.4
cont'd

digital outputs. Pin INT0 (RB0) is configured as the digital input since this is the event input.

Then, all the display digits are cleared, and register T0CON is loaded with 0xC5 to enable the TMR0 and set the prescaler to 64. The TMR0L register is loaded with 100 so that an interrupt can be generated every 5 ms. INTCON is then set to 0xB0 to enable global interrupts, timer interrupts, and external interrupts from input INT0. The remainder of the main program waits for interrupts to occur and does not do anything useful.

Inside the ISR, the program checks to determine the source of the interrupt. If the timer has caused the interrupt, then the timer register is reloaded, and the timer interrupt flag is cleared so that the processor can accept further interrupts from the timer. The display digits are then refreshed inside the timer ISR. Here, only one digit is enabled at any time.

```
/*************************************************************************
                    4-DIGIT 7-SEGMENT DISPLAY EVENT COUNTER
                    =======================================

In this project four common cathode 7-segment LED displays are connected to PORTD of a
PIC18F45K22 microcontroller and the microcontroller is operated from an 8 MHz crystal.
Four PORTA pins are used to enable/disable the LEDs. In addition, external interrupt input
INT0 (RB0) is used to receive external events. An event is assumed to occur if pin INT0
goes from logic 0 to logic 1.

The program uses two ISR routines: the timer routine is used to refresh the Display every 5 ms.
External interrupt ISR is used to increment the event count. The event count is displayed
continuously on the 7-segment displays.

In this version of the program leading zeroes are blanked.

Author:         Dogan Ibrahim
Date:           August 2013
File:           MIKROC-EVENT2.C
*************************************************************************/
#define DIGIT1 PORTA.RA0
#define DIGIT2 PORTA.RA1
#define DIGIT3 PORTA.RA2
#define DIGIT4 PORTA.RA3

unsigned int Cnt = 0;
unsigned char flag = 0;
unsigned int D1,D2,D3,D4,D5,D6;

//
// This function finds the bit pattern to be sent to the port to display a number
// on the 7-segment LED. The number is passed in the argument list of the function.
//
unsigned char Display(unsigned char no)
{
   unsigned char Pattern;
   unsigned char SEGMENT[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

   Pattern = SEGMENT[no];                          // Pattern to return
   return (Pattern);
}

//
// Interrupt Service Routine
//
void interrupt (void)
{
    if(INTCON.TMR0IF == 1)                         // If Timer interrupt occured
    {
      TMR0L = 100;                                 // Reload timer register
      INTCON.TMR0IF = 0;                           // Clear timer interrupt flag
```

Figure 6.5: mikroC Pro for the PIC-Modified Program.

```
switch(flag)
{
 case 0:
 {
  DIGIT1 = 0;                          // Disable digit 1
  D1 = Cnt/1000;                       // 1000s  digit
  if(D1 != 0)                          // Check if blanking required
  {
    PORTD = Display(D1);               // Send to PORTD
    DIGIT4 = 1;                        // Enable digit 4
  }
  flag = 1;
  break;
 }
 case 1:
 {
  DIGIT4 = 0;                          // Disable digit 4
  D2 = Cnt % 1000;
  D3 = D2/100;                         // 100s digit
  if(D3 != 0 || D1 != 0)               // Check if blanking required
  {
    PORTD = Display(D3);               // Send to PORTD
    DIGIT3 = 1;                        // Enable digit 3
  }
  flag = 2;
  break;
 }
 case 2:
 {
  DIGIT3 = 0;                          // Disable digit 3
  D4 = D2 % 100;
  D5 = D4/10;                          // 10s digit
  if(D5 != 0 || D3 != 0 || D1 != 0)    // Check if blanking is required
  {
    PORTD = Display(D5);               // Send to PORTD
    DIGIT2 = 1;                        // Enable digit 2
  }
  flag = 3;
  break;
 }
 case 3:
 {
  DIGIT2 = 0;                          // Disable digit 2
  D6 = D4 % 10;
  PORTD = Display(D6);                 // Send to PORTD
  DIGIT1 = 1;                          // Enable digit 1
  flag = 0;
  break;
 }
}
}
```

Figure 6.5
cont'd

```
    if(INTCON.INT0IF == 1)                          // If external interrupt occurred
    {
      Cnt++;                                        // Increment event count
      INTCON.INT0IF = 0;                            // Clear ext interrupt flag
    }
}



//
// Start of MAIN Program
//
void main()
{

    ANSELA = 0;                                     // Configure PORTA as digital
    ANSELD = 0;                                     // Configure PORTD as digital
    ANSELB = 0;                                     // Configure PORTB as digital
    TRISA = 0;                                      // Configure PORTA as outputs
    TRISD = 0;                                      // Configure PORTD as outputs
    TRISB = 1;                                      // RB0 is event input

    DIGIT1 = 0;                                     // Disable digit 1
    DIGIT2 = 0;                                     // Disable digit 2
    DIGIT3 = 0;                                     // Disable digit 3
    DIGIT4 = 0;                                     // Disable digit 4
 //
 // Configure TIMER0 interrupts
 //
    T0CON = 0xC5;                                   // TIMER0 in 8-bit mode
    TMR0L = 100;                                    // Load Timer register
 //
 // Configure External interrupts and enable interrupts
 //
    INTCON = 0xB0;

    for(;;)                                         // Endless loop
    {                                               // Wait and process interrupts
    }
 }
```

Figure 6.5
cont'd

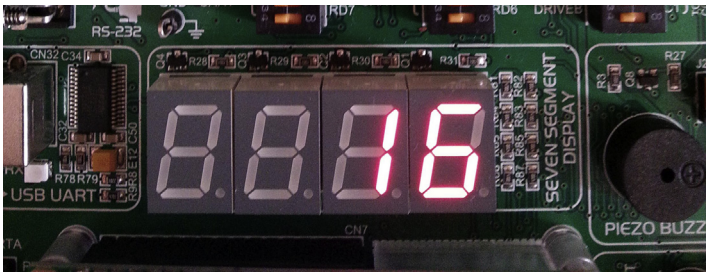


Figure 6.6: Example Display. (For color version of this figure, the reader is referred to the online version of this book.)

For each successive interrupt, data are sent to the corresponding digit, and its digit is enabled. A *switch* statement together with a *flag* variable is used to determine which digit should be refreshed.

If on the other hand the source of the interrupt is the external event, then the event count (Cnt) is incremented by 1, and the INT0 interrupt flag is cleared so that the processor can accept further interrupts from this source.

Function Display as before determines the bit pattern to be sent to the port to display a given number.

Modified Program

In Figure 6.4, the display shows leading zeroes, for example, number 14 is displayed as 0014. The program could easily be modified to blank leading zeroes. The modified program (called MIKROC-EVENT2.C) is shown in Figure 6.5.

Figure 6.6 shows an example display.

MPLAB XC8

The MPLAB XC8 version of the program is shown in Figure 6.7 (XC8-EVENT1.C). The operation of the program is the same as in Figure 6.5.

```
/*********************************************************************************
                    4-DIGIT 7-SEGMENT DISPLAY EVENT COUNTER
                    =======================================

In this project four common cathode 7-segment LED displays are connected to PORTD of a
PIC18F45K22 microcontroller and the microcontroller is operated from an 8 MHz crystal.
Four PORTA pins are used to enable/disable the LEDs. In addition, external interrupt input
INT0 (RB0) is used to receive external events. An event is assumed to occur if pin INT0 goes
from logic 0 to logic 1.

The program uses two ISR routines: the timer routine is used to refresh the Display every 5 ms.
External interrupt ISR is used to increment the event count. The event count is displayed
continuously on the 7-segment displays.

In this version of the program leading zeroes are blanked.

Author:       Dogan Ibrahim
Date:         August 2013
File:         XC8-EVENT2.C
*********************************************************************************/

#include <xc.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

#define DIGIT1 PORTAbits.RA0
#define DIGIT2 PORTAbits.RA1
#define DIGIT3 PORTAbits.RA2
#define DIGIT4 PORTAbits.RA3

unsigned int Cnt = 0;
unsigned char flag = 0;
unsigned int D1,D2,D3,D4,D5,D6;

//
// This function finds the bit pattern to be sent to the port to display a number
// on the 7-segment LED. The number is passed in the argument list of the function.
//
unsigned char Display(unsigned char no)
{
   unsigned char Pattern;
   unsigned char SEGMENT[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

   Pattern = SEGMENT[no];                          // Pattern to return
   return (Pattern);
}


//
// Interrupt Service Routine
//
```

Figure 6.7: MPLAB XC8 Program Listing.

```
void interrupt isr (void)
{
   if(INTCONbits.TMR0IF == 1)                    // If Timer interrupt occurred
   {
    TMR0L = 100;                                 // Reload timer register
    INTCONbits.TMR0IF = 0;                       // Clear timer interrupt flag
    switch(flag)
    {
     case 0:
     {
      DIGIT1 = 0;                                // Disable digit 1
      D1 = Cnt/1000;                             // 1000s  digit
      if(D1 != 0)                                // Check if blanking required
      {
        PORTD = Display(D1);                     // Send to PORTD
        DIGIT4 = 1;                              // Enable digit 4
      }
      flag = 1;
      break;
     }
     case 1:
     {
      DIGIT4 = 0;                                // Disable digit 4
      D2 = Cnt % 1000;
      D3 = D2/100;                               // 100s digit
      if(D3 != 0 || D1 != 0)                     // Check if blanking required
      {
        PORTD = Display(D3);                     // Send to PORTD
        DIGIT3 = 1;                              // Enable digit 3
      }
      flag = 2;
      break;
     }
     case 2:
     {
      DIGIT3 = 0;                                // Disable digit 3
      D4 = D2 % 100;
      D5 = D4/10;                                // 10s digit
      if(D5 != 0 || D3 != 0 || D1 != 0)          // Check if blanking is required
      {
        PORTD = Display(D5);                     // Send to PORTD
        DIGIT2 = 1;                              // Enable digit 2
      }
      flag = 3;
      break;
     }
     case 3:
     {
      DIGIT2 = 0;                                // Disable digit 2
      D6 = D4 % 10;
      PORTD = Display(D6);                       // Send to PORTD
```

Figure 6.7
cont'd

```
        DIGIT1 = 1;                                  // Enable digit 1
        flag = 0;
        break;
       }
     }
   }
   if(INTCONbits.INT0IF == 1)                        // If external interrupt occurred
   {
     Cnt++;                                          // Increment event count
     INTCONbits.INT0IF = 0;                          // Clear ext interrupt flag
   }
}

//
// Start of MAIN Program
//
void main()
{

   ANSELA = 0;                                       // Configure PORTA as digital
   ANSELD = 0;                                       // Configure PORTD as digital
   ANSELB = 0;                                       // Configure PORTB as digital
   TRISA = 0;                                        // Configure PORTA as outputs
   TRISD = 0;                                        // Configure PORTD as outputs
   TRISB = 1;                                        // RB0 is event input

   DIGIT1 = 0;                                       // Disable digit 1
   DIGIT2 = 0;                                       // Disable digit 2
   DIGIT3 = 0;                                       // Disable digit 3
   DIGIT4 = 0;                                       // Disable digit 4
 //
 // Configure TIMER0 interrupts
 //
   T0CON = 0xC5;                                     // TIMER0 in 8-bit mode
   TMR0L = 100;                                      // Load Timer register
 //
 // Configure External interrupts and enable interrupts
 //
   INTCON = 0xB0;

   for(;;)                                           // Endless loop
   {                                                 // Wait and process interrupts
   }
 }
```

Figure 6.7
cont'd

Project 6.2—Calculator with a Keypad and Liquid Crystal Display

Project Description

Keypads are small keyboards that are used to enter numeric or alphanumeric data to microcontroller systems. Keypads are available in a variety of sizes and styles from 2×2 to 4×4 or even bigger.

In this project, a 4×4 keypad and a liquid crystal display (LCD) are used, and a simple calculator is designed. Figure 6.8 shows the picture of the keypad used.

Figure 6.9 shows the structure of the keypad used in this project, which consists of 16 switches, formed in a 4×4 array, and named **0**–**9**, **Enter**, **+**,**−**,***** and **/**. Assuming that the keypad is connected to PORTC, the steps to detect which key is pressed is as follows:

- A logic 1 is applied to the first column via RC0.
- Port pins RC4–RC7 are read. If the data are nonzero, then a switch is pressed. If RC4 is 1, key 1 is pressed, if RC5 is 1, key 4 is pressed, if RC6 is 1, key 9 is pressed, and so on.
- A logic 1 is applied to the second column via RC1.
- Again Port pins RC4–RC7 are read. If the data are nonzero, then a switch is pressed. If RC4 is 1, key 2 is pressed, if RC5 is 1, key 6 is pressed, if RC6 is 1, key 0 is pressed, and so on.
- The above process is repeated for all the four columns continuously.

In this project, a simple integer calculator is designed. The calculator can add, subtract, multiply, and divide integer numbers and show the result on the LCD. The operation of

Figure 6.8: A 4 × 4 Keypad. (For color version of this figure, the reader is referred to the online version of this book.)

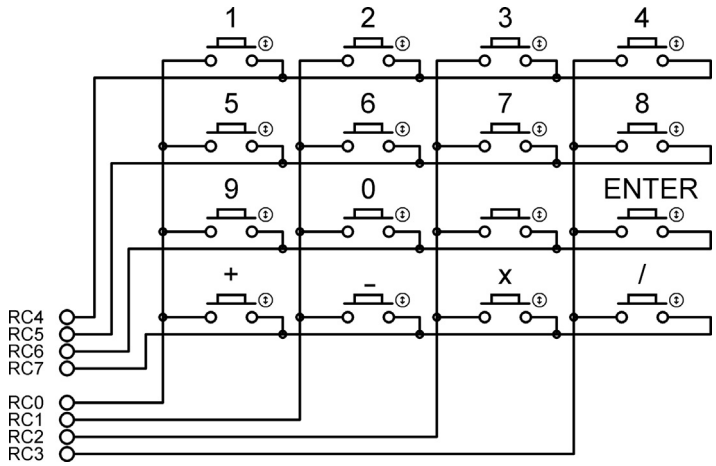


Figure 6.9: The 4 × 4 Keypad Structure.

the calculator is as follows: when power is applied to the system, the LCD displays text **CALCULATOR** for 2 s. Then, text **No1:** is displayed in the first row of the LCD, and the user is expected to type the first number and then press the **ENTER** key. Then, text **No2:** is displayed in the second row of the LCD where the user enters the second number and press the **ENTER** key. After this, the required operation key should be pressed. The result will be displayed on the LCD for 5 s, and then the LCD will be cleared, ready for the next calculation. The example below shows how numbers 12 and 20 can be added:

```
No1: 12 ENTER
No2: 20 ENTER
Op: +
Res = 32
```

In this project, the keypad is labeled as follows:

```
1   2   3   4
5   6   7   8
9   0       ENTER
+   –   X   /
```

One of the keys, between 0 and ENTER is not used in the project.

Project Hardware

The block diagram of the project is shown in Figure 6.10. The circuit diagram is given in Figure 6.11. A PIC18F45K22 microcontroller with an 8 MHz crystal is used in the project. Columns of the keypad are connected to port RC0–RC3, and rows are

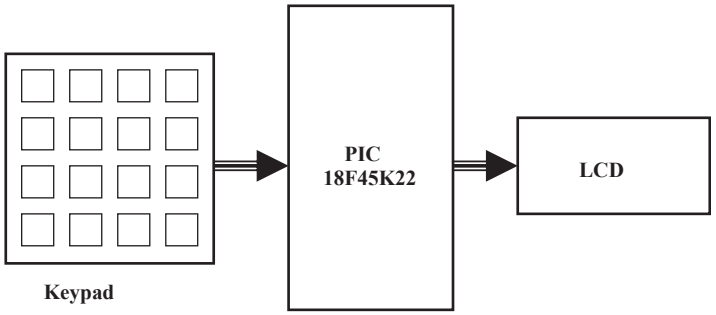


Figure 6.10: Block Diagram of the Project.

connected to port RC4–RC7 via pull-down resistors. The LCD is connected to PORTB in the default mode. An external reset button also provides to reset the microcontroller should it be necessary.

If you are using the EasyPIC V7 development board with the mikroElektronika 4×4 Keypad, then simply connect the keypad IDC10 ribbon cable connector to PORTC header at the edge of the board and enable the PORTC Pull-Down resistors by moving them downward (see Figure 6.11 for pull-down resistor requirements).

Project PDL

The project PDL is shown in Figure 6.12. The program consists of two parts: function **getkeypad** and the main program. Function **getkeypad** receives a key from the keypad. Inside the main program, PORTB is configured as the digital output, the LCD is initialized, and the heading "CALCULATOR" is displayed for 2 s. The program then executes in an endless loop. Inside this loop, two numbers and the required operation are received from the keypad. The microcontroller performs the required operation and displays the result on the LCD.

Project Program

The program listing (MIKROC-KEYPAD.C) is given in Figure 6.13. Each key is given a numeric value as follows:

```
0    1    2    3
4    5    6    7
8    9    10   11
12   13   14   15
```

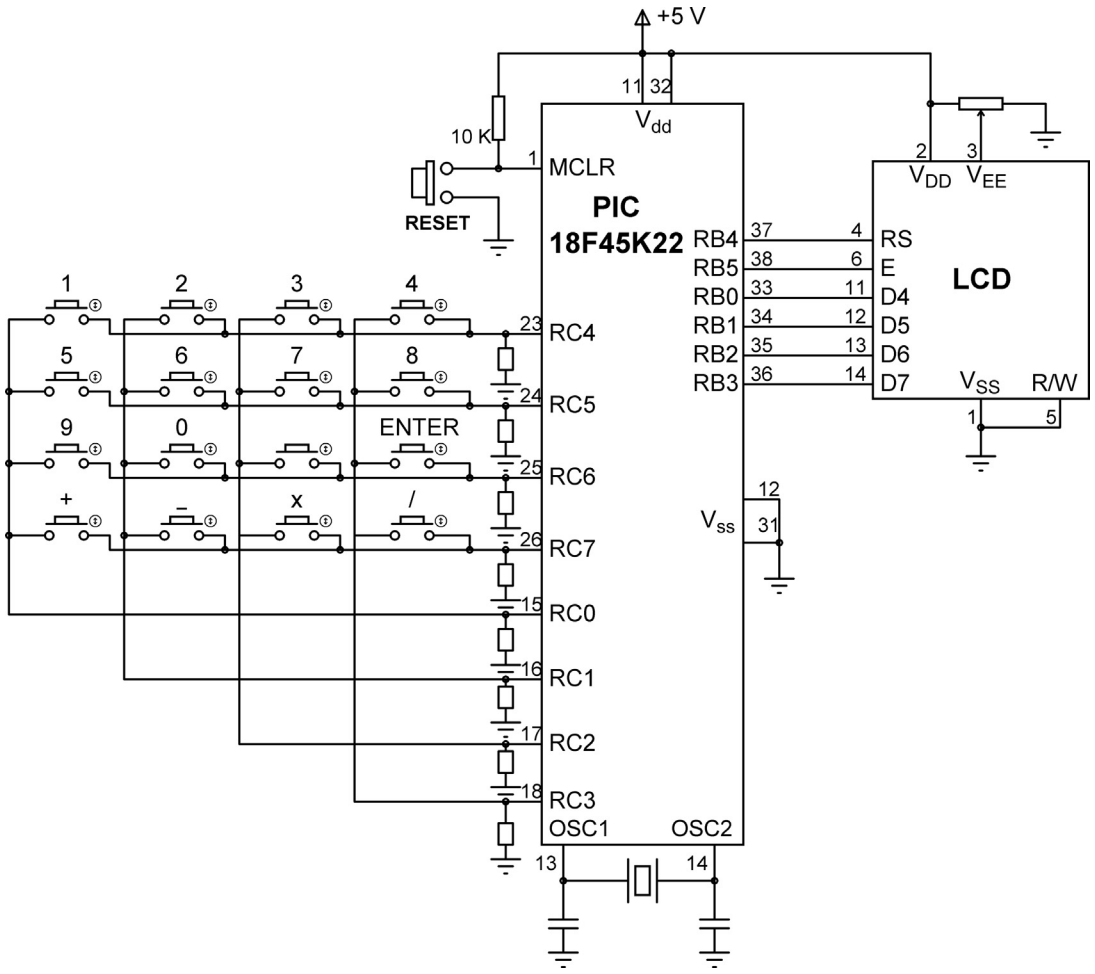


Figure 6.11: Circuit Diagram of the Project.

The program consists of a function called **getkeypad** that reads the pressed keys, and the main program. Variable **MyKey** stores the key value (0−15) pressed, variables **Op1** and **Op2** store the first and the second numbers entered by the user, respectively. All these variables are cleared to zero at the beginning of the program. A **while** loop is then formed to read the first number and store in variable **Op1**. This loop exits when the user presses the **ENTER** key. Similarly, the second number is read from the keyboard in a second **while** loop. Then, the operation to be performed is read and stored in variable **MyKey**, and a **switch** statement is used to perform the required operation and store the result in variable **Calc**. The result is converted into a string array using function **LongToStr** and leading blank characters are removed. The program displays the result on the LCD, waits

```
BEGIN
    Configure LCD connections
    Configure PORTB as digital output
    Initialize LCD
    Display heading
    Wait 2 s
    Clear heading
    DO FOREVER
        Display No1:
        Read first number
        Display No2:
        Read second number
        Display Op:
        Read operation
        Perform operation
        Display result
        Wait 5 s
     ENDDO
END


BEGIN/GETKEYPAD
    IF a key is pressed
        Get the key code (0 to 15)
        Return the key code
     ENDIF
END/GETKEYPAD
```

Figure 6.12: Project PDL.

for 5 s, and then clears the screen and is ready for the next calculation. This process is repeated forever.

Function **getkeypad** receives a key from the keypad. We start by sending a 1 to column 1, and then, we check all the rows. When a key is pressed, a logic 1 is detected in the corresponding row, and the program jumps out of the **while** loop. Then, a **for** loop is used to find the actual key pressed by the user as a number from 0 to 15.

It is important to realize that when a key is pressed or released, we get what is known as **contact noise** where the key output pulses up and down momentarily, and this produces a number of logic 0 and 1 pulses at the output. Switch contact noise is usually removed either in hardware or by programming, and this process is called **contact debouncing**. In software, the simplest way to remove the contact

```
/*************************************************************************
                        CALCULATOR WITH KEYPAD AND LCD
                        ==============================

In this project a 4 x 4 keypad is connected to PORTC of a PIC18F45K22 microcontroller. Also an
LCD is connected to PORTB. The project is a simple calculator which can perform integer arithmetic.

The keys are organised as follows:

  0   1   2   3
  4   5   6   7
  8   9  10  11
  12 13  14  15

The keys are labeled as follows:

  1  2  3  4
  5  6  7  8
  9  0     Enter
  +  –  *  /


Author:   Dogan Ibrahim
Date:     August 2013
File:     MIKROC-KEYPAD.C
*************************************************************/
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

#define MASK 0xF0
#define Enter 11
#define Plus 12
#define Minus 13
#define Multiply 14
#define Divide 15

//
// This function gets a key from the keypad and returns it to calling program
//
```

Figure 6.13: mikroC Pro for PIC Program Listing.

```
unsigned char getkeypad()
{
     unsigned char i, Key = 0;

     PORTC = 0x01;                                  // Start with column 1
     while((PORTC & MASK) == 0)                     // While no key pressed
     {
        PORTC = (PORTC << 1);                       // next column
        Key++;                                      // column number
        if(Key == 4)
        {
           PORTC = 0x01;                            // Back to column 1
           Key = 0;
        }
     }
     Delay_Ms(20);                                  // Switch debounce

     for(i = 0x10; i !=0; i <<=1)                   // Find the key pressed
     {
       if((PORTC & i) != 0)break;
       Key = Key + 4;
     }

     PORTC = 0x0F;
     while((PORTC & MASK) != 0);                    // Wait until key released
     Delay_Ms(20);                                  // Switch debounce

     return (Key);                                  // Return key number
}



//
// Start of MAIN program
//
void main()
{
   unsigned char MyKey, i,j,op[12];
   unsigned long Calc, Op1, Op2;
   char *lcd;

   ANSELB = 0;                                      // Configure PORTB as digital
   ANSELC = 0;                                      // Configure PORTC as digital
   TRISB = 0;                                       // PORTB are outputs (LCD)
   TRISC = 0xF0;                                    // RC4–RC7 are inputs

//
// Configure LCD
//
     Lcd_Init();                                    // Initialize LCD
```

Figure 6.13
cont'd

```
    Lcd_Cmd(_LCD_CLEAR);                    // Clear LCD
    Lcd_Out(1,1,"CALCULATOR");              // Display CALCULATOR
    Delay_ms(2000);                         // Wait 2 s
    Lcd_Cmd(_LCD_CLEAR);                    // Clear display
//
// Program loop
//
   for(;;)                                  // Endless loop
   {
     MyKey = 0;
     Op1 = 0;
     Op2 = 0;

     Lcd_Out(1,1,"No1: ");                  // Display No1:
     while(1)                               // Get first no
     {
       MyKey = getkeypad();
       if(MyKey == Enter)break;             // If ENTER pressed
       MyKey++;                             // Key number pressed
       if(MyKey == 10)MyKey = 0;            // If 0 key pressed
       Lcd_Chr_Cp(MyKey + '0');
       Op1 = 10 * Op1 + MyKey;              // First number in Op1
     }

     Lcd_Out(2,1,"No2: ");                  // Display No2:
     while(1)                               // Get second no
     {
       MyKey = getkeypad();
       if(MyKey == Enter)break;             // If ENTER pressed
       MyKey++;
       if(MyKey == 10)MyKey = 0;            // If 0 key pressed
       Lcd_Chr_Cp(MyKey + '0');
       Op2 = 10 * Op2 + MyKey;              // Second number in Op2
     }

     Lcd_Cmd(_LCD_CLEAR);                   // Clear LCD
     Lcd_Out(1,1,"Op: ");                   // Display Op:

     MyKey = getkeypad();                   // Get operation
     Lcd_Cmd(_LCD_CLEAR);
     Lcd_Out(1,1,"Res=");                   // Display Res=
     switch(MyKey)                          // Perform the operation
     {
      case Plus:
           Calc = Op1 + Op2;                // If ADD
           break;
      case Minus:
           Calc = Op1 – Op2;                // If Subtract
           break;
      case Multiply:
           Calc = Op1 * Op2;                // If Multiply
           break;
```

Figure 6.13
cont'd

```
            case Divide:
                Calc = Op1/Op2;                    // If Divide
                break;
            }

            LongToStr(Calc, op);                   // Convert to string in op
            lcd = Ltrim(op);                       // Remove leading blanks

            Lcd_Out_Cp(lcd);                       // Display result
            Delay_ms(5000);                        // Wait 5 s
            Lcd_Cmd(_LCD_CLEAR);                   /  / Clear LCD
        }
    }
```

Figure 6.13
cont'd

noise is to wait for about 20 ms after a switch key is pressed, and also after a switch key is released. In this project, **contact debouncing** is done in function **getkeypad**.

Program Using Built-in Keypad Function

In the program listing in Figure 6.13, a function called **getkeypad** has been developed to read a key from a keypad. mikroC Pro for PIC language has a built-in keypad library with functions **Keypad_Key_Press** and **Keypad_Key_Click** to read a key from a keypad when a key is pressed. The returned key has the code 1−16 (note that the returned key number is not from 0 to 15). Figure 6.14 shows a modified program (MIKROC-KEYPAD2.C) listing using the **Keypad_Key_Click** function to implement the calculator project. The circuit diagram is the same as in Figure 6.11.

Before using any keypad function, we have to call the **Keypad_Init** function to initialize the keypad library. Also, the connection port of the keypad must be declared at the beginning of the program. In this project, the keypad is connected to PORTC, and the following declaration must be made at the beginning of the program:

```
char keypadPort at PORTC;
```

```
/*************************************************************************
                        CALCULATOR WITH KEYPAD AND LCD
                        ==============================

In this project a 4 x 4 keypad is connected to PORTC of a PIC18F45K22 microcontroller. Also an
LCD is connected to PORTB. The project is a simple calculator which can perform integer
arithmetic.

The keys are labeled as follows:

  1  2  3  4
  5  6  7  8
  9  0     Enter
  +  –  *  /

In this version of the program built-in keypad libray is used.

Author:   Dogan Ibrahim
Date:     August 2013
File:     MIKROC-KEYPAD2.C
*************************************************************************/
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

// Keypad module connections
char keypadPort at PORTC;
// End of keypad module connections

#define Enter 12
#define Plus 13
#define Minus 14
#define Multiply 15
#define Divide 16

//
// Start of MAIN program
//
void main()
```

Figure 6.14: Modified mikroC Pro for PIC Program Listing.

```
{
    unsigned char MyKey, i,j,op[12];
    unsigned long Calc, Op1, Op2;
    char *lcd;

    ANSELB = 0;                                  // Configure PORTB as digital
    ANSELC = 0;                                  // Configure PORTC as digital
    TRISB = 0;                                   // PORTB are outputs (LCD)
    TRISC = 0xF0;                                // RC4–RC7 are inputs

    Keypad_Init();                               // Initialize keypad library
//
// Configure LCD
//
     Lcd_Init();                                 // Initialize LCD
     Lcd_Cmd(_LCD_CLEAR);
     Lcd_Out(1,1,"CALCULATOR");                  // Display CALCULATOR
     Delay_ms(2000);                             // Wait 2 s
     Lcd_Cmd(_LCD_CLEAR);                        // Clear display
//
// Program loop
//
    for(;;)                                      // Endless loop
    {
      MyKey = 0;
      Op1 = 0;
      Op2 = 0;

      Lcd_Out(1,1,"No1: ");                      // Display No1:
      while(1)                                   // Get first no
      {
       do
           MyKey = Keypad_Key_Click();
       while(!MyKey);
        if(MyKey == Enter)break;                 // If ENTER pressed
        if(MyKey == 10)MyKey = 0;                // If 0 key pressed
        Lcd_Chr_Cp(MyKey + '0');
        Op1 = 10 * Op1 + MyKey;                  // First number in Op1
      }

      Lcd_Out(2,1,"No2: ");                      // Display No2:
      while(1)                                   // Get second no
      {
       do
           MyKey = Keypad_Key_Click();
       while(!MyKey);
        if(MyKey == Enter)break;                 // If ENTER pressed
        if(MyKey == 10)MyKey = 0;                // If 0 key pressed
        Lcd_Chr_Cp(MyKey + '0');
        Op2 = 10 * Op2 + MyKey;                   // Second number in Op2
      }
```

Figure 6.14
cont'd

```
        Lcd_Cmd(_LCD_CLEAR);                    // Clear LCD
        Lcd_Out(1,1,"Op: ");                    // Display Op:

        do
            MyKey = Keypad_Key_Click();         // Get operation
        while(!MyKey);
        Lcd_Cmd(_LCD_CLEAR);
        Lcd_Out(1,1,"Res=");                    // Display Res=
        switch(MyKey)                           // Perform the operation
        {
         case Plus:
              Calc = Op1 + Op2;                 // If ADD
              break;
         case Minus:
              Calc = Op1 – Op2;                 // If Subtract
              break;
         case Multiply:
              Calc = Op1 * Op2;                 // If Multiply
              break;
         case Divide:
              Calc = Op1/Op2;                   // If Divide
              break;
        }

        LongToStr(Calc, op);                    // Convert to string in op
        lcd = Ltrim(op);                        // Remove leading blanks

        Lcd_Out_Cp(lcd);                        // Display result
        Delay_ms(5000);                         // Wait 5 s
        Lcd_Cmd(_LCD_CLEAR);                    // Clear LCD
    }
}
```

Figure 6.14
cont'd

MPLAB XC8

The program listing for the MPLAB XC8 version of the program is shown in Figure 6.15. Note here that the default LCD connections are slightly different when using the MPLAB XC8 compiler, where RB4 is connected to E instead of RS, and RB5 is connected to RS instead of E. Also, the RW pin of the LCD is connected to RB6 of the microcontroller.

```
/*************************************************************************
                        CALCULATOR WITH KEYPAD AND LCD
                        ==============================

In this project a 4 x 4 keypad is connected to PORTC of a PIC18F45K22 microcontroller. Also an
LCD is connected to PORTB. The project is a simple calculator which can perform integer arithmetic.

The LCD is connected to the microcontroller as follows:

Microcontroller         LCD
===============         ===

  RB0                   D4
  RB1                   D5
  RB2                   D6
  RB3                   D7
  RB4                   E
  RB5                   R/S
  RB6                   RW

The keys are organised as follows:

  0   1    2    3
  4   5    6    7
  8   9   10   11
  12  13  14   15

The keys are labeled as follows:

  1  2  3  4
  5  6  7  8
  9  0     Enter
  +  –  *  /


Author:   Dogan Ibrahim
Date:     August 2013
File:     XC8-KEYPAD.C
*************************************************************************/

#include <xc.h>
#include <stdlib.h>
#include <plib/xlcd.h>
#include <plib/delays.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

#define MASK 0xF0
#define Enter 11
#define Plus 12
#define Minus 13
```

Figure 6.15: MPLAB XC8 Program Listing.

```
#define Multiply 14
#define Divide 15


//
// This function creates seconds delay. The argument specifies the delay time in seconds.
//
void Delay_Seconds(unsigned char s)
{
   unsigned char i,j;

   for(j = 0; j < s; j++)
   {
      for(i = 0; i <  100; i++)__delay_ms(10);
   }
}

//
// This function creates 18 cycles delay for the xlcd library
//
void DelayFor18TCY( void )
{
Nop(); Nop(); Nop(); Nop();
Nop(); Nop(); Nop(); Nop();
Nop(); Nop(); Nop(); Nop();
Nop(); Nop();
return;
}


//
// This function creates 15 ms  delay for the xlcd library
//
void DelayPORXLCD( void )
{
   __delay_ms(15);
   return;
}


//
// This function creates 5 ms delay for the xlcd library
//
void DelayXLCD( void )
{
   __delay_ms(5);
   return;
}

//
// This function clears the screen
//
```

Figure 6.15
cont'd

```
void LCD_Clear()
{
   while(BusyXLCD());
   WriteCmdXLCD(0x01);
}

//
// This function moves the cursor to position row,column
//
void LCD_Move(unsigned char row, unsigned char column)
{
 char ddaddr = 40 * (row–1) + column;
 while( BusyXLCD() );
 SetDDRamAddr( ddaddr );
}


//
// This function gets a key from the keypad and returns it to calling program
//
unsigned char getkeypad()
{
      unsigned char i, Key = 0;

      PORTC = 0x01;                                    // Start with column 1
      while((PORTC & MASK) == 0)                       // While no key pressed
      {
         PORTC = (PORTC << 1);                         // next column
         Key++;                                        // column number
         if(Key == 4)
         {
            PORTC = 0x01;                              // Back to column 1
            Key = 0;
         }
      }
      __delay_ms(20);                                  // Switch debounce

      for(i = 0x10; i !=0; i <<=1)                     // Find the key pressed
      {
        if((PORTC & i) != 0)break;
        Key = Key + 4;
      }

      PORTC = 0x0F;
      while((PORTC & MASK) != 0);                      // Wait until key released
      __delay_ms(20);                                  // Switch debounce

      return (Key);                                    // Return key number
}
```

Figure 6.15
cont'd

```
//
// Start of MAIN program
//
void main()
{
   unsigned char MyKey, i,j,op[10];
   unsigned long Calc, Op1, Op2;

   ANSELB = 0;                                  // Configure PORTB as digital
   ANSELC = 0;                                  // Configure PORTC as digital
   TRISB = 0;                                   // PORTB are outputs (LCD)
   TRISC = 0xF0;                                // RC4–RC7 are inputs

   OpenXLCD(FOUR_BIT & LINES_5X7);               // Initialize LCD
   while(BusyXLCD());                           // Wait if the LCD is busy
   WriteCmdXLCD(DON);                           // Turn Display ON
   while(BusyXLCD());                           // Wait if the LCD is busy
   WriteCmdXLCD(0x06);                          // Move cursor right
   putrsXLCD("CALCULATOR");                     // Display heading
   Delay_Seconds(2);                            // 2 s delay
   LCD_Clear();                                 // Clear display
//
// Program loop
//
   for(;;)                                      // Endless loop
   {
     MyKey = 0;
     Op1 = 0;
     Op2 = 0;

     LCD_Move(1,1);                             // Move to row = 1,column = 1
     putrsXLCD("No1: ");                        // Display No1:
     while(1)                                   // Get first no
     {
       MyKey = getkeypad();
       if(MyKey == Enter)break;                 // If ENTER pressed
       MyKey++;                                 // Key number pressed
       if(MyKey == 10)MyKey = 0;                // If 0 key pressed
       putcXLCD(MyKey + '0');
       Op1 = 10 * Op1 + MyKey;                  // First number in Op1
     }

     LCD_Move(2,1);                             // Move to row = 2,column = 1
     putrsXLCD("No2: ");                        // Display No2:
     while(1)                                   // Get second no
     {
       MyKey = getkeypad();
       if(MyKey == Enter)break;                 // If ENTER pressed
       MyKey++;
```

Figure 6.15
cont'd

```
        if(MyKey == 10)MyKey = 0;                    // If 0 key pressed
        putcXLCD(MyKey + '0');
        Op2 = 10 * Op2 + MyKey;                      // Second number in Op2
      }

      LCD_Clear();                                   // Clear LCD
      LCD_Move(1,1);                                 // Move to row = 1,column = 1
      putrsXLCD("Op: ");                             // Display Op:

      MyKey = getkeypad();                           // Get operation
      LCD_Clear();
      LCD_Move(1,1);
      putrsXLCD("Res=");                             // Display Res=
      switch(MyKey)                                  // Perform the operation
      {
       case Plus:
            Calc = Op1 + Op2;                        // If ADD
            break;
       case Minus:
            Calc = Op1 – Op2;                        // If Subtract
            break;
       case Multiply:
            Calc = Op1 * Op2;                        // If Multiply
            break;
       case Divide:
            Calc = Op1/Op2;                          // If Divide
            break;
      }

      ltoa(op, Calc, 10);                            // Convert to string in op
      putsXLCD(op);                                  // Display result
      Delay_Seconds(5);                              // Wait 5 s
      LCD_Clear();
    }
}
```

Figure 6.15
cont'd

Project 6.3—The High/Low Game

Project Description

This project uses a 4 × 4 keypad and an LCD to create the classical High/Low game. For those of you who do not know how to play the game, the rules for this version of the game are as follows:

- The computer will generate a secret random number between 1 and 32767.
- The top row of the LCD will display "Guess Now…".
- The player will try to guess what the number is by entering a number on the keypad and then pressing the ENTER key.

- If the guessed number is higher than the secret number, then the bottom row of the LCD will display "HIGH—Try Again".
- If the guessed number is lower than the secret number, then the bottom row of the LCD will display "LOW—Try Again".
- If the player guesses the number, then the bottom row will display "Well Done…".
- The program waits for 5 s, and the game restarts

Generating a Random Number

In our program, we will be generating a random integer number using the mikroC Pro for PIC library functions "srand" and "rand". Function "srand" must be called with an integer argument (or "seed") to prepare the random number generator library. Then, every time function "rand" is called a new random number will be generated between 1 and 32767. The *set of numbers* generated is the same if the program is restarted with the same "seed" applied to function "srand". Thus, if the game is restarted after resetting the microcontroller, the same set of numbers will be generated.

Block Diagram

The block diagram of the project is as in Figure 6.10.

The keys are organized on the keypad as shown below:

```
1 2 3 A
4 5 6 B
7 8 9 C
* 0 # D
```

The mikroC Pro for PIC returns the following numbers when a key is pressed on the keypad:

Key Pressed	Number Returned
1	1
2	2
3	3
A	4
4	5
5	6
6	7
B	8
7	9
8	10
9	11
C	12
*	13
0	14
#	15
D	16

We will be using key "C" as the ENTER key in our program. Also, we will be correcting the key numbering in our program so that, for example, when "7" is pressed on the keypad a 7 is returned and not a 9 as in the above table.

Circuit Diagram

The circuit diagram of the project is shown in Figure 6.16. The LCD is connected to PORTB as in the earlier projects. The rows and columns of the keypad are connected to PORTC.

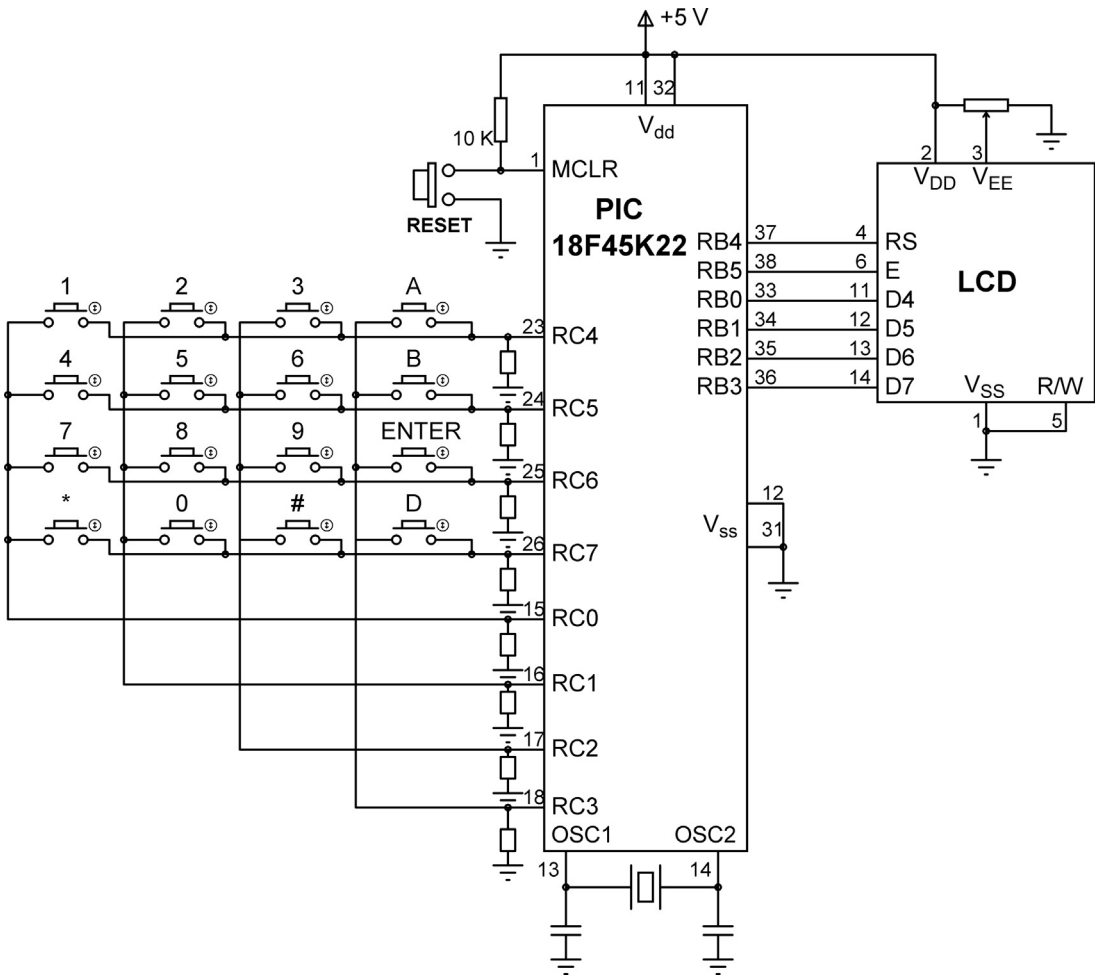


Figure 6.16: Circuit Diagram of the Project.

```
BEGIN
    Declare keypad port number
    Define LCD to microcontroller pin connections
    Configure PORTB as digital
    Initialize keypad library
    Initialize LCD
    Display heading "High/Low Game"
    Set new game flag
    Wait 2 s
    DO FOREVER
        IF new game flag is set THEN
            Clear LCD
            Turn OFF cursor
            Generate a random number (secret number)
            Display "Guess Now.." on row 1
        ENDIF
        Read and display (on row 2) numbers until ENTER is pressed
        IF entered number > secret number THEN
            Display "HIGH—Try Again"
            Wait 1 s
            Clear second row of LCD
        ELSE IF entered number < secret number THEN
            Display "LOW—Try Again"
            Wait 1 s
            Clear second row of LCD
        ELSE IF entered number = secret number THEN
            Display "Well Done.."
            Wait 5 s
            Set new game flag
        ENDIF
END
```

Figure 6.17: PDL of the Project.

Project PDL

The PDL of this project is given in Figure 6.17.

Project Program

mikroC Pro for PIC

The mikroC Pro for PIC program is named MIKROC-HILO.C, and the program listing of the project is shown in Figure 6.18.

```
/*************************************************************************
                        High/Low Game Using Keypad and LCD
                        ==================================

This project implements the High/Low game using the 4 x 4 keypad and an LCD.
The program generates a random number between 1 and 32767 and expects the
player to guess the number. The LCD displays "Guess Now.." on top row of the
display.

The player then guesses the number by entering a number via the keypad and then
pressing the ENTER key. If the guessed number is bigger than the generated number
the message “HIGH—Try Again” will be displayed on the bottom row of the LCD.

If the guessed number is lower than the generated number then the message
“LOW—Try Again” will be generated on the bottom row of the LCD. If on the
other hand the player guesses the number correctly, the bottom row of the LCD
will display the message "Well Done..".

The game will re-start after 5 s delay.

The microcontroller in this project is PIC18F45K22 and is operated from an 8 MHz
crystal as before. An 4x4 keypad is connected to PORTC. The LCD is connected to
PORTB.

Programmer:    Dogan Ibrahim
Date:          September 2013
File:          MIKROC-HILO.C

*************************************************************************/
// Declare LCD connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End of LCD connections

// Declare keypad connection
char keypadPORT at PORTC;
// End of keypad connection

#define ENTER_KEY 12
```

Figure 6.18: Program Listing of the Project.

```
unsigned char kp, new_game;
unsigned int GuessNumber, PlayerNumber;
int Diff;


void main()
{
    unsigned char Txt1[4];

        ANSELB = 0;                                     // Configure PORT B as digital
        ANSELC = 0;                                     // Configure PORTC as digital
        Keypad_Init();                                  // Initialize keypad library
        Lcd_Init();                                     // Initialize LCD
        Lcd_Out(1, 1, "High/Low Game");                 // Display heading
        Delay_ms(2000);                                 // Wait 2 s

        new_game = 1;
        srand(5);                                       // Random number seed

    for(;;)                                             // DO FOREVER
        {
                if(new_game == 1)
            {
                        Lcd_Cmd(_LCD_CLEAR);            // Clear LCD
                        Lcd_Cmd(_LCD_CURSOR_OFF);       // Turn OFF cursor
                        Lcd_Out(1, 1, "Guess Now..");   // Display "Guess Now.."
                        GuessNumber = rand();           // Generate a random number
                }
                kp = 0;
                PlayerNumber = 0;
                Lcd_Out(2, 1, "");                      // Position cursor at 1,1
                while(kp != ENTER_KEY)                  // Until ENTER pressed
                {
          do
          {
            kp = Keypad_Key_Click();                    // Look for key press
          }while(!kp);

          if(kp != ENTER_KEY)                           // If not ENTER key
          {
                        if(kp > 4 && kp <9)kp = kp = kp–1;      // 5 is 4, 6 is 5....
            if(kp > 8 && kp < 12)kp = kp–2;             // 7 is 9, 8 is 10...
            if(kp == 14)kp = 0;                         // 0 is14
            PlayerNumber = 10 * PlayerNumber + kp;
            ByteToStr(kp, Txt1);
            Txt1[0] = Txt1[2];                          // Get the number
            Txt1[1] = '\0';                             // Make a string
            Lcd_Out_Cp(Txt1);                           // Display on LCD
          }
         }
```

Figure 6.18
cont'd

```
        Delay_ms(1000);                                  // Wait one s
        Diff = PlayerNumber - GuessNumber;               // Find the diff

        if(Diff > 0)                                     // Greater ?
        {
                        Lcd_Out(2, 1, "HIGH - Try Again");
                        new_game = 0;                    // Not a new game
                        Delay_ms(1000);
                        Lcd_Out(2,1,"            ");     // Clear second row
        }

        if(Diff < 0)                                     // Less ?
        {
                        Lcd_Out(2, 1, "LOW - Try Again");
                        new_game = 0;                    // Not a new game
                        Delay_ms(1000);
                        Lcd_Out(2,1,"            ");     // Clear second row
        }

        if(Diff == 0)                                    // Equal ?
        {
                        Lcd_Out(2,1, "Well Done..");
                        new_game = 1;                    // New game
                        Delay_ms(5000);                  // Wait 5 s to
            }                                            // re-start game
    }                                                    // End of for
}                                                        // End of program
```

Figure 6.18
cont'd

At the beginning of the program, keypad PORT is declared as PORTC, and some other variables used in the program are also declared. Then, PORTB is configured as a digital output, keypad library is initialized, the LCD is initialized, and message "High/Low Game" is displayed on the LCD. After a 2 s delay, the program continues in an endless loop.

If this is a new game, the LCD is cleared, and message "Guess Now…" is displayed in the first row of the LCD. Then, a random number is generated between 1 and 32767 by calling library function "rand", and this number is stored in variable "GuessNumber". Note that the "srand" library function must be called with an integer number before calling "rand".

The keypad is then checked, and numbers are received until the ENTER key (key C) is pressed. The key numbers are then adjusted such that if, for example, 4 is pressed, number 4 is used by the program instead of 5. Similarly, if key 0 is pressed, number 0 is used by the program instead of 14 returned by the keypad library routine. The numbers entered by

the player are displayed in the second row of the LCD as they are entered so that the player can see what he/she has entered. After the player presses the ENTER key, a 1 s delay is introduced. The number entered by the player is stored in variable "PlayerNumber" in decimal format.

The program then calculates the difference between the secret number in "GuessNumber" and the number entered by the player (in PlayerNumber). This difference is stored in variable "Diff".

If "Diff" is positive, that is, if the number entered by the player is greater than the secret number, then the program displays message "HIGH—Try Again", waits for a second, and clears the second row of the LCD, ready for the player to try another number.

Figure 6.19: Display from the Game—Start of the Game.

Figure 6.20: Display from the Game—User Guessed 258.

Figure 6.21: Display from the Game—The Guess was Low.

If "Diff" is negative, that is, if the number entered by the player is less than the secret number, then the program displays message "LOW—Try Again", waits for a second, and clears the second row of the LCD, ready for the player to try another number.

If "Diff" is 0, that is, if the number entered by the player is equal to the secret number, then the program displays message "Well Done…" waits for 5 s, and sets the "new_game" flag so that a new secret number can be generated by the program. The game continues as before.

Figures 6.19–6.21 show various displays from the game. Note that the keypad keys are not debounced in the keypad library, and sometimes, you may get double key strokes even though you press a key once. You should be firm and quick when pressing a key to avoid this from happening.

Project 6.4—Generating Waveforms

Project Description

This project demonstrates how various waveforms can be generated using a microcontroller. The following waveforms will be generated in this project: sawtooth wave, triangle wave, any arbitrary wave, sine wave, and square wave. The first three waveforms will be generated using a digital-to-analog converter (DAC).

Figure 6.22 shows the block diagram of a typical microcontroller-based waveform generation system. Here, the microcontroller generates the required waveform as a digital signal, and then, the DAC converts this signal into analog. In practical applications, a low-pass filter is used after the DAC to clean the signal and remove any high-frequency components.

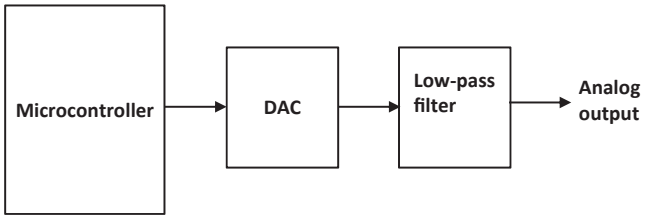


Figure 6.22: Block Diagram of Microcontroller-Based Waveform Generation.

Basically, two methods are used for waveform generation:

- The microcontroller calculates the waveform points in real-time and sends them to the DAC.
- The waveform points are stored in a look-up table. The microcontroller reads these points from the table and sends them to the DAC (this method is used to generate any arbitrary waveform, or to generate higher frequency waveforms).

As we shall see later, the rate at which the waveform points are sent to the DAC determines the frequency of the waveform.

Before going into the details of waveform generation, perhaps it is worthwhile to look at the operation of a DAC.

DAC Converter

A DAC converts a digital signal into an analog signal. The block diagram of a typical DAC is shown in Figure 6.23. This has a digital input, represented by D, analog output, represented by V_o, and a stable and accurate voltage reference, V_{ref}. In addition, some

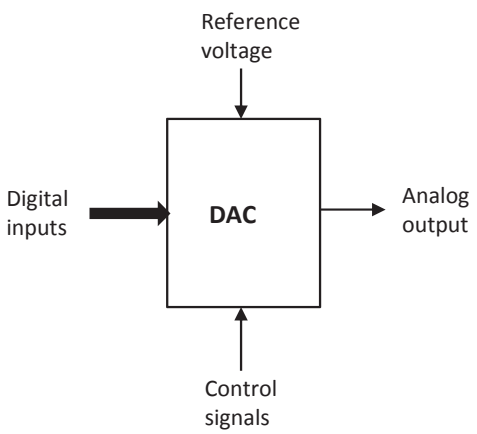


Figure 6.23: The Block Diagram of a Typical DAC.

control lines are also used, such as chip select, gain select, and so on. The digital input can either be in serial or in parallel form. In parallel converters, the width of the digital input is equal to the width of the converter. For example, a 12-bit converter has 12 input bits. Serial converters in general use the SPI or the I^2C bus, and basically, a clock and a data signal are used to send data to the converter. Parallel converters provide much faster conversion times, but they are housed in larger packages.

DACs are manufactured as either unipolar or bipolar as far as the output voltages are concerned. Unipolar converters can provide only positive output voltages, whereas bipolar converters provide both positive and negative voltages. In this book, we will only be using unipolar converters.

The relationship between the digital input−output and the voltage reference are given by

$$V_o = \frac{DV_{ref}}{2^n} \tag{6.1}$$

Where V_o is the output voltage, V_{ref} is the reference voltage, and n is the width of the converter. For example, in a 12-bit converter (resolution = 12 bits) with a +5 V reference voltage,

$$V_o = \frac{5D}{2^{12}} = 1.22D \text{ mV} \tag{6.2}$$

Thus, for example, if the input digital value is 1, the analog output voltage will be 1.22 mV, if the input value is 2, the analog output voltage will be 2.44 mV, and so on.

In this book, we shall be using a serial DAC for convenience and low cost. Most of the serial DACs use the SPI bus for communicating with a microcontroller. It is worth looking at the basic principles of the SPI bus before continuing with the project.

The SPI Bus

The SPI bus is one of the most commonly used protocols for serial communication between a microcontroller and a peripheral device. The SPI bus is a master−slave type bus protocol. In this protocol, one device (usually the microcontroller) is designated the *master*, and one or more other devices (usually sensors, converters, etc.) are designated *slaves*. In a minimum configuration, only one master and one slave are used. The master communicates with the slaves and controls all the activity on the bus.

Figure 6.24 shows a configuration with one master and three slaves. The SPI bus used three signals: clock (SCK), data in (SDI), and data out (SDO). The SDO of the master is connected to the SDIs of the slaves, and SDOs of the slaves are connected to the

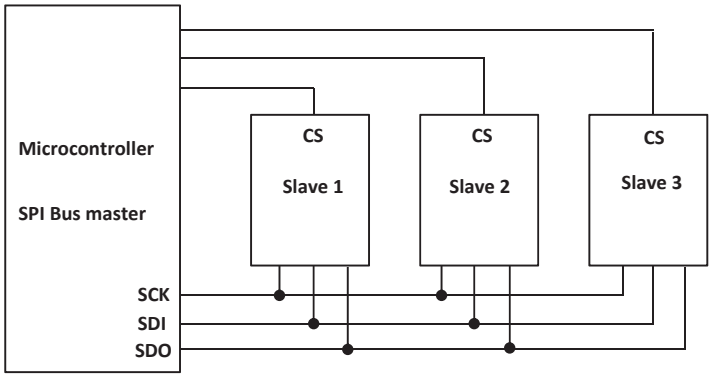


Figure 6.24: SPI Bus with One Master and Multiple Slaves.

SDI of the master. The master generates the SCK signals to enable data to be transferred on the bus. In every clock pulse, 1 bit of data is moved from the master to the slave, or from the slave to the master. The communication is only between a master and a slave, and the slaves cannot communicate with one an other. It is important to note that only one slave can be active at any time since there is no mechanism to identify the slaves. Thus, slave devices have enable lines (e.g. CS), which are normally controlled by the master. A typical communication between a master and several slaves can be as follows:

- Master enables slave 1.
- Master sends SCK signals to read or write data to slave 1.
- Master disables slave 1 and enables slave 2.
- Master sends SCK signals to read or write data to slave 2.
- The above process continues as required.

PIC18F microcontrollers provide one or more sets of special SPI bus compatible pins to enable the microcontroller to be connected to SPI slave peripheral devices. mikroC Pro for PIC and MPLAB XC8 compilers both provide SPI libraries to simplify programming and communication on the SPI bus.

Generating Sawtooth Waveform

In this part of the project, we will be generating a sawtooth waveform with the following specifications:

Output voltage	0 to +5 V
Frequency	100 Hz (period: 10 ms)
Step size	0.1 ms

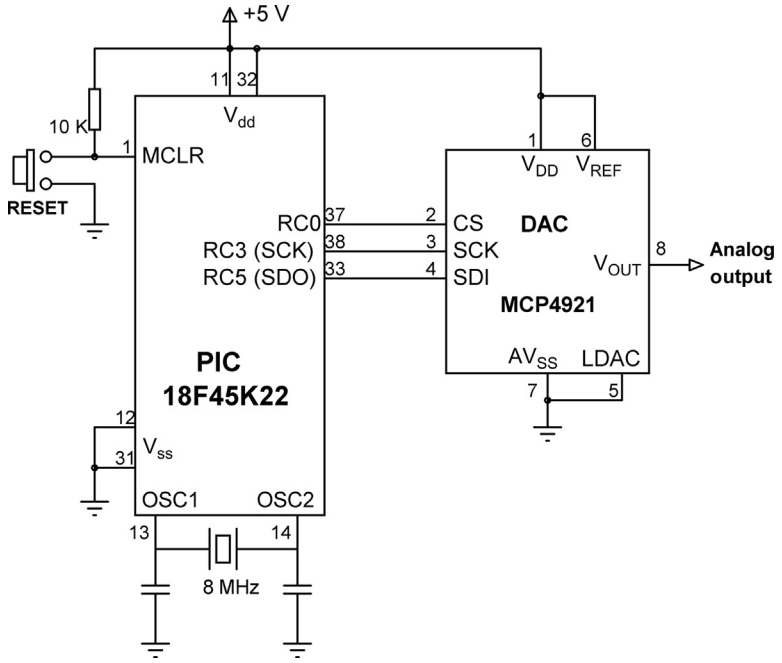


Figure 6.25: Circuit Diagram of the Project.

The circuit diagram of the project is shown in Figure 6.25. An MCP4921-type serial DAC is connected to the SPI port (PORTC) of a PIC18F45K22 microcontroller. The following connection is used between the microcontroller and the DAC:

Microcontroller Pin	DAC Pin
RC3 (SCK)	SCK
RC5 (SDO)	SDI
RC0	CS

If you are using the EasyPIC V7 development board and the 12-Bit DAC board, just plug in the DAC board to the PORTC IDC10 connector of the development board and configure the DAC board DIL jumper for the above connections. Also, set the reference voltage jumper on the DAC board to +5 V.

MCP4921 is a 12-bit serial DAC manufactured by Microchip Inc., having the following basic specifications:

- A 12-bit resolution,
- Up to a 20-MHz clock rate (SPI),
- Fast settling time of 4.5 μs,
- Unity or 2x gain output,
- External V_{ref} input,

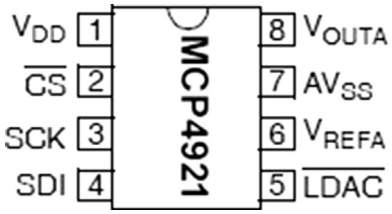


Figure 6.26: Pin Layout of MCP4921 DAC.

- A 2.7−5.5 V operation,
- Extended temperature range (−40 to +125 °C),
- An 8-pin DIL package.

Figure 6.26 shows the pin layout of the MCP4921. The pin definitions are as follows:

V_{DD}, AV_{SS}	power supply and ground
CS	chip select (LOW to enable the chip)
SCK,SDI	SPI bus clock and data in
V_{OUTA}	analog output
V_{REFA}	reference input voltage
LDAC	DAC input latch (transfers the input data to the DAC registers. Normally tied to ground so that CS controls the data transfer).

Data are written to the DAC in 2 bytes. The lower byte specified bits D0:D8 of the digital input data. The upper byte consists of the following bits:

D8:D11	bits D8:D11 of the digital input data
SHDN	1: output power down mode, 0: disable output buffer
GA	output gain control. 0: gain is 2x, 1: gain is 1x
BUF	0: input unbuffered, 1: input buffered
A/B	0: write to DAC_A, 1: write to DAC_B (MCP4921 supports only DAC_A)

Project PDL

The PDL of the project is shown in Figure 6.27.

Project Program

mikroC Pro for PIC

The mikroC Pro for the PIC program is called MIKROC-WAVE1.C and is given in Figure 6.28. At the beginning of the program, the CS enable input of the DAC is defined. PORTC is configured as digital, and the SPI bus module is initialized using built-in function SPI_Init. Inside the main program, an endless loop is formed, and the steps of the sawtooth waveform are sent out inside a *for* loop. Since there are 11 steps

```
BEGIN
        Define DAC port
        Configure PORTC as digital
        Initialize SPI module
        DO FOREVER
                Generate 11 step sawtooth wave
                Send the steps to DAC
                Wait 909 ms
        ENDDO
END


BEGIN/DAC
        Enable DAC
        Send high byte with 1x gain
        Send low byte
        Disable DAC
END/DAC
```

Figure 6.27: PDL of the Project.

(0−10) in the waveform and the required frequency is 100 Hz, that is, period 10 ms, then the duration of each step should be 10,000/11 = 909 μs. The delay_us function is used to generate the required delay. Function DAC receives the digital data (0−4095) in its argument and sends the data to the DAC. The chip is enabled (CS = 0), and the high byte is sent out first by setting the output gain to 1x. Then, the low byte is sent through the SPI bus.

Figure 6.29 shows the output waveform obtained using the PSCGU250 digital oscilloscope. Here, the vertical axis is 1 V per division, and the horizontal axis is 5 ms per division. The graph is moved down the 0 V point for clarity. Note that the period of the waveform is around 13 ms (i.e. frequency of about 77 kHz, and not 100 Hz). This is because of the delay caused by the statements inside the *for* loop. We will see in the next section how to improve the frequency.

mikroC Pro for the PIC SPI library supports the following functions ("x" in these functions is either 1 or 2, and it designates the SPI module number to be used):

SPIx_Init: Initialize the SPI module in the master mode with $F_{OSC}/4$ clock, data transmitted on low to high edge, and data sampled at the middle of the interval.

SPIx_Init_Advanced: Similar to SPI_Init, but various initialization parameters can be selected.

SPIx_Read: Read a byte from the SPI bus.

SPIx_Write: Write a byte via the SPI bus.

```
/*************************************************************************
                        Sawtooth Waveform Generation
                        ============================

This project shows how a sawtooth wavefor with specified frequency can be generated using a
microcontroller. The PIC18F45K22 microcontroller is used with an 8 MHz crystal.

The generated sawtooth waveform has amplitude 0 to +5 V and frequency of 100 Hz (period: 10 ms).

The MCP4921 DAC chip is used to convert the generated signal into analog.  This is a 12-bit
converter controlled with the SPI bus signals.

Programmer:   Dogan Ibrahim
Date:         September 2013
File:         MIKROC-WAVE1.C

*************************************************************************/
// DAC module connections
sbit Chip_Select at RC0_bit;
sbit Chip_Select_Direction at TRISC0_bit;
// End DAC module connections

//
// This function sends 12 bits digital data to the DAC. The data is passed
// through the function argument called "value"
//
void DAC(unsigned int value)
{
 char temp;

 Chip_Select = 0;                                  // Enable DAC chip

 // Send High Byte
 temp = (value >> 8) & 0x0F;                       // Store bits 8:11 to temp
 temp |= 0x30;                                     // Define DAC setting (choose 1x gain)
 SPI1_Write(temp);                                 // Send high byte via SPI

 // Send Low Byte
 temp = value;                                     // Store bits 0:7 to temp
 SPI1_Write(temp);                                 // Send low byte via SPI

 Chip_Select = 1;                                  // Disable DAC chip
}



void main()
{
   float i;
   unsigned int DAC_Value;
```

Figure 6.28: mikroC Pro for PIC Program Listing.

```
    ANSELC = 0;                                  // Configure PORTC as digital
    Chip_select = 1;                             // Disable DAC
    Chip_Select_Direction = 0;                   // Set CS as output
    SPI1_Init();                                 // Initialize SPI module
  //
  // Generate the Sawtooth waveform
  //
    for(;;)                                      // Endless loop
    {
      for(i = 0; i <= 1; i = i + 0.1)            // Generate waveform steps
      {
        DAC_Value = i * 4095;
        DAC(DAC_Value);                          // Send to DAC converter
        Delay_us(909);                           // Wait 909 ms
      }
    }
}
```

Figure 6.28
cont'd

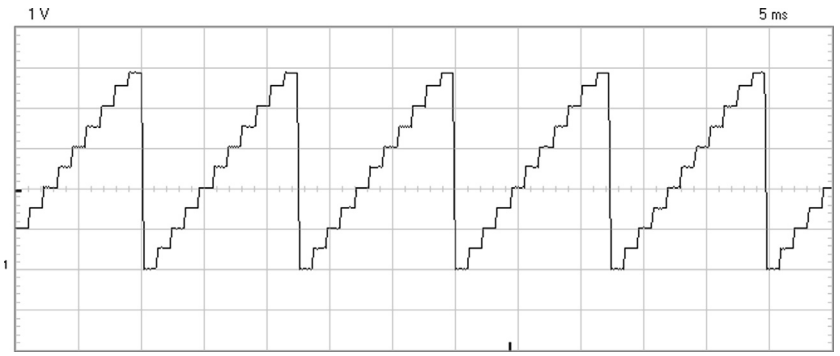


Figure 6.29: The Generated Sawtooth Waveform.

SPI_Set_Active: Sets the active SPI module to be used (for processors having more than one SPI module).

Modified Sawtooth Program

The program given in Figure 6.29 can be improved by using a timer to generate interrupts very close to 909 μs. Data can then be sent to the DAC inside the ISR.

Figure 6.30 shows the modified program (MIKROC-WAVE2.C). Timer0 is used in the 8-bit mode. To generate a delay of 909 μs, the value to be loaded into the timer register is calculated as (Project 6.1)

```
/*************************************************************************
                        Sawtooth Waveform Generation
                        ============================

This project shows how a sawtooth waveform with specified frequency can be generated using a
microcontroller. The PIC18F45K22 microcontroller is used with an 8 MHz crystal.

The generated sawtooth waveform consists of 11 steps from 0 to +5 V and has a frequency of
100 Hz (period of 10 ms).

The MCP4921 DAC chip is used to convert the generated signal into analog. This is a 12-bit
converter controlled with the SPI bus signals.

This version of the program uses Timer0 interrupts to generate the waveform with the specified
frequency.

Programmer:   Dogan Ibrahim
Date:         September 2013
File:         MIKROC-WAVE2.C

*************************************************************************/
// DAC module connections
sbit Chip_Select at RC0_bit;
sbit Chip_Select_Direction at TRISC0_bit;
// End DAC module connections

float Sample = 0.0;

//
// This function sends 12 bits digital data to the DAC. The data is passed
// through the function argument called "value"
//
void DAC(unsigned int value)
{
 char temp;

 Chip_Select = 0;                          // Enable DAC chip

 // Send High Byte
 temp = (value >> 8) & 0x0F;               // Store bits 8:11 to temp
 temp |= 0x30;                             // Define DAC setting (choose 1x gain)
 SPI1_Write(temp);                         // Send high byte via SPI

 // Send Low Byte
 temp = value;                             // Store bits 0:7 to temp
 SPI1_Write(temp);                         // Send low byte via SPI

 Chip_Select = 1;                          // Disable DAC chip
}

//
```

Figure 6.30: Modified Program.

```
// Timr interrupt service routine. Program jumps here at every 10 ms
//
void interrupt (void)
{
     unsigned int DAC_Value;

     TMR0L = 28;                                   // Reload timer register
     INTCON.TMR0IF = 0;                            // Clear timer interrupt flag
//
// Generate the Sawtooth waveform
//
      DAC_Value = Sample * 4095;
      DAC(DAC_Value);                              // Send to DAC converter
      Sample = Sample + 0.1;                       // Next sample
      if(Sample > 1.0)Sample = 0.0;
}


void main()
{
    ANSELC = 0;                                   // Configure PORTC as digital
    Chip_select = 1;                              // Disable DAC
    Chip_Select_Direction = 0;                     // Set CS as output
    SPI1_Init();                                  // Initialize SPI module
//
 // Configure TIMER0 interrupts
 //
    T0CON = 0xC2;                                 // TIMER0 in 8-bit mode
    TMR0L = 28;                                   // Load Timer register
    INTCON = 0xA0;                                // Enable global and Timer0 interrupts
    for(;;)                                       // Wait for interrupts
    {
    }
}
```

Figure 6.30
cont'd

$$\text{Time} = (4 \times \text{clock period}) \times \text{Prescaler} \times (256 - \text{TMR0L})$$

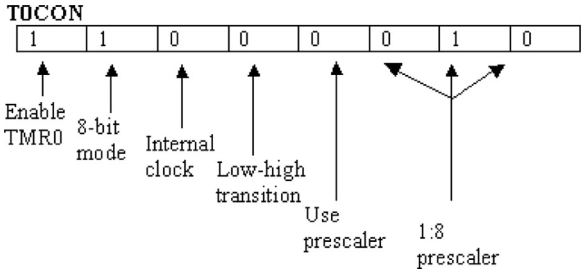
Where Prescaler is the selected prescaler value, and TMR0L is the value loaded into timer register TMR0L to generate timer interrupts for every Time period. In our application, the clock frequency is 8 MHz, that is, clock period = 0.125 μs and Time = 909 μs. Selecting a prescaler value of 8, the number to be loaded into TMR0L can be calculated as follows:

$$\text{TMR0L} = 256 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

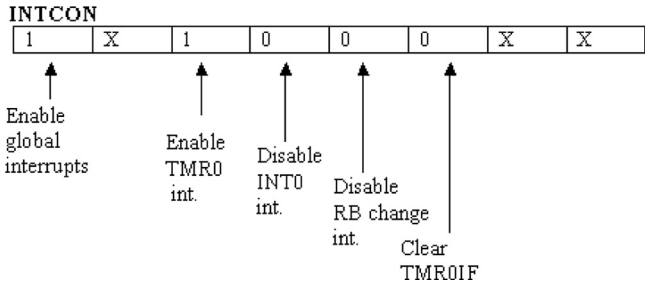
or

$$\text{TMR0L} = 256 - \frac{909}{4 * 0.125 * 8} = 28$$

Thus, TMR0L should be loaded with 28. The value to be loaded into the TMR0 control register T0CON can then be found as follows:



Thus, the T0CON register should be loaded with hexadecimal 0xC2. The next register to be configured is the interrupt control register INTCON:



Taking the do not care entries (X) as 0, the hexadecimal value to be loaded into register INTCON is thus 0xA0.

Figure 6.31 shows the new waveform with the vertical axis of 1 V per division and the horizontal axis of 5 ms per division. Clearly, this waveform has the specified period of 10 ms.

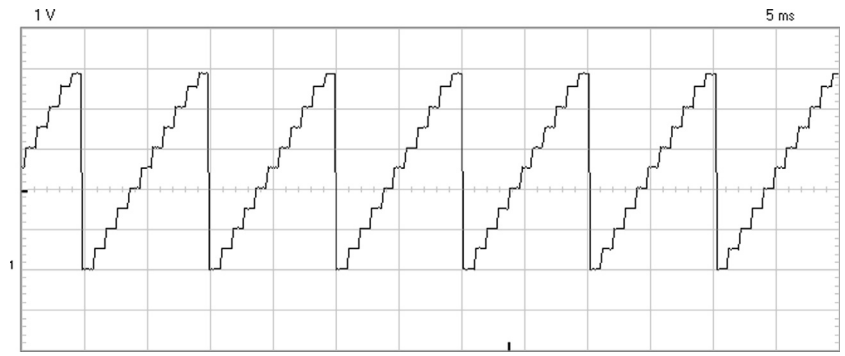


Figure 6.31: Generated Waveform.

MPLAB XC8

The MPLAB XC8 version of the program (XC8-WAVE2.C) is shown in Figure 6.32. MPLAB XC8 compiler supports the following SPI functions ("x" is 1 or 2, and it designates the SPI module number in multiple SPI processors. In single SPI processors, "x" can be omitted):

OpenSPIx: Initialize the SPIx module for SPI communication. This function takes three arguments as follows:

sync_mode

SPI_FOSC_4	Master mode, clock = FOSC/4
SPI_FOSC_16	Master mode, clock = FOSC/16
SPI_FOSC_64	Master mode, clock = FOSC/64
SPI_FOSC_TMR2	Master mode, clock = TMR2 output/2
SLV_SSON	Slave mode, /SS pin control enabled
SLV_SSOFF	Slave mode, /SS pin control disabled

bus_mode

MODE_00	Setting for SPI bus Mode 0,0 (clock is idle low, transmit on rising edge)
MODE_01	Setting for SPI bus Mode 0,1 (clock is idle low, transmit on falling edge)
MODE_10	Setting for SPI bus Mode 1,0 (clock is idle high, transmit on falling edge)
MODE_11	Setting for SPI bus Mode 1,1 (clock is idle high, transmit on rising edge)

smp_phase

SMPEND	Input data sample at the end of data out
SMPMID	Input data sample at the middle of data out
CloseSPIx	This function disables the SPIx module
DataRdySPIx	This function determines if a new value is available in the SPIx buffer. The Function returns 0 if there are no data and 1 if there are data
getcSPIx	This function reads a byte from the SPIx bus
getsSPIx	This function reads a string from the SPIx bus
putcSPIx	This function writes a byte to the SPIx bus
putsSPIx	This function writes a string to the SPIx bus
ReadSPIx	This function reads a byte from the SPIx bus
WriteSPIx	This function writes a byte to the SPIx bus

Note that the SPI signals SCK and SDO must be configured as outputs in the MPLAB XC8 version. The SPI bus is initialized using the OpenSPI function. Data are sent to the SPI bus using the WriteSPI function. The remainder of the program is the same as the mikroC Pro for the PIC version.

```
/*********************************************************************************
                                Sawtooth Waveform Generation
                                ============================

This project shows how a sawtooth waveform with specified frequency can be generated
using a microcontroller. The PIC18F45K22 microcontroller is used with an 8 MHz crystal.

The generated sawtooth waveform consists of 11 steps from 0 to +5 V and has a frequency
of 100 Hz (period of 10 ms).

The MCP4921 DAC chip is used to convert the generated signal into analog. This is a 12-bit
converter controlled with the SPI bus signals.

This program uses Timer0 interrupts to generate the waveform with the specified frequency.

Programmer:    Dogan Ibrahim
Date:          September 2013
File:          XC8-WAVE2.C

*********************************************************************************/

#include <xc.h>
#include <plib/spi.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

// DAC module connections
#define Chip_Select PORTCbits.RC0
#define Chip_Select_Direction TRISCbits.TRISC0
// End DAC module connections

// SPI bus signal directions
#define SCK_Direction TRISCbits.TRISC3
#define SDO_Direction TRISCbits.TRISC5
// End of SPI bus signal directions

float Sample = 0.0;

//
// This function sends 12 bits digital data to the DAC. The data is passed through the
// function argument called "value"
//
void DAC(unsigned int value)
{
 char temp;

 Chip_Select = 0;                                    // Enable DAC chip

 // Send High Byte
 temp = (value >> 8) & 0x0F;                         // Store bits 8:11 to temp
 temp |= 0x30;                                       // Define DAC setting (choose 1x gain)
```

Figure 6.32: MPLAB XC8 Program Listing.

```
 WriteSPI1(temp);                                  // Send high byte via SPI

 // Send Low Byte
 temp = value;                                     // Store bits 0:7 to temp
 WriteSPI1(temp);                                  // Send low byte via SPI

 Chip_Select = 1;                                  // Disable DAC chip
}


//
// Timer interrupt service routine. Program jumps here at every 909 ms
//
void interrupt isr (void)
{
     unsigned int DAC_Value;

     TMR0L = 28;                                   // Reload timer register
     INTCONbits.TMR0IF = 0;                        // Clear timer interrupt flag
//
// Generate the Sawtooth waveform
//
      DAC_Value = (unsigned int)(Sample * 4095);
      DAC(DAC_Value);                              // Send to DAC converter
      Sample = Sample + 0.1;                       // Next sample
      if(Sample > 1.0)Sample = 0.0;
}


void main()
{
   ANSELC = 0;
   Chip_Select_Direction = 0;
   SCK_Direction = 0;
   SDO_Direction = 0;

   Chip_Select = 1;                                // Disable DAC
   OpenSPI1(SPI_FOSC_4, MODE_00, SMPMID);          // Initialize SPI module

//
 // Configure TIMER0 interrupts
 //
   T0CON = 0xC2;                                   // TIMER0 in 8-bit mode
   TMR0L = 28;                                     // Load Timer register
   INTCON = 0xA0;                                  // Enable global and Timer0 interrupts
   for(;;)                                         // Wait for interrupts
   {
   }
}
```

Figure 6.32
cont'd

Generating Triangle Waveform

In this part of the project, we will be generating a triangle waveform with the following specifications:

Output voltage	0 to +5 V
Frequency	100 Hz (period: 10 ms)
Step size	0.1 ms

The circuit diagram of the project is as in Figure 6.25. Since the required period is 10 ms, the rising and falling parts of the waveform will each be 454 μs. The value to be loaded into the timer register should therefore change to

$$\text{TMR0L} = 256 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

or

$$\text{TMR0L} = 256 - \frac{454}{4 * 0.125 * 8} = 142$$

Figure 6.33 shows the program listing (MIKROC-WAVE3.C). The ISR code is also changed to generate the required triangle waveform.

The generated waveform is shown in Figure 6.34 using a PGSCU250 PC-based oscilloscope. The vertical axis is 1 V per division and the horizontal axis 5 ms per division.

Generating an Arbitrary Waveform

In this part of the project, we will be generating an arbitrary waveform. One period of the shape of the waveform will be sketched, and values of the waveform at different points will be extracted and loaded into a look-up table. The program will output the data points at the appropriate times to generate the required waveform.

The shape of one period of the waveform to be generated is shown in Figure 6.35. Assume that the required period is 20 ms (50 Hz).

```
/*************************************************************************
                          Triangle Waveform Generation
                          ============================

This project shows how a triangle waveform with specified frequency can be generated using a
microcontroller. The PIC18F45K22 microcontroller is used with an 8 MHz crystal.

The generated triangle waveform consists of 11 steps rising and 11 steps falling from 0V to +5V and
has a frequency of 100 Hz (period of 10 ms, 5 ms rising and 5 ms falling).

The MCP4921 DAC chip is used to convert the generated signal into analog.  This is a 12-bit
converter controlled with the SPI bus signals.

This version of the program uses Timer0 interrupts to generate the waveform with the specified
frequency.

Programmer:    Dogan Ibrahim
Date:          September 2013
File:          MIKROC-WAVE3.C

*************************************************************************/
// DAC module connections
sbit Chip_Select at RC0_bit;
sbit Chip_Select_Direction at TRISC0_bit;
// End DAC module connections

float Sample = 0.0, Inc = 0.1;

//
// This function sends 12 bits digital data to the DAC. The data is passed
// through the function argument called "value"
//
void DAC(unsigned int value)
{
  char temp;

  Chip_Select = 0;                              // Enable DAC chip

  // Send High Byte
  temp = (value >> 8) & 0x0F;                   // Store bits 8:11 to temp
  temp |= 0x30;                                 // Define DAC setting (choose 1x gain)
  SPI1_Write(temp);                             // Send high byte via SPI

  // Send Low Byte
  temp = value;                                 // Store bits 0:7 to temp
  SPI1_Write(temp);                             // Send low byte via SPI

  Chip_Select = 1;                              // Disable DAC chip
}

//
```

Figure 6.33: mikroC Pro for the PIC Program Listing.

```
// Timer interrupt service routineProgram jumps here at every 10 ms
//
void interrupt (void)
{
     unsigned int DAC_Value;

     TMR0L = 142;                                   // Reload timer register
     INTCON.TMR0IF = 0;                             // Clear timer interrupt flag
//
// Generate the Triangle waveform
//
      DAC_Value = Sample * 4095;
      DAC(DAC_Value);                               // Send to DAC converter
      Sample = Sample + Inc;                        // Next sample
      if(Sample > 1.0 || Sample < 0)
      {
        Inc = –Inc;
        Sample = Sample + Inc;
}     }


void main()
{
   ANSELC = 0;                                      // Configure PORTC as digital
   Chip_select = 1;                                 // Disable DAC
   Chip_Select_Direction = 0;                       // Set CS as output
   SPI1_Init();                                     // Initialize SPI module
//
 // Configure TIMER0 interrupts
 //
   T0CON = 0xC2;                                    // TIMER0 in 8-bit mode
   TMR0L = 142;                                     // Load Timer register
   INTCON = 0xA0;                                   // Enable global and Timer0 interrupts
   for(;;)                                          // Wait for interrupts
   {
   }
}
```

Figure 6.33
cont'd

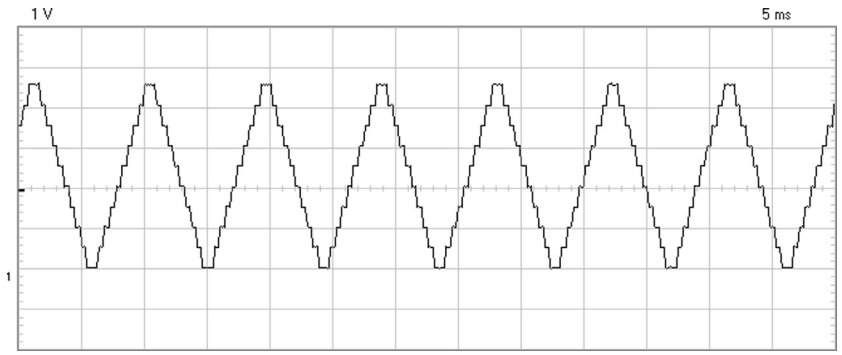


Figure 6.34: Generated Triangle Waveform.

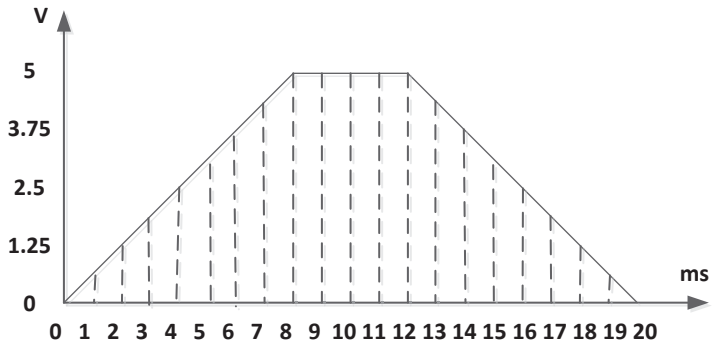


Figure 6.35: Waveform to be Generated.

The waveform takes the following values:

Time (ms)	Amplitude (V)	Time (ms)	Amplitude (V)
0	0	11	5.000
1	0.625	12	5.000
2	1.250	13	4.364
3	1.875	14	3.750
4	2.500	15	3.125
5	3.125	16	2.500
6	3.750	17	1.875
7	4.375	18	1.250
8	5.000	19	0.625
9	5.000	20	0
10	5.000		

The circuit diagram of the project is as given in Figure 6.25. Since the required period is 20 ms, each sample time is 1 ms. The value to be loaded into the timer register is found as follows:

$$\text{TMR0L} = 256 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

or

$$\text{TMR0L} = 256 - \frac{1000}{4 * 0.125 * 8} = 6$$

Figure 6.36 shows the mikroC Pro for PIC program listing (named MIKROC-WAVE4.C). At the beginning of the program, the waveform points are stored in a floating point array called Waveform, and a pointer called Sample is used to index this array. The timer is configured to generate interrupts at every millisecond. Inside the timer ISR, the waveform samples are sent to the DAC and the pointer Sample is incremented ready for the next sample. The remainder of the program is the same as in the previous project.

```
/*************************************************************************
                        Arbitrary Waveform Generation
                        =============================

This project shows how an arbitrary waveform with specified frequency can be generated using
a microcontroller. The PIC18F45K22 microcontroller is used with an 8 MHz crystal.

The generated waveform is first drawn, the waveform points extracted and stored in a floating
point array. In this example the period of the Waveform is 20 ms, defined using 20 Waveform
points.

The MCP4921 DAC chip is used to convert the generated signal into analog.  This is a 12-bit
converter controlled with the SPI bus signals.

This version of the program uses Timer0 interrupts to generate the waveform with the specified
frequency.

Programmer:    Dogan Ibrahim
Date:          September 2013
File:          MIKROC-WAVE4.C

*************************************************************************/
// DAC module connections
sbit Chip_Select at RC0_bit;
sbit Chip_Select_Direction at TRISC0_bit;
// End DAC module connections

unsigned char Sample = 0;
//
// Store the waveform points in an array
//
float Waveform[] = {0, 0.625, 1.250, 1.875, 2.5, 3.125, 3.750, 4.375, 5, 5,
            5, 5, 5, 4.375, 3.750, 3.125, 2.5, 1.875, 1.250, 0.625};

//
// This function sends 12 bits digital data to the DAC. The data is passed
// through the function argument called "value"
//
void DAC(unsigned int value)
{
 char temp;

 Chip_Select = 0;                                   // Enable DAC chip

 // Send High Byte
 temp = (value >> 8) & 0x0F;                        // Store bits 8:11 to temp
 temp |= 0x30;                                      // Define DAC setting (choose 1x gain)
 SPI1_Write(temp);                                  // Send high byte via SPI

 // Send Low Byte
 temp = value;                                      // Store bits 0:7 to temp
```

Figure 6.36: mikroC Pro for PIC Program Listing.

```
 SPI1_Write(temp);                                  // Send low byte via SPI

 Chip_Select = 1;                                   // Disable DAC chip
}

//
// Timer interrupt service routine. Program jumps here at every 10 ms
//
void interrupt (void)
{
     unsigned int DAC_Value;

     TMR0L = 6;                                     // Reload timer register
     INTCON.TMR0IF = 0;                             // Clear timer interrupt flag
//
// Generate the arbitrary waveform
//
      DAC_Value = Waveform[Sample] * 4095/5;
      DAC(DAC_Value);                               // Send to DAC converter
      Sample = Sample++;                            // Next Sample
      if(Sample == 20)Sample = 0;
}


void main()
{
   ANSELC = 0;                                      // Configure PORTC as digital
   Chip_select = 1;                                 // Disable DAC
   Chip_Select_Direction = 0;                       // Set CS as output
   SPI1_Init();                                     // Initialize SPI module
//
 // Configure TIMER0 interrupts
 //
   T0CON = 0xC2;                                    // TIMER0 in 8-bit mode
   TMR0L = 6;                                       // Load Timer register
   INTCON = 0xA0;                                   // Enable global and Timer0 interrupts
   for(;;)                                          // Wait for interrupts
   {
   }
}
```

Figure 6.36
cont'd

Figure 6.37 shows the waveform generated by the program. Here, the vertical axis is 1 V per division and the horizontal axis 5 ms per division.

Generating Sine Waveform

In this part of the project, we will see how to generate a low-frequency sine wave using the built-in trigonometric **sin** function, and then send the output to a DAC.

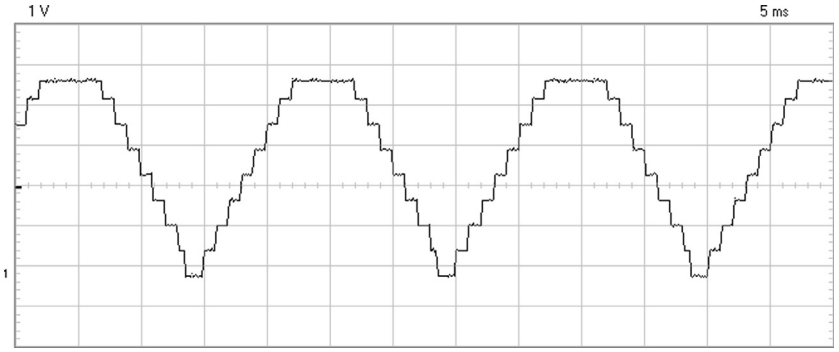


Figure 6.37: Generated Waveform.

The generated sine wave has an amplitude of 2 V, frequency of 50 Hz, and offset of 2.5 V.

The circuit diagram of the project is as in Figure 6.25 where the DAC is connected to PORTC of a PIC18F45K22 microcontroller.

The frequency of the sine wave to be generated is 50 Hz. This wave has a period of 20 ms, or 20,000 μs. If we assume that the sine wave will consist of 100 samples, then each sample should be output at 20,000/100 = 200 μs intervals. Thus, we will configure Timer0 to generate interrupts at every 200 μs, and inside the ISR, we will output a new sample of the sine wave. The sample values will be calculated using the trigonometric **sin** function of the compiler.

The sin function will have the following format:

$$\sin\left(\frac{2\pi \times \text{Count}}{\text{T}}\right)$$

where T is the period of the waveform and is equal to 100 samples. Count is a variable that ranges from 0 to 100 and is incremented by 1 inside the ISR every time a timer interrupt occurs. Thus, the sine wave is divided into 100 samples, and each sample is output at 200 μs. The above formula can be rewritten as follows:

$$\sin\left(0.0628 \times \text{Count}\right)$$

It is required that the amplitude of the waveform should be 2 V. With a reference voltage of +5 V and a 12-bit DAC converter (0−4095 quantization levels), 2 V is equal to decimal number 1638. Thus, we will multiply our sine function with the amplitude at each sample to give

$$1638 * \sin\left(0.0628 \times \text{Count}\right)$$

The D/A converter used in this project is unipolar and cannot output negative values. Therefore, an offset is added to the sine wave to shift it so that it is always positive. The offset should be larger than the absolute value of the maximum negative value of the sine wave, which is 1638 when the sin function above is equal to 1. In this project, we are adding a 2.5 V offset, which corresponds to a decimal value of 2048 at the DAC output. Thus, at each sample, we will calculate and output the following value to the DAC:

$$2048 + 2457 * \sin(0.0628 \times \text{Count})$$

The value to be loaded to Timer0 to generate interrupts at 200 μs intervals is found as (with a prescaler of 8):

$$\text{TMR0L} = 256 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

or

$$\text{TMR0L} = 256 - \frac{200}{4 * 0.125 * 8} = 206$$

mikroC Pro for PIC

Figure 6.38 shows the mikroC Pro for the PIC program listing (named MIKROC-WAVE5.C). At the beginning of the program, the chip select connection of the DAC chip is defined. Then, the sine wave amplitude is set to 1638, offset is set to 2048, and variable R is defined as $2\pi/100$. The chip select direction is configured as the output, DAC is disabled by setting its chip enable input, and SPI2 is initialized. The sine waveform values for a period are obtained offline outside the ISR using the following statement. The reason for calculating these values outside the ISR is to minimize the time inside the ISR so that higher frequency sine waves can be generated (it is also possible to generate higher frequency waveforms by increasing the clock frequency. For example, by enabling the clock PLL, the frequency can be multiplied by 4 to be 32 MHz):

```
for(i = 0; i < 100; i++)sins[i] = offset + Amplitude * sin(R * i);
```

The main program then configures Timer0 to generate interrupts at every 200 μs. Timer prescaler is taken as 8, and the timer register is loaded with 206. The main program then waits in an endless loop where the processing continues inside the ISR whenever the timer overflows.

Figure 6.39 shows the waveform generated by the program. It is clear from this figure that the generated sine waveform has period 20 ms as designed. Here, the vertical axis is 1 V per division, and the horizontal axis is 20 ms per division.

```
/*************************************************************************
                              GENERATE SINE WAVE
                               ==================

In this project a DAC is connected to the microcontroller output and a 50 Hz sine wave with an
amplitude of 2 V and an offset of 2.5 V is generated in real-time at the output of the DAC.

The MCP4921 12-bit DAC is used in the project. This converter can operate from +3.3 to +5 V,
with a typical conversion time of 4.5 ms. Operation of the DAC is based on the standard
SPI interface.

mikroC PRO for PIC trigonometric "sin" function is used to calculate the sine points.
50 Hz waveform has the period T = 20 ms, or, T = 20,000 us. If we take 100 points to sample
the sine wave, then each Sample occurs at 200 us. Therefore, we need a timer interrupt service
routine that will generate interrupts at every 200 us, and inside this routine we will calculate a
new sine point and send it to the DAC. The result is that we will get a 50 Hz sine wave. Because
the DAC is unipolar, we have to shift the output waveform to a level greater than its maximum
negative value so that the waveform is always positive and can be output by the DAC.


Author:     Dogan Ibrahim
Date:       September 2013
File:       MIKROC-WAVE5.C
*************************************************************************/
// DAC module connections
sbit Chip_Select at RC0_bit;
sbit Chip_Select_Direction at TRISC0_bit;
// End DAC module connections

#define T 100                                   // 100 samples
#define R 0.0628                                // 2 * PI/T
#define Amplitude 1638                          // 2 V * 4096/5 V
#define offset 2048                             //2.5 * 4096/5 V

unsigned char temp, Count = 0;
float Sample;
unsigned int Value;
float sins[100];

//
// This function sends 12 bits digital data to the DAC. The data is passed
// through the function argument called "value"
//
void DAC(unsigned int value)
{
 char temp;

 Chip_Select = 0;                               // Enable DAC chip

 // Send High Byte
 temp = (value >> 8) & 0x0F;                    // Store bits 8:11 to temp
```

Figure 6.38: mikroC Pro for PIC Program Listing.

```
 temp |= 0x30;                                   // Define DAC setting (choose 1x gain)
 SPI1_Write(temp);                               // Send high byte via SPI

 // Send Low Byte
 temp = value;                                    // Store bits 0:7 to temp
 SPI1_Write(temp);                               // Send low byte via SPI

 Chip_Select = 1;                                // Disable DAC chip
}

//
// Timer ISR. Program jumps here at every 200 ms
//
void interrupt (void)
{
    TMR0L = 206;                                 // Reload timer register
//
// Get sine wave samples and send to DAC
//
    Value = sins[Count];
    DAC(Value);                                  // Send to DAC converter
    Count++;
    if(Count == 100)Count = 0;
    INTCON.TMR0IF = 0;                           // Clear timer interrupt flag
}


void main()
{
   unsigned char i;
   ANSELC = 0;                                   // Configure PORTC as digital
   Chip_select = 1;                              // Disable DAC
   Chip_Select_Direction = 0;                    // Set CS as output
   SPI1_Init();                                  // Initialize SPI module
//
// Generate the sine wave samples offline and load into an array called sins
//
   for(i = 0; i < 100; i++)sins[i] = offset + Amplitude * sin(R * i);
//
 // Configure TIMER0 interrupts
 //
   T0CON = 0xC2;                                 // TIMER0 in 8-bit mode
   TMR0L = 206;                                  // Load Timer register
   INTCON = 0xA0;                                // Enable global and Timer0 interrupts
   for(;;)                                       // Wait for interrupts
   {
   }
}
```

Figure 6.38
cont'd

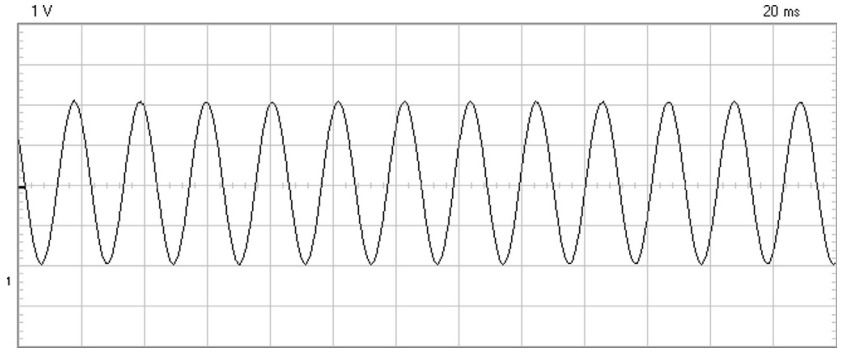


Figure 6.39: Generated Waveform.

Note that the code generated by the mikroC Pro for the PIC compiler is >2 K, and the licensed full version of the compiler is required to compile the program given in Figure 6.38.

MPLAB XC8

The MPLAB XC8 version of the program is shown in Figure 6.40 (XC8-WAVE5.C). In this program, the PLL clock multiplier is enabled by adding the PLLCFG = ON to the configuration register definition at the beginning of the program. The clock frequency (_XTAL_FREQ) is then changed to 32 MHz. The value to be loaded to Timer0 to generate interrupts at 200 μs intervals is calculated as (with a prescaler of 16):

$$\text{TMR0L} = 256 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

or

$$\text{TMR0L} = 256 - \frac{200}{4 * 0.03125 * 16} = 156$$

Note that the clock frequency is now 32 MHz, which has a period of 0.03125 μs.

Generating Square Waveform

The square wave is perhaps the easiest waveform to generate. If an accurate frequency is not required, then a loop can be formed to generate a square wave signal. For example, assuming we wish to generate 1 kHz (period = 1 ms) square wave signal on pin RB0 of

```
/*************************************************************************
                              GENERATE SINE WAVE
                              ==================

In this project a DAC is connected to the microcontroller output and a 50 Hz sine wave with an
amplitude of 2 V and an offset of 2.5 V is generated in real-time at the output of the DAC.

The MCP4921 12-bit DAC is used in the project. This converter can operate from +3.3 to +5 V,
with a typical conversion time of 4.5 ms. Operation of the DAC is based on the  standard
SPI interface.

mikroC PRO for PIC trigonometric "sin" function is used to calculate the sine points. 50 Hz
waveform has the period T = 20 ms, or, T = 20,000 us. If we take 100 points to sample the
sine wave, then each Sample occurs at 200 us. Therefore, we need a timer interrupt service
routine that will generate interrupts at every 200 us, and inside this routine we will calculate
a new sine point and send it to the DAC. The result is that we will get a 50 Hz sine wave.
Because the DAC is unipolar, we have to shift the output waveform to a level greater than
its maximum negative value so that the waveform is always positive and can be output by
the DAC.


Author:    Dogan Ibrahim
Date:      September 2013
File:      XC8-WAVE5.C
*************************************************************************/

#include <xc.h>
#include <math.h>
#include <plib/spi.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP, PLLCFG = ON
#define _XTAL_FREQ 32000000

// DAC module connections
#define Chip_Select PORTCbits.RC0
#define Chip_Select_Direction TRISCbits.TRISC0
// End DAC module connections

// SPI connections
#define SCK_Direction TRISCbits.RC3
#define SDO_Direction TRISCbits.RC5
//

#define T 100                                   // 100 samples
#define R 0.0628                                // 2 * PI/T
#define Amplitude 1638                          // 2 V * 4096/5 V
#define offset 2048                             //2.5 * 4096/5 V

unsigned char temp, Count = 0;
float Sample;
unsigned int Value;
float sins[100];
```

Figure 6.40: MPLAB XC8 Program Listing.

```
//
// This function sends 12 bits digital data to the DAC. The data is passed
// through the function argument called "value"
//
void DAC(unsigned int value)
{
 char temp;

 Chip_Select = 0;                                    // Enable DAC chip

 // Send High Byte
 temp = (value >> 8) & 0x0F;                         // Store bits 8:11 to temp
 temp |= 0x30;                                       // Define DAC setting (choose 1x gain)
 WriteSPI1(temp);                                    // Send high byte via SPI

 // Send Low Byte
 temp = value;                                       // Store bits 0:7 to temp
 WriteSPI1(temp);                                    // Send low byte via SPI

 Chip_Select = 1;                                    // Disable DAC chip
}

//
// Timer interrupt service routine. Program jumps here at every 200 us
//
void interrupt isr (void)
{
    TMR0L = 156;                                     // Reload timer register
//
// Get sine wave samples and send to DAC
//
    Value = (unsigned int)sins[Count];
    DAC(Value);                                      // Send to DAC converter
    Count++;
    if(Count == 100)Count = 0;
    INTCONbits.TMR0IF = 0;                           // Clear timer interrupt flag
}



void main()
{

   unsigned char i;
   ANSELC = 0;
   Chip_Select_Direction = 0;
   SCK_Direction = 0;
   SDO_Direction = 0;

   Chip_Select = 1;                                  // Disable DAC
```

Figure 6.40
cont'd

```
    OpenSPI1(SPI_FOSC_4, MODE_00, SMPMID);              // Initialize SPI module

  //
  // Generate the sine wave samples offline and load into an array called sins
  //
    for(i = 0; i < 100; i++)sins[i] = offset + Amplitude * sin(R * i);
  //
  // Configure TIMER0 interrupts
  //
    T0CON = 0xC3;                                       // TIMER0 in 8-bit mode
    TMR0L = 156;                                        // Load Timer register
    INTCON = 0xA0;                                      // Enable global and Timer0 interrupts
    for(;;)                                             // Wait for interrupts
    {
    }
  }
```

Figure 6.40
cont'd

the microcontroller with equal ON and OFF times, the following PDL shows how the signal can easily be generated:

```
BEGIN
   Configure RB0 as digital output
   DO FOREVER
      Set RB0 = 1
      Wait 0.5 ms
      Set RB0 = 0
      Wait 0.5 ms
   ENDDO
END
```

The problem with this code is that the generated frequency is not accurate because of two reasons: first, the built-in delay functions are not meant to be very accurate, and second, the time taken to execute the other instructions inside the loop are not taken into account.

An accurate square wave signal can be generated using a timer interrupt routine as described in the previous waveform generation projects. Inside the ISR, all we have to do is toggle the output pin where the waveform, is to be generated from. This is illustrated with an example. In this example a square wave signal with frequency 1 kHz (period = 1 ms) is generated from port pin RD0 of a PIC18F45K22 microcontroller.

The required mikroC Pro for the PIC program listing is shown in Figure 6.41 (MIKROC-WAVE6.C). Inside the main program, PORTD is configured as an analog

```
/*************************************************************************
                        GENERATE SQUARE WAVE
                        ====================

In this project a square wave signal with frequency 1 kHz is generated. The program uses timer
interrupts to send the signal to output.

Timer0 is configured to provide interrupts at every 0.5 ms. Inside the ISR the output pin is toggled,
thus generating a square wave signal.

Author:     Dogan Ibrahim
Date:       September 2013
File:       MIKROC-WAVE6.C
*************************************************************************/

#define OUT_PIN LATD.LATD0
//
// Timer ISR. Program jumps here at every 100 ms
//
void interrupt (void)
{
     TMR0L = 131;                              // Reload timer register
     OUT_PIN = ~OUT_PIN;                       // Toggle the output pin
     INTCON.TMR0IF = 0;                        // Clear timer interrupt flag
}


void main()
{
   ANSELD = 0;                                 // Configure PORTC as digital
   TRISD = 0;
   OUT_PIN = 0;
//
 // Configure TIMER0 interrupts
 //
   T0CON = 0xC2;                               // TIMER0 in 8-bit mode
   TMR0L = 131;                                // Load Timer register
   INTCON = 0xA0;                              // Enable global and Timer0 interrupts
   for(;;)                                     // Wait for interrupts
   {
   }
}
```

Figure 6.41: mikroC Pro for PIC Program Listing

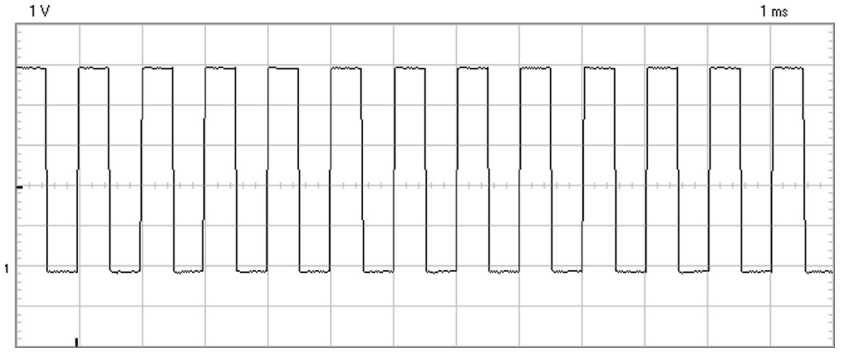


Figure 6.42: Generated Waveform.

output, and pin RD0 is configured as an output pin. Timer0 is configured to generate interrupts at every 500 μs. Inside the ISR, pin RD0 is toggled, and timer interrupt flag is cleared so that further interrupts can be accepted by the processor.

The value to be loaded to Timer0 to generate interrupts at 500 μs intervals is found as (with a prescaler of 8)

$$\text{TMR0L} = 256 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

or

$$\text{TMR0L} = 256 - \frac{500}{4 * 0.125 * 8} = 131$$

Figure 6.42 shows the waveform generated. Here, the vertical axis is 1 V per division, and the horizontal axis is 1 ms per division.

Another accurate method of generating a square wave is by using the PWM module of the microcontroller Chapter 2, Example 2.1. The PWM module makes use of ports CCP1, CCP2, CCP3, etc. on the microcontroller. But unfortunately, this module cannot be used to generate PWM signals with large periods such as 20 ms. The advantage of using the PWM module is that once configured this module works independent of the central processing unit (CPU), and thus, the CPU is free to do other tasks, while the PWM module is working.

An example is given in this section to show how to program the PWM module. In this program, it is required to generate a 20 kHz (period = 50 μs) square wave with equal ON and OFF times of 25 μs. The output signal will be available on pin RC2 (CCP1) of the

microcontroller. The required program listing is given in Figure 6.43 (MIKROC-WAVE7.C). The PWM module is configured as follows:

The value to be loaded into Timer2 register can be calculated as follows:

$$\text{PR2} = \frac{\text{PWM period}}{\text{TMR2PS} * 4 * \text{T}_{\text{osc}}} - 1$$

where

PR2 is the value to be loaded into the Timer2 register,
TMR2PS is the Timer2 prescaler value,
T_{osc} is the clock oscillator period (in seconds).

```
/*************************************************************************
                              GENERATE SQUARE WAVE
                              ====================

In this project a square wave signal with frequency 20 kHz (period 50 us) is generated. The
program uses the built-in PWM module to generate accurate square wave signal.

The output signal is available on pin RC2 (CCP1) of the microcontroller.

Author:    Dogan Ibrahim
Date:      September 2013
File:      MIKROC-WAVE7.C
*************************************************************************/

void main()
{

  ANSELC = 0;                          // Configure PORT C as digital
  TRISC = 0;                           // PORT C as output

  T2CON = 0b00000101;                  // Timer 2 with prescaler 4
  PR2 = 24;                            // Load PR2 register of Timer 2
  CCPTMRS0 = 0;                        // Enable PWM
  CCPR1L = 0X0C;                       // Load duty cycle
  CCP1CON = 0x2C;                      // Load duty cycle and enable PWM

  for(;;)                              // Wait here forever
  {
  }

}
```

Figure 6.43: mikroC Pro for PIC Program Listing.

Substituting the values into the equation, and assuming a prescaler of 4, an oscillator frequency of 8 MHz ($T_{osc} = 0.125$ μs), the PWM period of 50 μs, and duty cycle (ON time) of 25 μs, we get

$$PR2 = \frac{50 \times 10^{-6}}{4 \times 4 \times 0.125 \times 10^{-6}} - 1$$

Which gives PR2 = 24.

The 10-bit value to be loaded into PWM registers is given by (Chapter 2):

$$CCPR1L : CCP1CON\langle 5 : 4\rangle = \frac{\text{PWM duty cycle}}{TMR2PS * T_{osc}}$$

Where the upper 8 bits will be loaded into register CCPR1L, and the two LSB bits will be loaded into bits 4 and 5 of CCP1CON.

or

$$CCPR1L : CCP1CON\langle 5 : 4\rangle = \frac{25 \times 10^{-6}}{4 \times 0.125 \times 10^{-6}} = 50$$

This number in 12-bit binary is "00001100 10". Therefore, the value to be loaded into bits 4 and 5 of CCP1CON are the two LSB bits, that is, "10". Bits 2 and 3 of CCP1CON must be set to HIGH for PWM operation, and bits 6 and 7 are not used. Therefore, CCP1CON must be set to ("X" is do not care):

XX101100 i.e. hexadecimal 0x2C

The number to be loaded into CCPR1L is the upper 8 bits, that is, "00001100", that is, hexadecimal 0x0C.

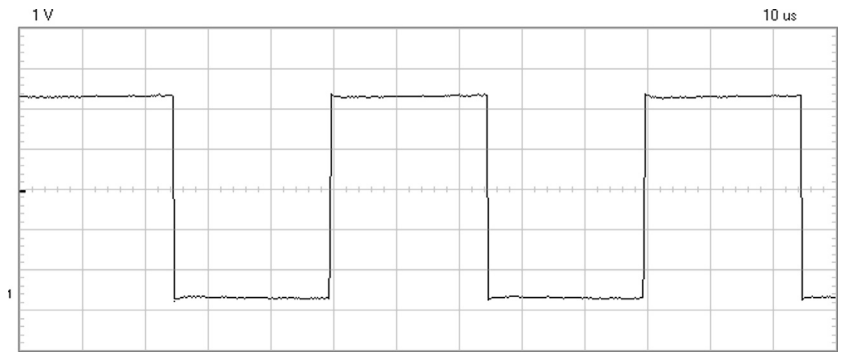


Figure 6.44: Generated Waveform.

At the beginning of the program, PORTC is configured as a digital output port. Then, Timer2 is configured and timer register PR2 is loaded to give the required period. PWM registers CCPR1L and CCP1CON are loaded with the duty cycle (ON time) and the PWM module is enabled. The main program then waits forever where the PWM works in the background to generate the required waveform.

Figure 6.44 shows the generated waveform, which has a period of exactly 50 μs. In this graph, the vertical axis is 1 V per division, and the horizontal axis is 10 μs per division.

The MPLAB XC8 version of the program is shown in Figure 6.45 (XC8-WAVE7.C).

```
/*********************************************************************************
                                GENERATE SQUARE WAVE
                                ====================

In this project a square wave signal with frequency 20 kHz (period 50 us) is generated. The
program uses the built-in PWM module to generate accurate square wave signal.

The output signal is available on pin RC2 (CCP1) of the microcontroller.

Author:     Dogan Ibrahim
Date:       September 2013
File:       XC8-WAVE7.C
*********************************************************************************/
#include <xc.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

void main()
{

  ANSELC = 0;                                  // Configure PORT C as digital
  TRISC = 0;                                   // PORT C as output

  T2CON = 0b00000101;                          // Timer 2 with prescaler 4
  PR2 = 24;                                    // Load PR2 register of Timer 2
  CCPTMRS0 = 0;                                // Enable PWM
  CCPR1L = 0X0C;                               // Load duty cycle
  CCP1CON = 0x2C;                              // Load duty cycle and enable PWM

  for(;;)                                      // Wait here forever
  {
  }
}
```

Figure 6.45: MPLAB XC8 Program Listing.

Project 6.5—Ultrasonic Human Height Measurement

Project Description

This project is about designing a microcontroller-based device to measure the human height using ultrasonic techniques. Having the correct height is very important especially during the child development ages. Human height is usually measured using a stadiometer. A stadiometer can either be mechanical or electronic. Most stadiometers are portable, although they can also be wall mounted. A mechanical stadiometer is commonly used in schools, clinics, hospitals, and doctors' offices.

As shown in Figure 6.46, a mechanical stadiometer consists of a long ruler, preferably mounted vertically on a wall, with a movable horizontal piece that rests on the head of the person whose height is being measured. The height is then read on the ruler corresponding to the point of the horizontal piece. By using such devices, one can measure the height from about 14 to 200 cm with graduations of 0.1 cm.

In this project, we will see how to design an electronic stadiometer based on the principle of ultrasonic waves. Figure 6.47 shows the block diagram of the measurement system. Basically, a pair of ultrasonic transducers (a transmitter TX, and a receiver RX) are mounted at the top of a pole whose height from the ground level is known, say H. The person whose height is to be measured stands under the ultrasonic transducers. The system sends an ultrasonic signal through the TX. This signal hits the person's head and is received by the RX. By knowing the speed of sound in the air and the time the signal takes to return, we can calculate the distance from the transducers to the head of the person. If this distance is called h, then the height of the person is simply given by the difference H – h.

Figure 6.46: A Mechanical Stadiometer.

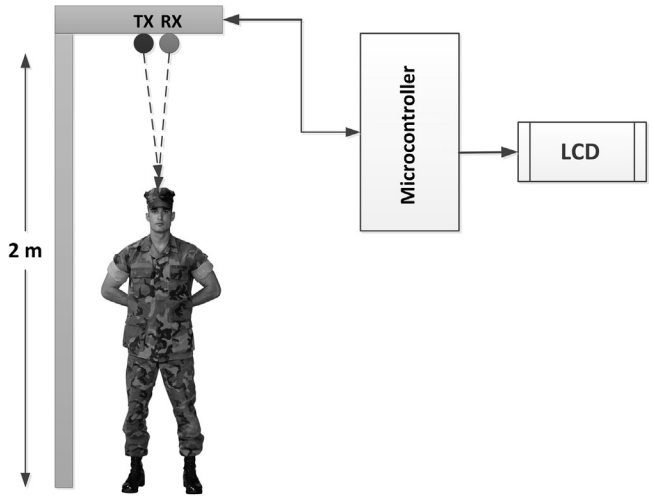


Figure 6.47: Block Diagram of the Height Measurement System.

In this project, the PING))) ultrasonic transducer pair (Figure 6.48), manufactured by Parallax is used. This transducer pair is mainly developed for distance measurement.

Project Hardware

The circuit diagram of the project is shown in Figure 6.49. The PING))) can be used to measure distances from 2 cm to 3 m. The specifications of this device are as follows:

- A 5 V supply voltage,
- A 30 mA supply current,
- A 40 kHz operation frequency,
- Small size (22 × 46 × 16 mm),
- Requirement of only one pin for connection to a microcontroller.

Figure 6.48: PING))) Ultrasonic Transducer Pair.

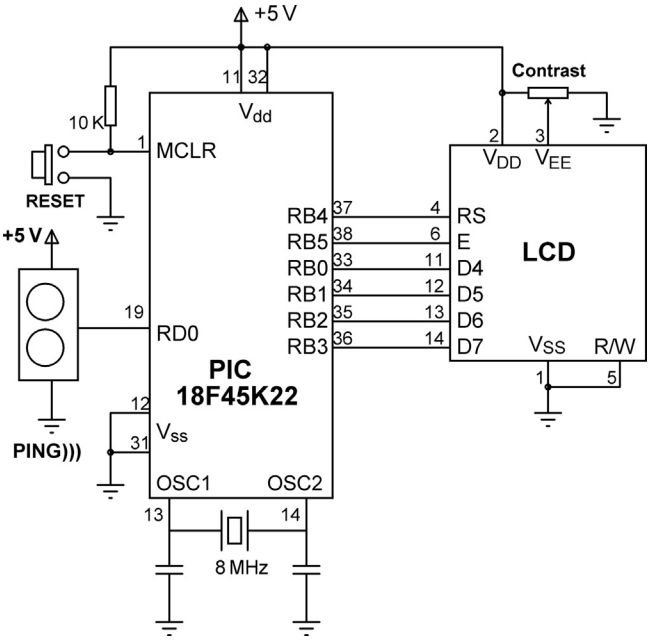


Figure 6.49: Circuit Diagram of the Project.

The device is connected to a microcontroller via the SIG pin, which acts as both an input and an output. In Figure 6.49, this I/O pin is connected to pin RD0 of a PIC18F45K22 microcontroller, operated from an 8 MHz crystal. The pin layout of the PING))) is shown in Figure 6.50. The operation of PING))) is as follows:

The device operates by emitting a short ultrasonic burst at 40 kHz from the TX output and then listening for the echo. This pulse travels in the air at the speed of sound, hits an

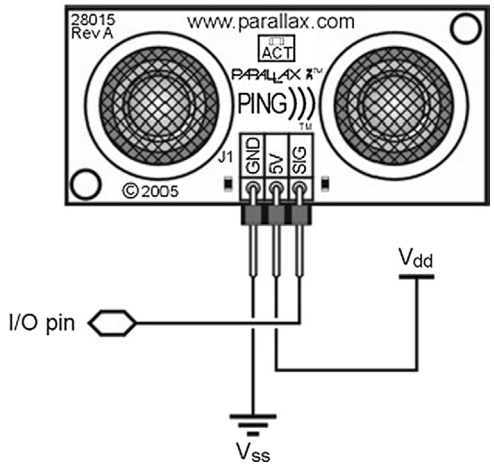


Figure 6.50: Pin Layout of PING))).

```
BEGIN
    Define the LCD connections
    Define PING))) connection
    Configure PORTD as digital
    Configure PORTB as digital
    Initialize LCD
    Send start-up message to the LCD
    Configure Timer0 as 16-bit counter with count time of 1 µs
    DO FOREVER
        Send a pulse to the ultrasonic module
        Start timer
        Wait until echo pulse is received
        Calculate the elapsed time
        Divide the time by 2 to find the time to the object
        Calculate the distance to the object (h)
        Calculate the height of the person (H–h)
        Display the height of the person on the LCD
        Wait 1 s
    ENDDO
END
```

Figure 6.51: PDL of the Project.

object, and then bounces back and is received by the RX sensor. The PING))) provides an output pulse to the microcontroller that will terminate when the echo is detected, and thus, the width of this pulse is proportional to the distance to the target.

An LCD, connected to PORTB, is used to display the height of the person.

Project PDL

The PDL of the project is shown in Figure 6.51. Timer0 of the microcontroller is configured to operate as a counter in the 16-bit mode, and the count time is set to 1 µs using a prescaler value of 2, where with a clock period of 0.125 µs, the count time is given by

$$\text{Count time} = 4 \times 0.125 \times \text{Prescaler} = 4 \times 0.125 \times 2 = 1\ \mu\text{s}.$$

For a prescaler of 2, the lower 2 bits of register T0CON must be loaded with "00".

Assuming that the total measured time is Tm, then the distance h to the object will be (assuming that the speed of sound in air is 340 m/s, or 34 cm/ms, or 0.034 cm/µs):

$$T = Tm/2 \qquad \text{(time T to the object in microseconds)}$$
$$h = 0.034 * T \qquad \text{(Distance} = \text{Speed} \times \text{Time, where h is in cm)}$$

where T is the time it takes for the signal to echo back after hitting the object. Tm is the total time in microseconds measured by Timer0. The distance h in the above equation is in centimeters, and the time T is in microseconds.

The above operation requires floating point arithmetic. Instead, we can use long integer arithmetic to calculate the height with a good accuracy as follows:

$$h = 34 * T/1000$$

where T is a long integer.

Assuming that the PING))) device is mounted on a pole H centimeters high, the height of the person is simply given by

$$\text{Height of person} = H - h$$

The height is displayed on the LCD. After 1 s, the above process is repeated.

Project Program

mikroC Pro for PIC

The mikroC Pro for the PIC program is called MIKROC-HEIGHT.C and is shown in Figure 6.52. At the beginning of the main program, the LCD and ultrasonic module connections are defined, and PORTB and PORTD are configured as digital I/O ports. Then, the LCD is initialized, and the heading "HEIGHT" is displayed on the LCD for 2 s. Timer register T0CON is configured so that TIMER0 operates in a 16-bit counter mode with the count rate of 1 μs. Timer0 prescaler is set to 2.

The height calculation is carried out inside an endless loop formed using a *for* loop. Inside this loop, the counter is cleared, a pulse is sent to the ultrasonic module, counter is started, and the program waits until the echo signal is received. When the echo signal is received, the counter is stopped, and the total elapsed time is read and stored in variable Tm. Then, the height of the person is calculated and stored in variable Person_Height. This number is converted into a string in variable Txt and is displayed on the LCD. The program repeats after a 1 s delay. If, for example, the person's height is 150 cm, it is displayed in the following format:

```
Height (cm)
  150
```

The program given in Figure 6.52 can be improved by the following modifications:

- The height calculation can be done using floating point arithmetic to get more accurate results.
- The program assumes that the speed of sound in the air is fixed and is 340 m/s. In reality, the speed of sound in the air depends on several factors such as the ambient temperature and to a lesser extent the atmospheric pressure and relative humidity. A temperature sensor can be added to the project and more accurate results for the speed can be obtained.

```
/*************************************************************************
                ULTRASONIC HUMAN HEIGHT MEASUREMENT
                ===================================

In this project the height of a person is found and displayed on a LCD. The project uses the
PING))) Ultrasonic distance measuring transducer pair, connected to RD0 pin of a PIC18F45K22
microcontroller, operated from an 8 MHz crystal.

The LCD is connected to PORTB of the microcontroller.

The program assumes that the transducer pair is mounted on a pole 200 cm above the ground
level where the person stands.

Author:    Dogan Ibrahim
Date:      September 2013
File:      MIKROC-HEIGHT.C
*************************************************************************/

// LCD module connections
sbit LCD_RS at LATB4_bit;
sbit LCD_EN at LATB5_bit;
sbit LCD_D4 at LATB0_bit;
sbit LCD_D5 at LATB1_bit;
sbit LCD_D6 at LATB2_bit;
sbit LCD_D7 at LATB3_bit;
sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

// Ultrasonic module connection
sbit Ultrasonic at RD0_bit;
sbit Ultrasonic_Direction at TRISD0_bit;
// End of Ultrasonic module connections

#define Pole_Height 200

void main()
{
   unsigned long Tm;
   unsigned char Tl, Th;
   unsigned int h, Person_Height;
   char Txt[7];

   ANSELB = 0;                          // Configure PORTB as digital
   ANSELD = 0;                          // Configure PORTD as digital

   Lcd_Init();                          // Initialize LCD
```

Figure 6.52: mikroC Pro for PIC Program Listing.

```
    Lcd_Cmd(_LCD_CURSOR_OFF);              // Disable cursor
    Lcd_Out(1,1,"HEIGHT");                 // Display heading
    Delay_Ms(2000);                        // Wait 2 s
//
// Configure Timer 0 as a counter to operate in 16-bit mode with 1 ms
// count time. The prescaler is set to 2. The timer is stopped at this point.
//
    T0CON = 0x00;
//
// Start of program loop
// Send a pulse, start timer, get echo, stop timer, calculate distance and display
//
    for(;;)
    {
        Ultrasonic_Direction = 0;          // RD0 in output mode
        TMR0H = 0;                         // Clear high byte of timer
        TMR0L = 0;                         // Clear low byte of timer

        Ultrasonic = 0;
        Delay_us(3);
        Ultrasonic = 1;                    // Send a PULSE to Ultrasonic module
        Delay_us(5);
        Ultrasonic = 0;
        Ultrasonic_Direction = 1;          // RD0 in input mode
        while(Ultrasonic == 0);            // Wait until echo is received
        T0CON.TMR0ON = 1;                  // Start Timer0
        while(Ultrasonic == 1);
        T0CON.TMR0ON = 0;                  // Stop Timer0
        Tl = TMR0L;                        // Read timer low byte
        Th = TMR0H;                        // Read timer high byte
        Tm = Th * 256 + Tl;                // Timer as 16 bit value
        //
        // Now find the distance to person's head
        Tm = Tm/2;                         // Tm is half the time
        Tm = 34 * Tm;
        Tm = Tm/1000;                      // Divide by 1000
        h = (unsigned int)Tm;              // h is the distance to person's head
        Person_Height= Pole_Height – h;    // Person's height
        //
        // Now display the height
        //
        IntToStr(Person_Height, Txt);      // Convert into string to display
        Lcd_Cmd(_LCD_CLEAR);               // Clear LCD
        Lcd_Out(1,1, "Height (cm)");       // Display heading
        Lcd_Out(2,1, Txt);                 // Display the height
        Delay_Ms(1000);                    // Wait 1 s
    }
}
```

Figure 6.52
cont'd

```
/*************************************************************************
                        ULTRASONIC HUMAN HEIGHT MEASUREMENT
                        ===================================

In this project the height of a person is found and displayed on a LCD. The project uses the
PING))) Ultrasonic distance measuring transducer pair, connected to RD0 pin of a PIC18F45K22
microcontroller, operated from an 8 MHz crystal.

The LCD is connected to the microcontroller as follows:

Microcontroller         LCD
===============         ===

   RB0                  D4
   RB1                  D5
   RB2                  D6
   RB3                  D7
   RB4                  E
   RB5                  R/S
   RB6                  RW

The program assumes that the transducer pair is mounted on a pole 200 cm above the ground
level where the person stands.

Author:     Dogan Ibrahim
Date:       September 2013
File:       XC8-HEIGHT.C
*************************************************************************/

#include <xc.h>
#include <plib/xlcd.h>
#include <stdlib.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

// Ultrasonic module connection
#define Ultrasonic PORTDbits. RD0
#define Ultrasonic_Direction TRISDbits.TRISD0
// End of Ultrasonic module connections

#define Pole_Height 200

//
// This function creates seconds delay. The argument specifies the delay time in seconds.
//
void Delay_Seconds(unsigned char s)
{
   unsigned char i,j;

   for(j = 0; j < s; j++)
   {
```

Figure 6.53: MPLAB XC8 Program Listing.

```
      for(i = 0; i <  100; i++)__delay_ms(10);
   }
}

//
// This function creates 18 cycles delay for the xlcd library
//
void DelayFor18TCY( void )
{
Nop(); Nop(); Nop(); Nop();
Nop(); Nop(); Nop(); Nop();
Nop(); Nop(); Nop(); Nop();
Nop(); Nop();
return;
}


//
// This function creates 15 ms delay for the xlcd library
//
void DelayPORXLCD( void )
{
   __delay_ms(15);
   return;
}


//
// This function creates 5 ms delay for the xlcd library
//
void DelayXLCD( void )
{
   __delay_ms(5);
   return;
}

//
// This function clears the screen
//
void LCD_Clear()
{
   while(BusyXLCD());
   WriteCmdXLCD(0x01);
}

//
// This function moves the cursor to position row,column
//
void LCD_Move(unsigned char row, unsigned char column)
{
 char ddaddr = 40 * (row–1) + column;
```

Figure 6.53
cont'd

```
  while( BusyXLCD() );
  SetDDRamAddr( ddaddr );
}



void main()
{
    unsigned long Tm;
    unsigned char Tl, Th;
    unsigned int h, Person_Height;
    char Txt[10];

    ANSELB = 0;                                    // Configure PORTB as digital
    ANSELD = 0;                                    // Configure PORTD as digital

    OpenXLCD(FOUR_BIT & LINES_5X7);                // Initialize LCD

    while(BusyXLCD());                             // Wait if the LCD is busy
    WriteCmdXLCD(DON);                             // Turn Display ON
    while(BusyXLCD());                             // Wait if the LCD is busy
    WriteCmdXLCD(0x06);                            // Move cursor right
    putrsXLCD("HEIGHT");                           // Display heading
    Delay_Seconds(2);                              // 2 s delay
    LCD_Clear();                                   // Clear display
//
// Configure Timer 0 as a counter to operate in 16-bit mode with 1 ms
// count time. The prescaler is set to 2. The timer is stopped at this point.
//
    T0CON = 0x00;
//
// Start of program loop
// Send a pulse, start timer, get echo, stop timer, calculate distance and display
//
    for(;;)
    {
        Ultrasonic_Direction = 0;                  // RD0 in output mode
        TMR0H = 0;                                 // Clear high byte of timer
        TMR0L = 0;                                 // Clear low byte of timer

        Ultrasonic = 0;
        __delay_us(3);
        Ultrasonic = 1;                            // Send a PULSE to Ultrasonic module
        __delay_us(5);
        Ultrasonic = 0;
        Ultrasonic_Direction = 1;                  // RD0 in input mode
        while(Ultrasonic == 0);                    // Wait until echo is received
        T0CONbits.TMR0ON = 1;                      // Start Timer0
        while(Ultrasonic == 1);
        T0CONbits.TMR0ON = 0;                      // Stop Timer0
        Tl = TMR0L;                                // Read timer low byte
```

Figure 6.53
cont'd

```
        Th = TMR0H;                          // Read timer high byte
        Tm = Th * 256 + Tl;                  // Timer as 16 bit value
        //
        // Now find the distance to person's head
        Tm = Tm/2;                           // Tm is half the time
        Tm = 34 * Tm;
        Tm = Tm/1000;                        // Divide by 1000
        h = (unsigned int)Tm;                // h is the distance to person's head
        Person_Height= Pole_Height – h;      // Person's height
        //
        // Now display the height
        //
        itoa(Txt, Person_Height, 10);        // Convert into string to display
        LCD_Clear();                         // Clear LCD
        LCD_Move(1,1);
        putrsXLCD("Height (cm)");            // Display heading
        LCD_Move(2,1);
        putrsXLCD(Txt);                      // Display the height
        Delay_Seconds(1);                    // Wait 1 s
    }
}
```

Figure 6.53
cont'd

The temperature dependency of the speed of sound in dry air is given by

$$\text{Speed} = 331.4 + 0.6T_C,$$

Where the speed is in meters per second and T_C is the ambient temperature in degrees centigrade. At 15 °C, the speed becomes nearly 340 m/s.

MPLAB XC8

The MPLAB XC8 program is called XC8-HEIGHT.C and is shown in Figure 6.53. Note that although the LCD is connected to PORTB of the microcontroller, the following pin connections differ from Figure 6.49:

- RB4 is connected to pin E of the LCD.
- RB5 is connected to pin RS of the LCD.
- RB6 is connected to pin RW of the LCD.

Project 6.6—Minielectronic Organ

Project Description

This project is about designing a microcontroller-based minielectronic organ using a 4×4 keypad with 16 keys to produce musical notes in one octave. Figure 6.54 shows the block diagram of the project.

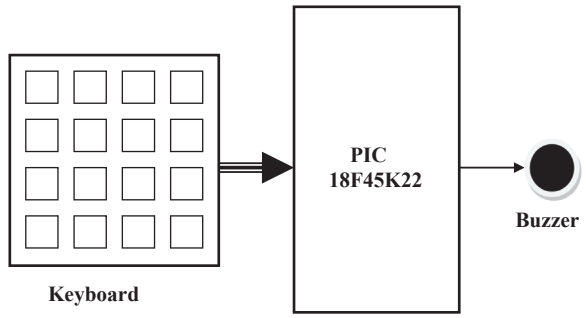


Figure 6.54: Block Diagram of the Project.

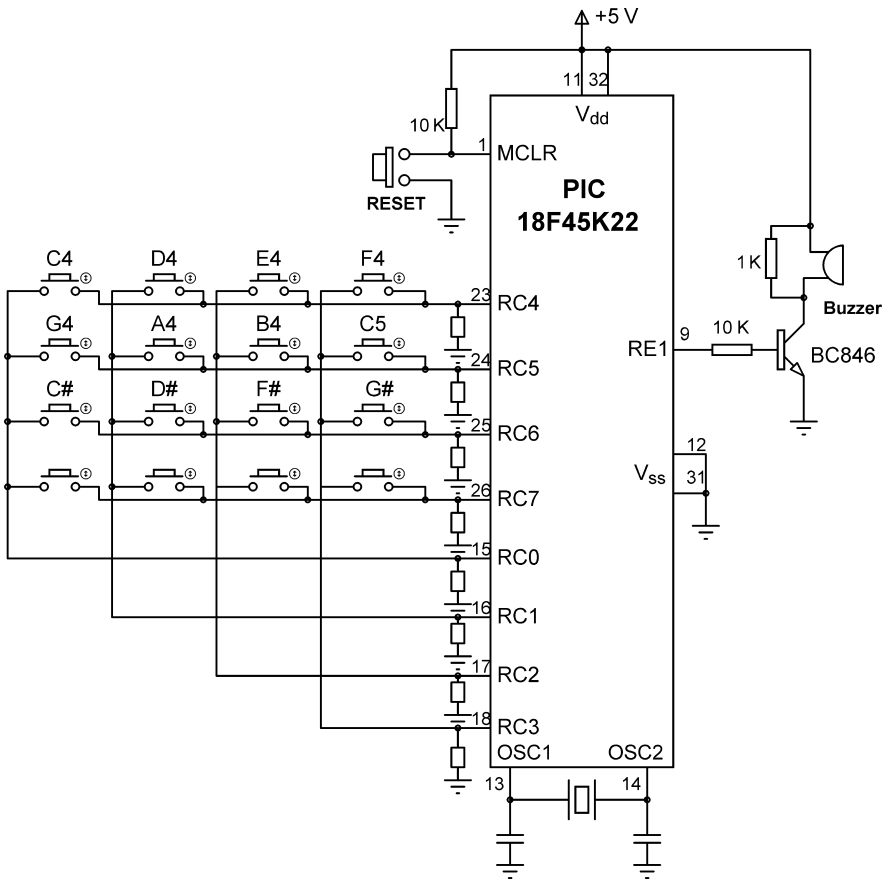


Figure 6.55: Circuit Diagram of the Project.

```
BEGIN
        Store musical notes in a table
        Configure PORTE as digital
        Configure PORTC as digital
        Initialize Keypad library
        Initialize Sound library
        DO FOREVER
                Get the code of the key pressed
                IF this is a valid key code (< 13)
                        Play the note corresponding to this note
                ENDIF
        ENDDO
END
```

Figure 6.56: PDL of the Project.

The musical notes are configured on the keypad as follows:

```
C4    D4    E4    F4
G4    A4    B4    C5
C4#   D4#   F4#   G4#
A4#
```

The frequencies of the notes are

Notes	C4	C4#	D4	D4#	E4	F4	F4#	G4	G4#	A4	A4#	B4
Hz	261.63	277.18	293.66	311.13	329.63	349.23	370	392	415.3	440	466.16	493.88

Project Hardware

The circuit diagram of the project is shown in Figure 6.55. The keypad is connected to PORTC of the microcontroller. A buzzer is connected to port pin RE1.

Project PDL

The project PDL is shown in Figure 6.56.

Project Program

mikroC Pro for PIC

The mikroC Pro for the PIC program listing is given in Figure 6.57 (MIKROC-MUSIC.C). The program is very simple. The notes are stored in an array called Notes. The program initializes the keypad library and the sound library. Then, an endless loop is formed, and

```
/*************************************************************************
                        MINI ELECTRONIC ORGAN
                        =====================

In this project a 4 x 4 keypad is connected to PORTC of a PIC18F45K22 microcontroller. Also a
buzzer is connected to port pin RE1. The keypad is organized such that pressing a key plays a
musical note. The notes on the keypad are organized as follows:

C4    D4    E4    F4
G4    A4    B4    C5
C4#   D4#   F4#   G4#
A4#

Author:   Dogan Ibrahim
Date:     September 2013
File:     MIKROC–MUSIC.C
*************************************************************************/
// Keypad module connections
char keypadPort at PORTC;
// End of keypad module connections

//
// Start of MAIN program
//
void main()
{
   unsigned char MyKey;
   unsigned Notes[] = {0,262,294,330,349,392,440,494,524,277,311,370,415,466};

   ANSELE = 0;                                    // Configure PORTE as digital
   ANSELC = 0;                                    // Configure PORTC as digital
   TRISC = 0xF0;                                  // RC4–RC7 are inputs

   Keypad_Init();                                 // Initialize keypad library
   Sound_Init(&PORTE, 1);                         // Initialize sound library
//
// Program loop
//
   for(;;)                                        // Endless loop
   {
      do
         MyKey = Keypad_Key_Press();              // Get code of pressed key
      while(!MyKey);

      if(MyKey <= 13)Sound_Play(Notes[MyKey], 100);  // Play the note
   }
}
```

Figure 6.57: mikroC Pro for PIC Program Listing.

inside this loop, the code of the pressed key is determined, and this is used as an index to the Notes array to play the note corresponding to the pressed key. Notes are played for a minimum of 100 ms when a key is pressed. Valid key codes are from 1 to 13, and keys with codes >13 are not played.

Project 6.7—Frequency Counter with an LCD Display

Project Description

In this project, we look at the design of a simple frequency counter with an LCD display.

There are basically two methods used for the measurement of the frequency of an external signal.

Method I

This is perhaps the easiest method. The signal whose frequency is to be measured is connected to the clock input of a microcontroller counter (timer). The counter is enabled for a known time window, and the total count is read. Since each count corresponds to a clock pulse of the external signal, we can easily determine its frequency. For example, assuming that the time window is set to 1 s and the counter reaches 1000 at the end of 1 s, then the frequency of the signal is 1 kHz. Thus, we can simply display the counter value as the frequency of the signal in hertz. This way, using a 16-bit counter, we can measure frequencies up to 65,535 Hz. If using a 32-bit counter, the maximum frequency that we can measure will be 4,294,967,295 Hz. For the measurement of high frequencies, we can actually use a 16-bit counter and increment a variable each time the counter overflows (65535−0). The total count can then be found by multiplying this variable by 65536 and adding the current counter reading.

The nice thing about using a 1 s time window is that we can directly display the frequency in hertz by simply reading the counter value. This also means that the measurement resolution is 1 Hz, which is acceptable for most measurements. Increasing the time window to 10 s will increase the resolution to 0.1 Hz. On the other hand, decreasing the time window to 0.1 s will reduce the resolution to 10 Hz. Figure 6.58 illustrates this method.

Method II

In this method, the time between two or more successive edges of the signal is measured. The period and the frequency are then calculated easily. The counter (timer) is started as

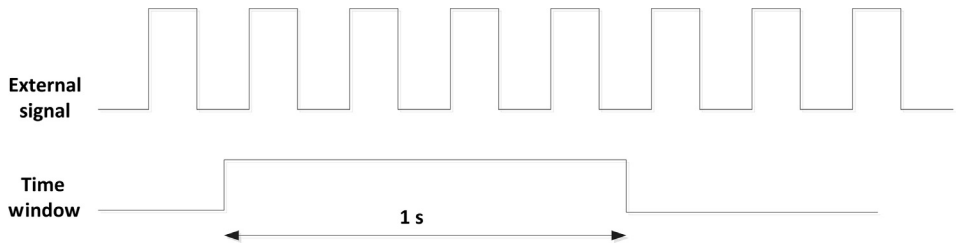


Figure 6.58: Frequency Measurement—Method I.

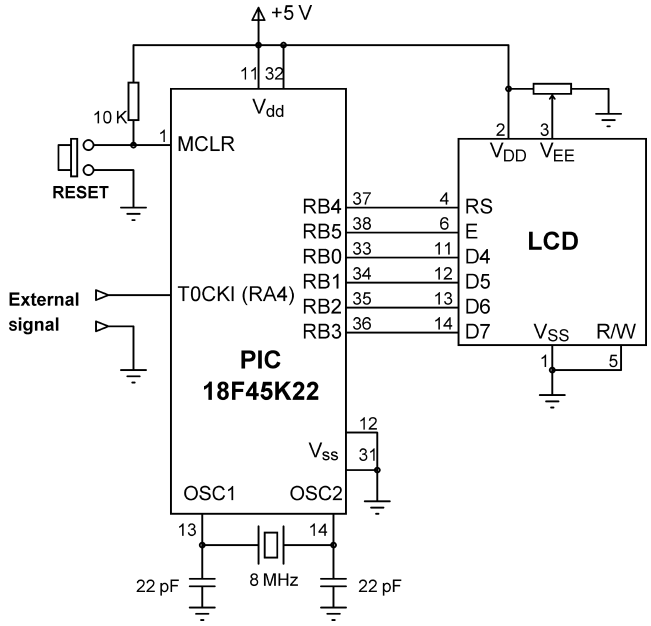


Figure 6.59: Circuit Diagram of the Project.

soon as the leading edge of the signal is detected. The counter is stopped when the next leading edge is detected. The elapsed time and hence the period of the signal and the frequency are then easily calculated. For example, assuming that the counter is configured to count using the internal clock at the rate of 1 μs, if the elapsed time between two edges is 1000 μs, then the period of the signal is 1 ms (1 kHz). The accuracy of the measurement depends on the accuracy of the internal clock and the frequency stability of the external signal. To improve the accuracy, sometimes more than two signal edges are taken.

In this project, the use of Method I will be illustrated by designing a frequency counter system.

Using Method I

Figure 6.59 shows the circuit diagram of the project. In this design, Timer0 is used in a 16-bit counter mode. The external signal whose frequency is to be measured is applied to Timer0 clock input T0CKI (RA4). An LCD is connected to PORTB of the microcontroller as in the previous LCD projects.

Timer1 is used to create the 1 s time window. It is not possible to generate a 1 s delay using Timer1 since the microcontroller clock frequency is high. Instead, the timer is configured to generate an interrupt every 250 ms, and when 4 interrupts are generated, it is assumed that 1 s has elapsed.

```
BEGIN
    Configure PORTA and PORTB as digital
    Configure RA4 (T0CKI) as input
    Initialize LCD
    DO FOREVER
        Clear Timer0 registers
        Load Timer1 registers for 250 ms interrupt
        Clear Overflow and Cnt
        Enable Timer0 and Timer1
        Wait until 1 s elapsed (Cnt = 4)
        Stop Timer0 and Timer1
        Calculate Timer0 count
        Convert count into string
        Clear Display
        Display heading "Frequency (Hz)"
        Display the frequency
        Wait 1 s
    ENDDO
END
```

Figure 6.60: PDL of the Project.

Assuming a prescaler setting of 8, the value to be loaded into Timer1 registers to generate interrupts at 250 ms (250,000 μs) intervals can be calculated from the following:

$$\text{TMR0L} = 65536 - \frac{\text{Time}}{4 * \text{clockperiod} * \text{prescaler}}$$

or

$$\text{TMR0L} = 65536 - \frac{250000}{4 * 0.125 * 8} = 3036$$

Decimal 3036 is equivalent to 0x0BDC in hexadecimals. Thus, TMR0H = 0x0B and TMR0L = 0xDC.

Project PDL

The project PDL is shown in Figure 6.60.

Project Program

mikroC Pro for PIC

The mikroC Pro for PIC program listing is given in Figure 6.61 (MIKROC-FREQ1.C). At the beginning of the program, PORTA and PORTB are configured as digital. RA4 is configured as an input, the LCD is initialized, and the cursor is disabled.

```
/*************************************************************************
                                Frequency Counter
                                ================

This project is a frequency counter. The signal whose frequency is to be measured is applied
to pin RA4 (T0CKI) of a PIC18F45K22 microcontroller, operating from an 8 MHz crystal. The
project measures the frequency and displays on an  LCD in Hz.

The resolution of the measurement is 1 Hz. The project can measure frequencies from 1 Hz to
several MHz.

The project uses 2 timers, TIMER0 and TIMER1. TIMER1 is used to open a time window of 1 s
width. The pulses of the external signal increment the counter during this 1 s
window. At the end of 1 s both timers are stopped. The count in TIMER0 gives us
directly the frequency in Hz.

Both timers generate interrupts for higher accuracy. TIMER0 uses a variable called "Overflow"
to find out how many times it has overflowed (if any). This variable is used in calculating the
 total count. TIMER1 interrupts at every 250 ms and a variable called Cnt is incremented at
each interrupt. When Cnt = 4 then it is assumed that 1 s has elapsed.

Programmer:    Dogan Ibrahim
Date:          September 2013
File:          MIKROC-FREQ1.C

*************************************************************************/
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

#define PULSE PORTA.RA4

unsigned int Overflow;
unsigned char Cnt;
//
// Timer interrupt service routineProgram jumps here at every 10 ms
//
void interrupt (void)
{
```

Figure 6.61: mikroC Pro for PIC Program Listing.

```
    if(INTCON.TMR0IF == 1)                    // If TIMER0 interrupt
    {
      Overflow++;                             // Increment Overflow count
      INTCON.TMR0IF = 0;                      // Clear Timer0 interrupt flag
    }
    if(PIR1.TMR1IF == 1)                      // If TIMER1 interrupt
    {
     TMR1H = 0x0B;                            // Reload timer register
     TMR1L = 0xDC;
     Cnt++;                                   // Increment Cnt
     PIR1.TMR1IF = 0;                         // Clear Timer1 interrupt flag
    }
}


void main()
{
    unsigned char Txt[11];
    unsigned long Elapsed;
    unsigned char L_Byte, H_Byte;

    ANSELA = 0;                               // Configure PORTA as digital
    ANSELB = 0;
    TRISA.RA4 = 1;                            // RA4 is input
    Lcd_Init();                               // Initialize LCD
    Lcd_Cmd(_LCD_CURSOR_OFF);                 //  Disable cursor

//
 // Configure TIMER0 for 16-bit mode, no prescaler, clock provided by external
 // signal into pin T0CKI. Timer is not started here
 //
    T0CON = 0x28;                             // TIMER0 16-bit,no prescaler,T0CKI clk
//
// Configure Timer1 for 250 ms Overflow. Timer1 is not started here
//
    T1CON = 0x36;

    PIE1 = 0x01;                              // Enable TIMER1 interrupts
    PIR1.TMR1IF = 0;                          // Clear TIMER1 interrupt flag
    INTCON = 0xE0;                            // Enable TIMER0 and TIMER1 interrupts

    for(;;)                                   // Wait for interrupts
    {
     TMR0H = 0;                               // Clear Timer0 registers
     TMR0L = 0;
     TMR1H = 0x0B;                            // Load Timer1 registers
     TMR1L = 0xDC;
     //
     Overflow = 0;                            // Clear Overflow count
     Cnt = 0;                                 // Clear Cnt
     //
```

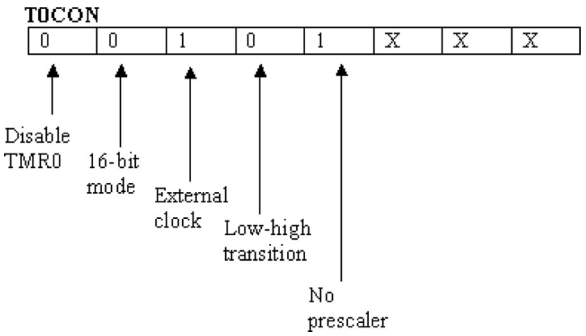
Figure 6.61
cont'd

```
    // Start TIMER0. It will increment each time an external pulse is detected.
    // TIMER0 increments on the rising edge of the external clock
    //
    T0CON.TMR0ON = 1;
    //
    // Start Timer1 to count 4 × 250 ms = 1 s
    //
    T1CON.TMR1ON = 1;
    while(Cnt != 4);                          // Wait until 1 s has elapsed
    //
    // 1 s has elapsed. Stop both timers
    //
    T0CON.TMR0ON = 0;                         // Stop TIMER0
    T1CON.TMR1ON = 0;                         // Stop Timer1
    // Get TIMER0 count
    L_Byte = TMR0L;
    H_Byte = TMR0H;
    //
    // Store TIMER0 count is variable Elapsed
    //
    Elapsed = (unsigned long)256 * H_Byte + L_Byte;
    Elapsed = 65536 * Overflow + Elapsed;
    //
    // Convert into string and display
    //
    LongWordToStr(Elapsed, Txt);              // Long to string conversion
    Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD
    Lcd_Out(1,1,"Frequency (HZ)");            // Display heading
    Lcd_Out(2,1,Txt);                         // Display measured frequency

    Delay_ms(1000);                           // Wait 1 s and repeat
  }
}
```

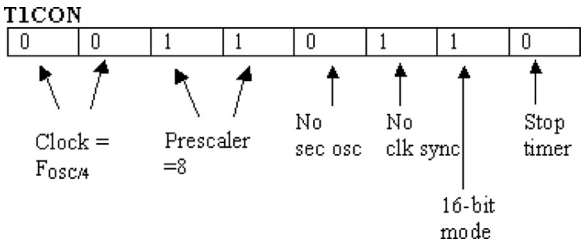
Figure 5.61
cont'd

The program then configures Timer0 as a counter. Timer0 control register T0CON is loaded as follows:



Thus, the T0CON register should be loaded with hexadecimal 0x28.

Similarly, Timer1 control register T1CON is loaded as follows:



Thus, the T1CON register should be loaded with hexadecimal 0x36.

Then, Timer1 interrupts are enabled by setting PIE1 = 1 and also the timer1 interrupt flag is cleared by clearing PIR1.TMR1IF = 0. The next step is to configure interrupt register INTCON to enable Timer0 interrupts, unmasked interrupts (e.g. Timer1), and global interrupts. This is done by setting INTCON to hexadecimal 0xE0. The remainder of the program is executed in an endless loop. The following operations are carried out inside this loop:

- Timer0 registers cleared.
- Timer1 registers loaded for 250 ms interrupt.
- Variables Overflow and Cnt are cleared.
- Timer0 is enabled so that the counter counts every time external pulse is received.
- Timer1 is enabled so that interrupts are generated at every 250 ms.
- The program then waits until 1 s has elapsed (Cnt = 4).
- At this point, Timer0 and Timer1 are stopped.
- Timer0 high and low count is read.
- Total Timer0 count is calculated.
- Timer0 count is converted into string format and displayed on the LCD.
- Program waits for 1 s and above process is repeated.

If, for example, the frequency is 25 kHz it is displayed as follows:

```
Frequency (Hz)
  25000
```

Project 6.8—Reaction Timer

Project Description

This project tests the reaction time of a person. Basically, a light emitting diode (LED) is turned ON after a random delay of 1−10 s. The project times how long it takes for the person to hit a switch in response. The reaction time is displayed on a LCD in milliseconds.

Figure 6.62 shows the block diagram of the project.

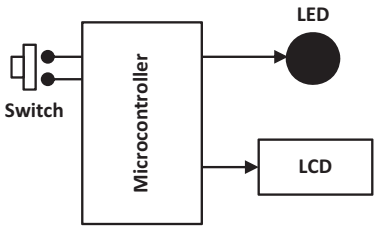


Figure 6.62: Block Diagram of the Project.

Project Hardware

Figure 6.63 shows the circuit diagram. The LED and the push-button switch are connected to port pins RC0 and RC7, respectively. The switch is configured in the active low mode so that when the switch is pressed logic 0 is sent to the microcontroller. The LCD is connected to PORTB as in the previous LCD projects.

Project PDL

The project PDL is shown in Figure 6.64. The program counts and displays the reaction time in milliseconds. Timer0 is used in the 16-bit mode to determine the reaction time. Using a prescaler of 256, Timer0 count rate is given by

$$4 \times 0.125 \times 256 = 128\ \mu s$$

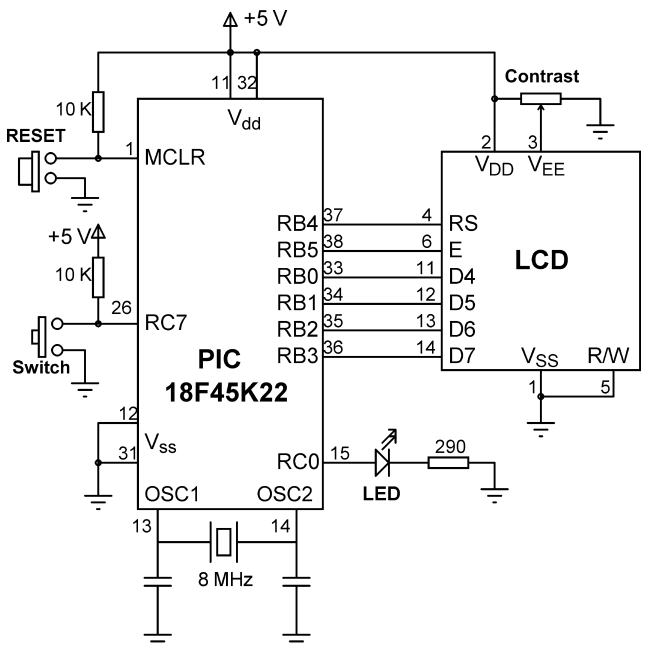


Figure 6.63: Circuit Diagram of the Project.

```
BEGIN
    Define interface between the LCD and microcontroller
    Configure PORTB and PORTC as digital
    Configure RC0 as output and RC7 as input
    Initialize LCD
    Configure Timer0 in 16-bit,prescaler 256
    Turn OFF LED
    Initialize random number seed
    DO FOREVER
        Generate random number 1 to 10
        Clear timer registers
        Turn ON LED
        Turn ON timer
        Wait until switch is pressed
        Stop timer
        Read timer count
        Convert count to milliseconds
        Convert count to string
        Display count on LCD
        Turn OFF LED
        Delay 2 s
        Clear LCD
    ENDDO
END
```

Figure 6.64: Project PDL.

We will start the timer as soon as the LED is turned ON and stop it when the switch is pressed. If the timer count is N, then the reaction time in milliseconds will be

$$\text{Reaction time} = 128 \times \text{N}/1000$$

The maximum 16-bit Timer0 count is 65535. This corresponds to $128 \times 65535 = 8388$ μs. Thus, the maximum reaction time that can be measured is just over 8.3 s (too long for anyone to react to the LED) after which time the timer overflows and timer flag TMR0IF is set to 1. If when the switch is pressed the timer flag is set, then this means that the timer has overflowed and the user is requested to try again.

For operating Timer0 in the 16-bit mode and for a prescaler of 256, the value to be loaded into register T0CON is "0000 0111", or hexadecimal 0x07. Note that the timer is stopped initially.

Project Program

mikroC Pro for PIC

The mikroC Pro for PIC program listing is given in Figure 6.65 (MIKRO-REACT.C). At the beginning of the program, the connection between the LCD and the microcontroller is defined; symbols LED and SWITCH are assigned to port pins RC0 and RC7,

```
/*************************************************************************
                                    Reaction Timer
                                    ==============

This project is a reaction timer. Basically an LED is connected to RC0, a push-button switch
is connected to RC7 and an LCD is connected to PORTB.

The LED turns ON after a random delay (between 1 and 10 s) and the user is expected
to press the button as soon as he/she sees the LED. The elapsed time between the LED  turning
ON and the button being pressed is displayed in milliseconds on the LCD

The project can measure reaction times from 1 ms to just over 8.3 s (In practice it is not
possible for any person to react in more than a few seconds). A message is sent to the LCD if
the maximum reaction time is exceeded.

Programmer:    Dogan Ibrahim
Date:          September 2013
File:          MIKROC-REACT.C

*************************************************************************/
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

#define LED PORTC.RC0
#define SWITCH PORTC.RC7

void main()
{
   unsigned char T_Low, T_High, Txt[11];
   unsigned int No;
   unsigned long Cnt;

   ANSELB = 0;                                  // Configure PORTB as digital
   ANSELC = 0;                                  // Configure PORTC as digital
   TRISC.RC0 = 0;                               // Configure RC0 as output
   TRISC.RC7 = 1;                               // Configure RC7 as input

   Lcd_Init();                                  // Initialize LCD
```

Figure 6.65: mikroC Pro for PIC Program Listing.

```
    Lcd_Cmd(_LCD_CURSOR_OFF);                    //  Disable cursor
    T0CON = 0x07;                                // Timer0, 16-bit, prescaler = 256
    LED = 0;                                     // Turn OFF LED to start with
    srand(10);                                   // Initialize random number seed
    INTCON.TMR0IF = 0;                           // Clear timer overflow flag

    for(;;)
    {
//
// Generate a random number between 1 and 32767 and change it to be between 1 and 10
//
     No = rand();                                // Random number between 1 and 32767
     No = No % 10 + 1;                           // Number between 1 and 10
     Vdelay_Ms(1000 * No);                       // Delay No seconds
     TMR0H = 0;                                  // Clear Timer0 H register
     TMR0L = 0;                                  // Clear Timer0 L register
     LED = 1;                                    // Turn ON LED
//
// Turn ON Timer0 (counts in multiples of 128 microseconds)
//
     T0CON.TMR0ON = 1;                           // Turn ON Timer0
     while(SWITCH == 1);                         // Wait until the switch is pressed
//
// Switch is pressed. Stop timer and get the count
//
     T0CON.TMR0ON = 0;                           // Stop Timer0
     T_Low = TMR0L;                              // Read timer low byte
     T_High = TMR0H;                             // read timer high byte
     Cnt = (unsigned long)256 * T_High + T_Low;  // Get timer count
     Cnt = 128 * Cnt/1000;                       // Convert to milliseconds

     if(INTCON.TMR0IF == 1)                      // If timer overflow detected
     {
      Lcd_Out(1,1,"Too long...");
      Lcd_Out(2,1,"Try again...");
      INTCON.TMR0IF = 0;                         // Clear timer overflow flag
     }
     else
     {
//
// Convert count to string and display on LCD
//
       LongWordToStr(Cnt, Txt);                  // Convert to string
       Lcd_Out(1,1,"Reaction (ms)");             // Display heading
       Lcd_Out(2,1,Txt);                         // Display reaction time in ms
      }
      LED = 0;                                   // Turn OFF LED
      Delay_ms(2000);                            // Wait 2 s and repeat
      Lcd_Cmd(_LCD_CLEAR);                       // Clear LCD
    }
}
```

Figure 6.65
cont'd

```
/*************************************************************************
                                    Reaction Timer
                                    ==============

This project is a reaction timer. Basically an LED is connected to RC0, a push-button switch is
connected to RC7 and an LCD is connected to PORTB.

The LED turns ON after a random delay (between 1 and 10 s) and the user is expected to
press the button as soon as he/she sees the LED. The elapsed time between the LED turning ON
and the button being pressed is displayed in milliseconds on the LCD

The project can measure reaction times from 1 ms to just over 8.3 s (In practice it is not
possible for any person to react in more than a few seconds). A message is sent to the LCD is
the maximum reaction time is exceeded.

Programmer:    Dogan Ibrahim
Date:          September 2013
File:          XC8-REACT.C

*************************************************************************/

#include <xc.h>
#include <plib/xlcd.h>
#include <stdlib.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

#define LED PORTCbits.RC0
#define SWITCH PORTCbits.RC7

//
// This function creates seconds delay. The argument specifies the delay time in seconds.
//
void Delay_Seconds(unsigned char s)
{
  unsigned char i,j;

  for(j = 0; j < s; j++)
  {
    for(i = 0; i <  100; i++)__delay_ms(10);
  }
}

//
// This function creates 18 cycles delay for the xlcd library
//
void DelayFor18TCY( void )
{
Nop(); Nop(); Nop(); Nop();
Nop(); Nop(); Nop(); Nop();
Nop(); Nop(); Nop(); Nop();
```

Figure 6.66: MPLAB XC8 Program Listing.

```
Nop(); Nop();
return;
}


//
// This fucntion creates 15 ms delay for the xlcd library
//
void DelayPORXLCD( void )
{
   __delay_ms(15);
   return;
}


//
// This function creates 5 ms delay for the xlcd library
//
void DelayXLCD( void )
{
   __delay_ms(5);
   return;
}

//
// This function clears the screen
//
void LCD_Clear()
{
   while(BusyXLCD());
   WriteCmdXLCD(0x01);
}

//
// This function moves the cursor to position row,column
//
void LCD_Move(unsigned char row, unsigned char column)
{
   char ddaddr = 40*(row – 1) + column;
   while( BusyXLCD() );
   SetDDRamAddr( ddaddr );
}



void main()
{
    unsigned char T_Low, T_High, Txt[11];
    unsigned int No;
    unsigned long Cnt;
```

Figure 6.66
cont'd

```
    ANSELB = 0;                                  // Configure PORTB as digital
    ANSELC = 0;                                  // Configure PORTC as digital
    TRISCbits.TRISC0 = 0;                        // Configure RC0 as output
    TRISCbits.TRISC7 = 1;                        // Configure RC7 as input

    OpenXLCD(FOUR_BIT & LINES_5X7);              // Initialize LCD

    while(BusyXLCD());                           // Wait if the LCD is busy
    WriteCmdXLCD(DON);                           // Turn Display ON
    while(BusyXLCD());                           // Wait if the LCD is busy
    WriteCmdXLCD(0x06);                          // Move cursor right
    LCD_Clear();                                 // Clear display

    T0CON = 0x07;                                // Timer0 for 16-bit, prescaler = 256
    LED = 0;                                     // Turn OFF LED to start with
    srand(10);                                    // Initialize random number seed
    INTCONbits.TMR0IF = 0;                        // Clear timer overflow flag

    for(;;)
    {
//
// Generate a random number between 1 and 32767 and change it to be between 1 and 10
//
      No = rand();                               // Random number between 1 and 32767
      No = No % 10 + 1;                          // Make the number between 1 and 10
      Delay_Seconds(No);                         // Delay No seconds
      TMR0H = 0;                                 // Clear Timer0 H register
      TMR0L = 0;                                 // Clear Timer0 L register
      LED = 1;                                   // Turn ON LED
//
// Turn ON Timer0 (counts in multiples of 128 microseconds)
//
      T0CONbits.TMR0ON = 1;                      // Turn ON Timer0
      while(SWITCH == 1);                        // Wait until the switch is pressed
//
// Switch is pressed. Stop timer and get the count
//
      T0CONbits.TMR0ON = 0;                      // Stop Timer0
      T_Low = TMR0L;                             // Read timer low byte
      T_High = TMR0H;                            // read timer high byte
      Cnt = (unsigned long)256 * T_High + T_Low; // Get timer count
      Cnt = 128 * Cnt/1000;                      // Convert to milliseconds

      if(INTCONbits.TMR0IF == 1)                  // If timer overflow detected
      {
        LCD_Move(1,1);
        putrsXLCD("Too long...");                // Display message
        LCD_Move(2,1);
        putrsXLCD("Try again...");
        INTCONbits.TMR0IF = 0;                   // Clear timer overflow flag
      }
```

Figure 6.66
cont'd

```
      else
      {
//
// Convert count to string and display on LCD
//
       utoa(Txt, Cnt, 10);                    // Convert to string
       LCD_Move(1,1);
       putrsXLCD("Reaction (ms)");            // Display heading
       LCD_Move(2,1);
       putrsXLCD(Txt);                        // Display reaction time in ms
      }
      LED = 0;                                // Turn OFF LED
      Delay_Seconds(2);                        // Wait 2 s and repeat
      LCD_Clear();                            // Clear LCD
    }
}
```

Figure 6.66
cont'd

respectively. Then, PORTB and PORTC are configured as digital; RC7 is configured as an input pin. The LCD is initialized, Timer0 is configured to operate in the 16-bit mode with a prescaler of 256, and the count is disabled at this point. The timer overflow flag (INT0IF) is cleared, the LED is turned OFF, and the random number seed srand is loaded with an integer number. The remainder of the program is executed inside an endless loop.

Inside the endless loop, the LED is turned ON and Timer0 is turned ON to start counting the reaction time. When the SWITCH is pressed, the timer is stopped, and the count is read and converted into milliseconds. If the timer has overflowed (INT0IF is set), then the measurement is ignored since it is not correct anymore and message "Too long…Try again…" is sent to the LCD. Otherwise, the count is converted into a string and displayed on the LCD. The program repeats after a 2 s delay.

If, for example, the reaction time is 2568 ms, then it will be displayed as follows:

```
Reaction (ms)
  2568
```

The program can be modified if desired to measure the reaction time to sound. This will require the replacement of the LED with a buzzer and the generation of an audible sound with the required frequency.

MPLAB XC8

The MPLAB XC8 version of the program is shown in Figure 6.66 (XC8-REACT.C). The program operates as in the mikroC Pro for PIC version.

Project 6.9—Temperature and Relative Humidity Measurement

Project Description

This project demonstrates how the ambient temperature and relative humidity can be measured and then displayed on an LCD.

In this project, the SHT11 relative humidity and temperature sensor chip is used. This is a tiny eight-pin chip with dimensions 4.93 × 7.47 mm and thickness 2.6 mm, manufactured by Sensirion (http://www.sensirion.com). A capacitive sensor element is used to measure the relative humidity, while the temperature is measured by a band-gap sensor. A calibrated digital output is given for ease of connection to a microcontroller. The relative humidity is measured with an accuracy of ±4.5%RH, and the temperature accuracy is ±0.5 °C.

Because the sensor is very small, it is available as mounted on a small printed circuit board for ease of handling. Figure 6.67 shows a picture of the SHT11 sensor.

The sensor is operated with four pins. Pin 1 and pin 4 are the supply voltage and ground pins. Pin 2 and pin 3 are the data and clock pins, respectively. The clock pin synchronizes all the activities of the chip. It is recommended by the manufacturers that the data pin should be pulled high through a 10 K resistor, and a 100 nF decoupling capacitor should be connected between the power lines.

The SHT11 is based on serial communication where data are clocked in and out, in synchronization with the SCK clock. The communication between the SHT11 and a microcontroller consists of the following protocols (see the SHT11 data sheet for more detailed information).

RESET

At the beginning of data transmission, it is recommended to send a RESET to the SHT11 just in case the communication with the device is lost. This signal consists of sending nine

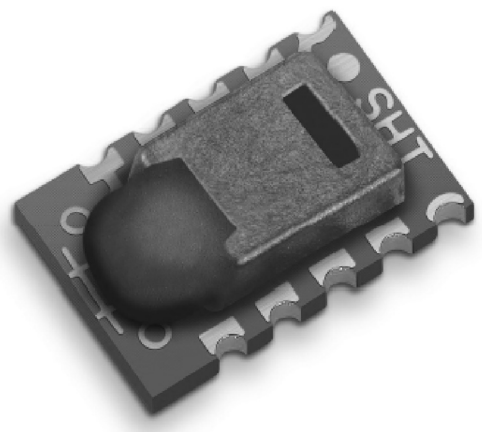


Figure 6.67: The SHT11 Temperature and Relative Humidity Sensor.

or more SCK signals while the DATA line is HIGH. Configuring the port pin as an input will force the pin to logic HIGH. A Transmission Start Sequence must follow the RESET.

The C code to implement the RESET sequence as a function is given below (SDA and SCK are the DATA and SCK lines, respectively). Note that the manufacturer's data sheet specifies that after SCK changes state it must remain in its new state for a minimum of 100 ns. Here, a delay of 1 μs is introduced between each SCK state change:

```
void Reset_Sequence()
{
 SCK = 0;                          // SCK low
 SDA_Direction = 1;                // Define SDA as input so that the SDA line
                                      becomes HIGH
 for (j = 0; j < 10; j++)          // Repeat 10 times
 {
  SCK = 1;                         // send 10 clocks on SCK line with 1 us delay
  Delay_us(1);                     // 1 us delay
  SCK = 0;                         // SCK is LOW
  Delay_us(1);                     // 1 us delay
 }
Transmission_Start_Sequence();     // Send Transmission-start-sequence
}
```

Transmission Start Sequence

Before a temperature or relative humidity conversion command is sent to the SHT11, the transmission start sequence must be sent. This sequence consists of lowering the DATA line while SCK is HIGH, followed by a pulse on the SCK and rising DATA again while the SCK is still HIGH.

The C code to implement the transmission start sequence is given below:

```
void Transmission_Start_Sequence()
{
 SDA_Direction = 1;      // Set SDA HIGH
 SCK = 1;                // SCK HIGH
 Delay_us(1);            // 1 us delay
 SDA_Direction = 0;      // SDA as output
 SDA = 0;                // Set SDA LOW
 Delay_us(1);            // 1 us delay
 SCK = 0;                // SCK LOW
 Delay_us(1);            // 1 us delay
 SCK = 1;                // SCK HIGH
 Delay_us(1);            // 1 us delay
 SDA_Direction = 1;      // Set SDA HIGH
 Delay_us(1);            // 1 us delay
 SCK = 0;                // SCK LOW
}
```

Conversion Command

After sending the transmission start sequence, the device is ready to receive a conversion command. This consists of three address bits (only "000" is supported) followed by five command bits. The list of valid commands is given in Table 6.1. For example, the commands for relative humidity and temperature are "00000101" and "00000011", respectively. After issuing a measurement command, the sensor sends an ACK pulse on the falling edge of the eighth SCK pulse. The ACK pulse is identified by the DATA line going LOW. The DATA line remains LOW until the ninth SCK pulse goes LOW. The microcontroller then has to wait for the measurement to complete. This can take up to 320 ms. During this time, it is recommended to stop generating clocks on the SCK line and release the DATA line. When the measurement is complete, the sensor pulls the DATA line LOW to indicate that the data are ready. At this point, the microcontroller can restart the clock on the SCK line to read the measured data. Note that the data are kept in the SHT11 internal memory until they are read out by the microcontroller.

The data read out consists of 2 bytes of data and 1 byte of CRC checksum. The checksum is optional and if not used the microcontroller may terminate the communication by keeping the DATA line HIGH after receiving the last bit of the data (LSB). The data bytes are transferred with MSB first and are right justified. The measurement can be for 8, 12, or 14 bits wide. Thus, the fifth SCK corresponds to the MSB data for a 12-bit operation. For an 8-bit measurement, the first byte is not used. The microcontroller must acknowledge each byte by pulling the DATA line LOW, and sending an SCK pulse. The device returns to the sleep mode after all the data have been read out.

Acknowledgment

After receiving a command from the microcontroller, the sensor issues an acknowledgment pulse by pulling the DATA line LOW for one clock cycle. This takes place after the falling edge of the eighth clock on the SCK line, and the DATA line is pulled LOW until the end of the ninth clock on the SCK line.

The Status Register

The Status register is an internal 8-bit register that controls some functions of the device, such as selecting the measurement resolution, end of battery detection, and use of the internal

Table 6.1: List of Valid Commands

Command	Code
00011	Measure temperature
00101	Measure relative humidity
00111	Read status register
00110	Write status register
11110	Soft reset (reset interface, clear status register)

heater. To write a byte to the Status register, the microcontroller must send the write command ("00110"), followed by the data byte to be written. Note that the sensor generates acknowledge signals in response to receiving both the command and the data byte. Bit 0 of the Status register controls the resolution, such that when this bit is 1, both the temperature resolution and the relative humidity resolution are 12 bits. When this bit is 0 (the default state), the temperature resolution is 14 bits, and the relative humidity resolution is 12 bits.

The sensor includes an on-chip heating element that can be enabled by setting bit 2 of the Status register (the heater is off by default). By using the heater, it is possible to increase the sensor temperature by 5−10 °C. The heater can be useful for analyzing the effects of changing the temperature on humidity. Note that, during temperature measurements, the sensor measures the temperature of the heated sensor element and not the ambient temperature.

The steps for reading the humidity and temperature are summarized below:

Humidity (Assuming 12-Bit Operation with No CRC)

- Send Reset_Sequence.
- Send Transmission_start_sequence.
- Send "00000101" to convert relative humidity.
- Receive ACK from sensor on eighth SCK pulse going LOW. The ACK is identified by the sensor lowering the DATA line.
- Wait for the measurement to be complete (up to 320 ms), or until DATA line is LOW.
- Ignore first four SCK pulses.
- Get the four upper nibble starting with the MSB bit.
- Send ACK to sensor at the end of eighth clock by lowering the DATA line and sending a pulse on the SCK.
- Receive low 8 bits.
- Ignore the CRC by keeping the DATA line HIGH.
- The next measurement can start by repeating the above steps.

Temperature

The steps for reading the temperature are similar, except that the command "00000011" is sent instead of "00000101".

Conversion of Signal Output

Relative Humidity Reading (SO_{RH})

The humidity sensor is nonlinear, and it is necessary to perform a calculation to obtain the correct reading. The manufacturer's data sheet recommends the following formula for the correction:

$$RH_{linear} = C_1 + C_2 + SO_{RH} + C_3 \cdot SO_{RH}^2 (\%RH) \tag{6.3}$$

Table 6.2: Coefficients for the RH Nonlinearity Correction

SO_{RH}	C_1	C_2	C_3
12 bits	−2.0468	0.0367	−1.5955E-6
8 bits	−2.0468	0.5872	−4.0845E-4

Where SO_{RH} is the value read from the sensor, and the coefficients are as given in Table 6.2.

For temperatures significantly different from 25 °C, the manufacturers recommend another correction to be applied to the relative humidity as follows:

$$RH_{TRUE} = (T - 25)\cdot(t_1 + t_2\cdot SO_{RH}) + RH_{linear} \tag{6.4}$$

Where T is the temperature in degrees centigrade where the relative humidity reading is taken, and the coefficients are as given in Table 6.3.

Temperature Reading (SO_T)

The manufacturers recommend that the temperature reading of the SHT11 should be corrected according to the following formula:

$$T_{TRUE} = d_1 + d_2\cdot SO_T \tag{6.5}$$

Where SO_T is the value read from the sensor, and the coefficients are as given in Table 6.4.

Block Diagram

The block diagram of the project is shown in Figure 6.68.

Circuit Diagram

The circuit diagram of the project is as shown in Figure 6.69. The SHT11 sensor is connected to PORTC of a PIC18F45K22 microcontroller, operated from an 8-MHz crystal. The DATA and SCK pins are connected to RC4 and RC3, respectively. The DATA pin is pulled up using a 10 K resistor as recommended by the manufacturers. Also, a 100 nF decoupling capacitor is connected between the V_{DD} pin and the ground.

Table 6.3: Coefficients for RH Temperature Correction

SO_{RH}	t_1	t_2
12 bits	0.01	0.00008
8 bits	0.01	0.00128

Table 6.4: Coefficients for Temperature Correction

V_{DD}	d_1 (°C)	d_1 (°F)
5	−40.1	−40.2
4	−39.8	−39.6
3.5	−39.7	−39.5
3	−39.6	−39.3
2.5	−39.4	−38.9
SO_T	**D_2(°C)**	**D_2(°F)**
14 bits	0.01	0.018
12 bits	0.04	0.072

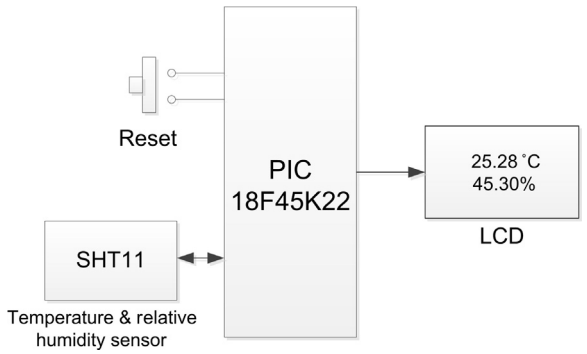


Figure 6.68: Block Diagram of the Project.

The project was tested using a plug-in SHT11 module (manufactured by mikroElektronika) together with the EasyPIC V7 development board, where the module was connected to the PORTC I/O connector located at the edge of the EasyPIC V7 development board (Figure 6.70).

Project PDL

The PDL of this project is given in Figure 6.71.

Project Program

mikroC Pro for PIC

The program listing of the project is shown in Figure 6.72 (MIKROC-SHT11.C). At the beginning of the program, the connections between the LCD and the microcontroller are defined. Then, the connections between the SHT11 sensor and the microcontroller are

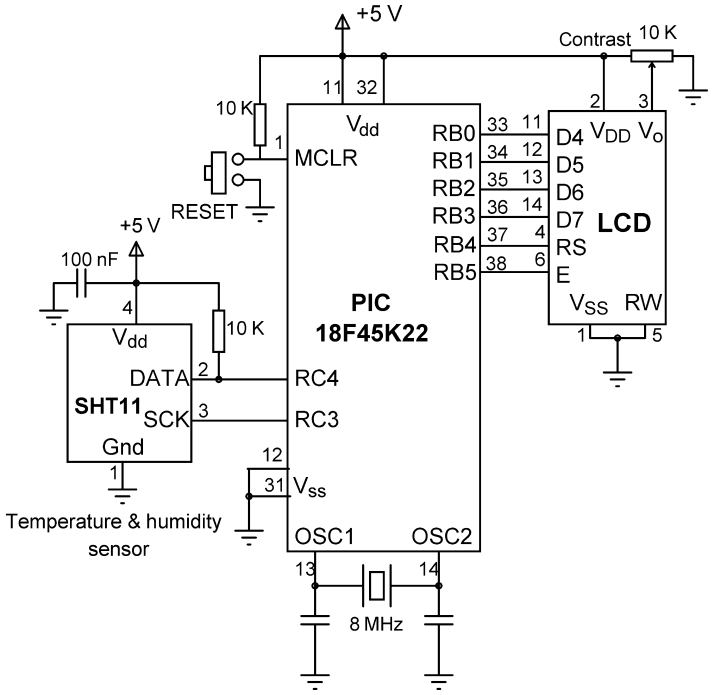


Figure 6.69: Circuit Diagram of the Project.

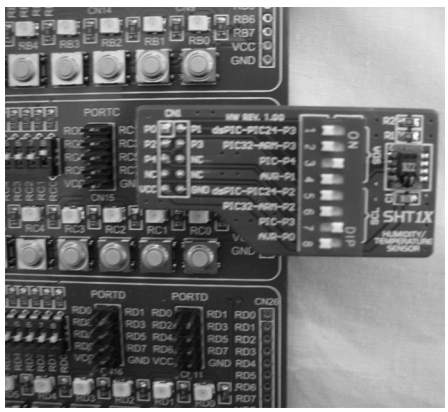


Figure 6.70: Use of the SHT11 Module with the EasyPIC V7 Development Board.

```
BEGIN
    Define the connections between the LCD and microcontroller
    Define the connections between the SHT11 and microcontroller
    Define SHT11 correction coefficients
    Configure PORTB, PORTC, PORTD as digital
    Initialise LCD
    Clear display
    CALL SHT11_Startup_Delay
    DO FOREVER
        CALL Measure to measure temperature
        CALL Measure to measure relative humidity
        Convert temperature to a string
        Convert relative humidity to a string
        Append degree symbol and letter "C" after the temperature value
        Append % sign after the relative humidity value
        Display temperature
        Display relative humidity
        Wait for 1 s
    ENDDO
END


BEGIN/SHT11_Startup_Delay
    Wait for 20 ms
END/SHT11_Startup_Delay

BEGIN/Reset_Sequence
    Implement SHT11 reset sequence
END/Reset_Seqeunce

BEGIN/Transmission_Start_Sequence
    Implement SHT11 transmission_start_sequence
END/Transmission_Start_Sequence

BEGIN/Send_ACK
    Send ACK signal to SHT11
END/Send_ACK

BEGIN/Measure
    Get type of measurement required
    Send Reset_Sequence
    Send Transmission_Start_Sequence
    Send address and temperature or humidity convert command to SHT11
    Send SCK pulse for the ACK signal
    Wait until measurement is ready (until DATA goes LOW)
    Read 8 bit measurement data
    Send ACK to SHT11
    Read remaining 8 bits
    Make corrections for temperature (or humidity)
END/Measure
```

Figure 6.71: PDL of the Project.

```
/*================================================================================
                    TEMPERATURE AND RELATIVE HUMIDITY MEASUREMENT
                    ============================================

This project measures both the ambient temperature and the relative humidity and then
displays the readings on an LCD.

The SHT11 single chip temperature and relative humidity sensor is used in this project. The
sensor is connected as follows to a PIC18F45K22 type microcontroller, operating at 8 MHz:

 Sensor   Microcontroller Port
 DATA          RC4
 SCK           RC3

A 10K pull-up resistor is used on the DATA pin. In addition, a 100 nF decoupling capacitor is
used between the VDD and the GND pins. The sensor is operated from a +5 V supply.

The connections between the LCD and the microcontroller is as in the earlier LCD based projects.

Author:  Dogan Ibrahim
Date:    September 2013
File:    MIKROC-SHT11.C
================================================================================*/
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

//SHT11 connections
sbit SHT11_SDA at RC4_bit;                          // SHT11 DATA pin
sbit SHT11_SCK at RC3_bit;                          // SHT11 SCK pin
sbit SHT11_SDA_Direction at TRISC4_bit;             // DATA pin direction
sbit SHT11_SCK_Direction at TRISC3_bit;             // SCK pin direction
//
// SHT11 Constants for calculating humidity (in 12 bit mode)
//
const float C1 = –2.0468;                           // –2.0468
const float C2 = 0.0367;                            // 0.0367
const float C3 = –1.5955E–6;                        // –1.5955* 10^–6
//
```

Figure 6.72: mikroC Pro for PIC Program Listing.

```
// SHT11 Constants for relative humidity temperature correction (in 12 bit mode)
//
const float t1 = 0.01;                                  // 0.01
const float t2 = 0.00008;                               // 0.00008
//
// SHT11 temperature conversion coefficients (14 bit mode)
//
const float d1 = –40.1;                                 // –40.1
const float d2 = 0.01;                                  // 0.01

unsigned char i, mode;
unsigned int buffer;
float Res, Ttrue, RHtrue;
char T[] = "T=          ";
char H[] = "H=          ";

//
// Function to send the Transmission_Start_Sequence
//
void Transmission_Start_Sequence(void)
{
 SHT11_SDA_Direction = 1;                               // Set SDA as input
 SHT11_SCK = 1;                                         // SCK HIGH
 Delay_us(1);                                           // 1 us delay
 SHT11_SDA_Direction = 0;                               // Set SDA as output
 SHT11_SDA = 0;                                         // SDA LOW
 Delay_us(1);                                           // 1 us delay
 SHT11_SCK = 0;                                         // SCK LOW
 Delay_us(1);                                           // 1 us delay
 SHT11_SCK = 1;                                         // SCK HIGH
 Delay_us(1);                                           // 1 us delay
 SHT11_SDA_Direction = 1;                               // Set SDA as input
 Delay_us(1);                                           // 1 us delay
 SHT11_SCK = 0;                                         // SCK low
}

//
// This function sends the Reset_Sequence
//
void Reset_Sequence()
{
    SHT11_SCK = 0;                                      // SCL low
    SHT11_SDA_Direction = 1;                            // Define SDA as input
    for (i = 1; i <= 10; i++)                           // Repeat 10 times
 {
    SHT11_SCK = 1;                                      // Send clock pulses
    Delay_us(1);
    SHT11_SCK = 0;
    Delay_us(1);
 }
 Transmission_Start_Sequence();
}
```

Figure 6.72
cont'd

```
//
// This function sends ACK
//
void Send_ACK()
{
 SHT11_SDA_Direction = 0;                    // Define SDA as output
 SHT11_SDA = 0;                              // SDA low
 SHT11_SCK = 1;                              // SCL high
 Delay_us(1);                                // 1 us delay
 SHT11_SCK = 0;                              // SCL low
 Delay_us(1);                                // 1 us delay
 SHT11_SDA_Direction = 1;                    // Define SDA as input
}


//
// This function returns temperature or humidity depending on the argument
//
float Measure(unsigned char command)
{
  mode = command;                            // Mode is 3 or 5
  Reset_Sequence();                          // Reset SHT11
  Transmission_Start_Sequence();             // Start transmission sequence

  SHT11_SDA_Direction = 0;                   // Set SDA as output
  SHT11_SCK = 0;                             // Set SCK as LOW
//
// Send address and command to SHT11 sensor. A total of 8 bits are sent
//
  for(i = 0; i < 8; i++)                     // Send address and command
  {
    if (mode.F7 == 1)SHT11_SDA_Direction = 1;// If MSB (bit 7) is 1, Set SDA to 1
    else                                     // If MSB is 0
    {                                        // else MSB is 0
      SHT11_SDA_Direction = 0;               // define SDA as output
      SHT11_SDA = 0;                         // Set SDA to 0
    }
    Delay_us(1);                             // 1 us delay
    SHT11_SCK = 1;                           // SCL high
    Delay_us(1);                             // 1 us delay
    SHT11_SCK = 0;                           // SCL low
    mode = mode << 1;                        / Move contents of j one place left
   }
//
// Give a SCK pulse for the ACK
//
  SHT11_SDA_Direction = 1;                   // Set SDA to input (to read ACK)
  SHT11_SCK = 1;                             // SCL high
```

Figure 6.72
cont'd

```
  Delay_us(1);                                          // 1 us delay
  SHT11_SCK = 0;                                        // SCL low
  Delay_us(1);                                          // 1 us delay
//
// Now wait until the measurement is ready (SDA goes LOW when data becomes ready)
//
  while (SHT11_SDA == 1)Delay_us(1);                    // Wait until SDA goes LOW
//
// Now, the data is ready, read the data as 2 bytes. Read all 16 bits even though the
// upper nibble may not be relevant
//
 buffer = 0;
 for (i = 1; i <=16; i++)                               // DO 16 times
 {
    buffer = buffer << 1;                               // Move MSB one place left
    SHT11_SCK = 1;                                      // SCK HIGH
    if (SHT11_SDA == 1)buffer = buffer | 0x0001;        // Get the bit as 1 (OR with data)
    SHT11_SCK = 0;
    if (i == 8)Send_ACK();                              // If counter i = 8 then send ACK
 }

//
// Now make the corrections to the measured value. If mode = 3 then temperature, if on the
// other hand, mode = 5 then relative humidity
//
 if(command == 0x03)                                    // Temperature correction
   Res = d1 + d2*buffer;
 else if(command == 0x05)                               // Relative humidity correction
 {
   Res = C1 + C2*buffer + C3*buffer*buffer;
   Res = (Ttrue – 25)*(t1 + t2*buffer) + Res;
 }
 return Res;                                            // Return temperature or humidity
}


//
// This is the SHT11 startup delay (20 ms)
//
void SHT11_Startup_Delay()
{
 Delay_ms(20);
}



//
// Start of MAIN program
//
void main()
{
```

Figure 6.72
cont'd

```
  ANSELB = 0;                                  // Configure PORT B as digital
  ANSELC = 0;                                  // Configure PORT C as digital
  TRISB = 0;
  TRISC = 0;
  SHT11_SCK_Direction = 0;                     // SCL is output

  LCD_Init();                                  // Initialise LCD
  Lcd_Cmd(_LCD_CURSOR_OFF);                    // Disable cursor
  SHT11_Startup_Delay();                       // SHT11 startup delay

  for(;;)                                      // DO FOREVER
  {
   SHT11_SCK_Direction = 0;                    // Define SCL1 as output
   Ttrue = Measure(0x03);                      // Measure Temperature
   RHtrue = Measure(0x05);                     // Measure Relative humidity
   SHT11_SCK_Direction = 1;                    // Define SCK as input
   FloatToStr(Ttrue, T+3);                     // Convert temperature to string
   FloatToStr(RHtrue, H+3);                    // Convert rel humidity to string

   Lcd_Cmd(_LCD_CLEAR);                        // Clear display
   T[8] = 178;                                 // Insert Degree sign
   T[9] = 'C';                                 // Insert C
   T[10] = 0x0;                                // Terminate with NULL
   H[8] = '%';                                 // Insert %
   H[9]=0x0;                                   // Terminate with NULL
   Lcd_Out(1,1,T);                             // Display temperature
   Lcd_Out(2,1,H);                             // Display humidity
   Delay_ms(1000);                             // Delay 1 s
  }
}
```

Figure 6.72
cont'd

defined. The temperature and relative humidity correction coefficients are then given as floating point numbers.

The main program then configures PORTB, PORTC, and PORTD as digital outputs, initializes the LCD, and clears the display. The program then enters an endless loop formed with a *for* statement. Inside this loop, the temperature is measured and corrected by calling function Measure with argument 3, and stored in a floating point variable T_{true}. Then, the relative humidity is read, corrected, and stored in floating point variable RH_{true}. The measured values are converted into strings by using built-in function FloatToStr. Finally, the degree symbol and letter "C" are appended to the temperature reading. Similarly, symbol "%" is appended to the relative humidity reading before it is displayed.

Function Measure is the most complicated function in the program. This function implements the measurement steps described earlier in the project. Argument *command* specifies the type of measurement required: 3 for temperature measurement and 5 for relative humidity measurement. After calling to functions Reset_Sequence and Transmission_Start_Sequence, the address and command are sent to the SHT11 device.

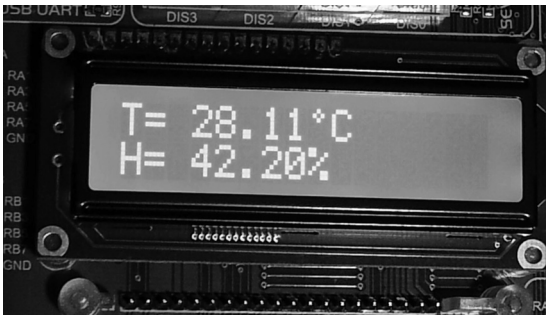


Figure 6.73: A Typical Display.

Bits of *mode* (3 or 5) are sent out through the MSB after shifting the data to the left in a loop. The program then waits until the conversion is ready, which is indicated by the DATA line going LOW. Once the data are ready, a loop is formed to read the 2 bytes from the sensor. At the end of the eighth clock pulse, an ACK signal is sent to the sensor. In the last part of this function, depending upon the type of conversion required, either the temperature or the relative humidity readings are corrected and returned to the calling program.

The program repeats after a delay of 1 s.

Figure 6.73 shows a typical display.

Project 6.10—Thermometer with an RS232 Serial Output

Project Description

Serial communication is a simple means of sending data to long distances quickly and reliably. The most commonly used serial communication method is based on the RS232 standard. In this standard, data are sent over a single line from a transmitting device to a receiving device in bit serial format at a prespecified speed, also known as the Baud rate, or the number of bits sent each second. Typical Baud rates are 4800, 9600, 19200, 38400 etc.

The RS232 serial communication is a form of asynchronous data transmission where data are sent character by character. Each character is preceded with a Start bit, seven or eight data bits, an optional parity bit, and one or more stop bits. The most commonly used format is eight data bits, no parity bit, and one stop bit. The least significant data bit is transmitted first, and the most significant bit is transmitted last.

A logic high is defined to be at -12 V, and a logic 0 is at $+12$ V. Figure 6.74 shows how character "A" (ASCII binary pattern 0010 0001) is transmitted over a serial line. The line is normally idle at -12 V. The start bit is first sent by the line going from high to low.

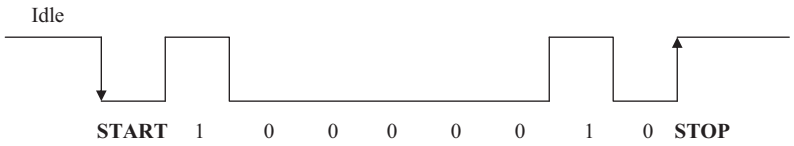


Figure 6.74: Sending Character "A" in Serial Format.

Then eight data bits are sent starting from the least significant bit. Finally, the stop bit is sent by raising the line from low to high.

In serial connection, a minimum of three lines are used for communication: transmit (TX), receive (RX), and ground (GND). Some high-speed serial communication systems use additional control signals for synchronization, such as CTS, DTR, and so on. Some systems use software synchronization techniques where a special character (XOFF) is used to tell the sender to stop sending, and another character (XON) is used to tell the sender to restart transmission. In this book, we will be using low-speed communication, and therefore, the basic pins shown in Table 6.5 will be used with no hardware or software synchronization.

Serial devices are connected to each other using two types of connectors: a nine-way connector and a 25-way connector. Table 6.5 shows the TX, RX, and GND pins of each types of connectors. The connectors used in RS232 serial communication are shown in Figure 6.75.

As described above, RS232 voltage levels are at ±12 V. On the other hand, microcontroller input−output ports operate at 0 to +5 V voltage levels. It is therefore necessary to translate the voltage levels before a microcontroller can be connected to an RS232 compatible device. Thus, the output signal from the microcontroller has to be converted into ±12 V, and the input from an RS232 device must be converted into 0 to +5 V before it can be connected to a microcontroller. This voltage translation is normally done using special RS232 voltage converter chips. One such popular chip is the MAX232.

Table 6.5: Minimum Required Pins for Serial Communication

Pin	Function
Nine-Pin Connector	
2	Transmit (TX)
3	Receive (RX)
5	Ground (GND)
Twenty Five-Pin Connector	
2	Transmit (TX)
3	Receive (RX)
7	Ground (GND)

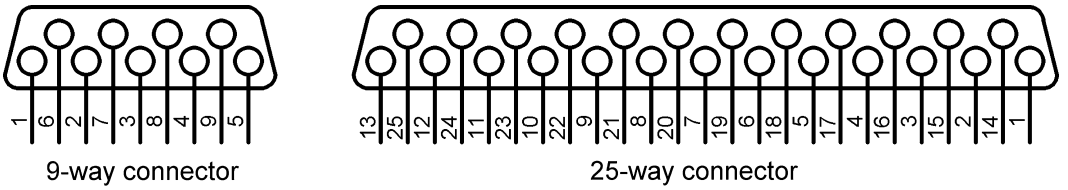


Figure 6.75: RS232 Connectors.

This is a dual converter chip having the pin configuration as shown in Figure 6.76. This particular device requires four external 1-μF capacitors for its operation.

In the PIC18F series of microcontrollers, serial communication can be handled either in the hardware or in the software. The use of the hardware option is easy. The PIC18F series of microcontrollers have built-in Universal Asynchronous Receiver Transmitter (USART) circuits providing special input−output pins for serial communication. For serial communication, all the data transmission is handled by the USART, but we have to configure the USART before receiving and transmitting data. With the software option, all the serial bit timing is handled in software and any input−output pin can be programmed and used for serial communication. In this book, we will only use the hardware UART functions.

In this project, a PC is connected to the microcontroller using an RS232 cable. The project receives the ambient temperature every second and then sends it to the PC where it is displayed on the PC screen.

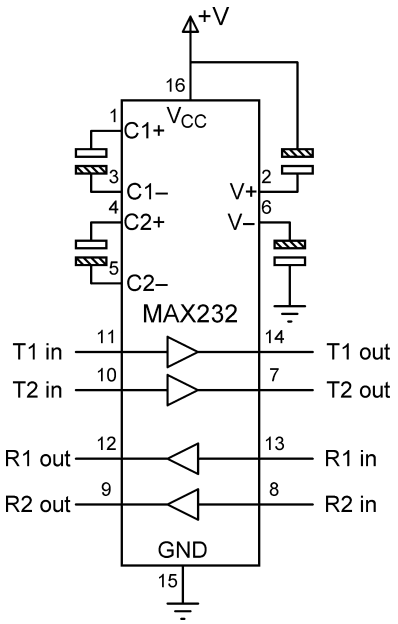


Figure 6.76: MAX232 Pin Configuration.

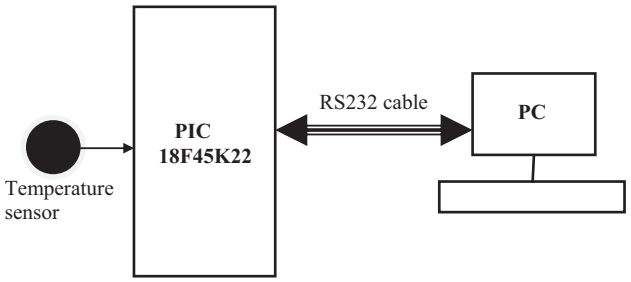


Figure 6.77: Block Diagram of the Project.

The block diagram of the project is shown in Figure 6.77.

Project Hardware

The circuit diagram of the project is shown in Figure 6.78. In this project, a PIC18F45K22 microcontroller is used with an 8 MHz crystal. The built-in USART of the microcontroller is used in this project. The serial communication lines of the microcontroller (RC6 and RC7) are connected to a MAX232 voltage translator chip and then to the serial input port (COM1) of a PC using a nine-pin connector. An LM35DZ-type analog temperature is connected to port pin AN0 (RA0) of the microcontroller.

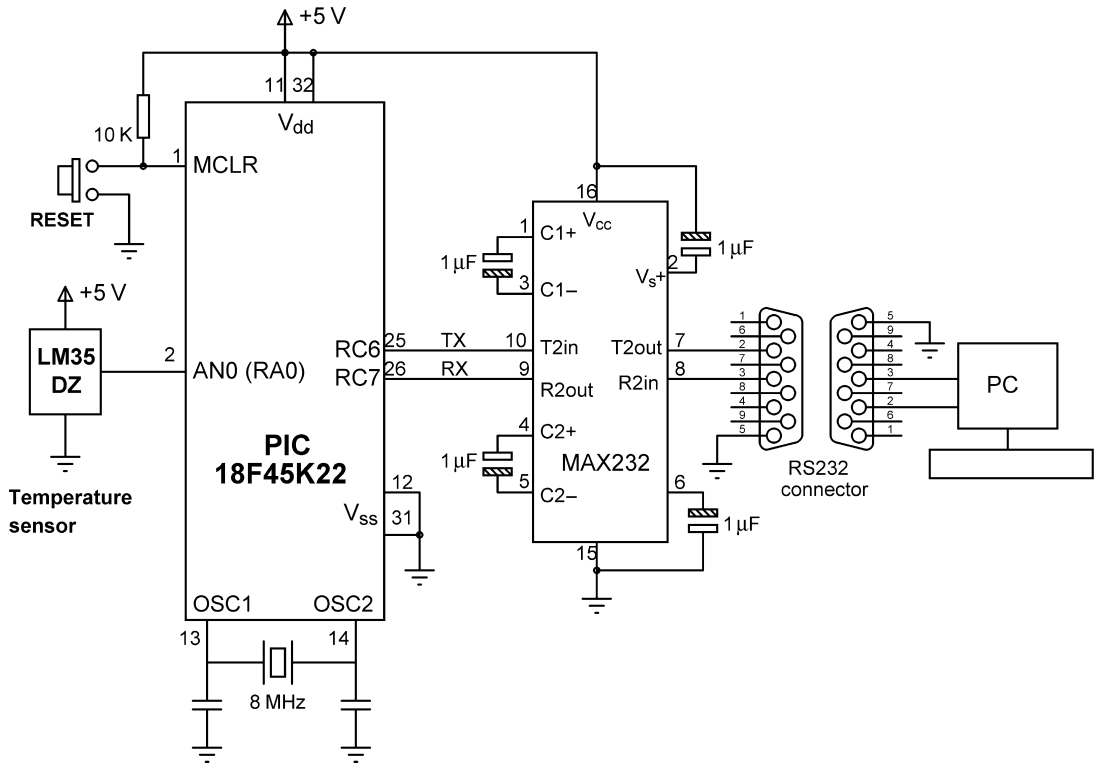


Figure 6.78: Circuit Diagram of the Project.

```
BEGIN/NEWLINE
        Send carriage return to USART
        Send line feed to USART
END/NEWLINE


Main program:

BEGIN
        Configure USART to 9600 Baud
        Display "AMBIENT TEMPERATURE"
        CALL Newline
        Display "=================="
        CALL Newline
        DO FOREVER
                Read temperature from analog sensor
                Convert temperature to Degrees C
                Send temperature over the serial line
                CALL Newline
                Wait 1 s
        ENDDO
END
```

Figure 6.79: Project PDL.

If you are using the EasyPIC V7 development board, the following jumpers must be configured on the board:

- Set Jumper J3 to RS232.
- Set Jumper J4 to RS232.
- Set DIL switch SW1, switch 1 to the ON position (to RC7).
- Set DIL switch SW2, switch 1 to the ON position (to RC6).
- Connect the development board RS232 port to your PC via a suitable RS232 cable.

Project PDL

The PDL of the project is shown in Figure 6.79. The project consists of a main program and a function called **Newline**. Function **Newline** sends a carriage return and line feed to the serial port. The main program reads the temperature every second and sends it to the PC through the serial link.

Project Program

mikroC Pro for PIC

The mikroC Pro for the PIC program listing of the project is shown in Figure 6.80 (MIKROC-RS232-1.C). The program consists of a main program and a function called **Newline**. Function **Newline** sends a carriage return and line feed to the USART to move the cursor to the next line. At the beginning of the program, the UART is initialized to operate at 9600 Baud, and the heading "AMBIENT TEMPERATURE" is sent to the serial line. Then, an endless loop is formed, and inside this loop, the temperature is read from

```
/*************************************************************************
                        THERMOMETER WITH RS232 SERIAL OUTPUT
                        ====================================

In this project a PC is connected to a PIC18F452K2 microcontroller. The project reads the ambient
temperature using an LM35DZ type analog temperature sensor. The temperature is then sent to
a PC over the RS232 communications line. The PC displays the temperature on the screen.

This program uses the built in USART of the microcontroller. The USART is configured to operate
with 9600 Baud rate.

The serial TX pin is RC6 and the serial RX pin is RC7.

Author:   Dogan Ibrahim
Date:     September 2013
File:     MIKROC-RS232-1.C
*************************************************************************/

//
// This functions send carriage-return and line-feed to USART
//
void Newline()
{
  Uart1_Write(0x0D);                    // Send carriage return
  Uart1_Write(0x0A);                    // Send line feed
}


//
// Start of MAIN program
//
void main()
{
   unsigned char Txt[14];
   unsigned char msg1[] = "AMBIENT TEMPERATURE";
   unsigned char msg2[] = "====================";
   unsigned int temp;
   float mV;
   char *res;

   ANSELA = 1;                          // Configure RA0 as analog
   ANSELC = 0;                          // Configure PORTC as digital
   TRISA = 1;                           // Configure RA0 as input

   Uart1_Init(9600);                    // Initialize UART to Baud = 9600

   Uart1_Write_Text(msg1);              // Display heading
   Newline();                           // Newline
   Uart1_Write_Text(msg2);
   Newline();
```

Figure 6.80: mikroC Pro for PIC Program.

```
    for(;;)                                  // Endless loop
    {

      temp = ADC_Read(0);                    // Read temperature from channel 0
      mV = temp * 5000.0/1024;               // Convert to millivolts
      mV = mV/10.0;                          // Convert to Degrees Centigrade
      FloatToStr(mV, Txt);                   // Convert into a string
      res = strchr(Txt, '.');                // Locate "."
      *(res+3) = '\0';                       // Insert NULL character 2 digits after "."
      Uart1_Write_Text(Txt);                 // Send temperature over the serial line
      Newline();
      Delay_Ms(1000);                        // Wait 1 s
    }
}
```

Figure 6.80
cont'd

channel 0 and is converted into millivolts after multiplying with 5000/1024. The result is then divided by 10 to give the temperature in Degrees Centigrade. The temperature is converted into a string using built-in function **FloatToStr**. The resulting string contains several digits after the decimal point as in the following example:

29.23687

The number of digits to display after the decimal point can be selected by finding the position of the decimal point and inserting a NULL character to terminate the string at the required point. In this program, the built-in string function **strchr** is used to find the address (pointer to) of the decimal point, and this is stored in character pointer **res**. Then, a NULL character is added to string **Txt** three digits after the decimal point. The result is that the string will be displayed with two digits after the decimal point. Thus, 29.23687 will be displayed as 29.23.

mikroC Pro for the PIC compiler supports the following hardware UART functions ("x" is 1 or 2 depending upon the UART module used in multiple UART processors):

UARTx_Init: This function initializes the UART module. The required Baud rate must be specified in the argument.

UARTx_Data_Ready: This function is used to check if data are available in the receive buffer. The function returns 1 if data are available for reading.

UARTx_Tx_Idle: This function is used to test if the transmit buffer is empty. The function returns 1 if the transmit buffer is empty.

UARTx_Read: This function reads a byte from the UART. Function UARTx_Data_-Ready should be used to make sure that data are available in the receive buffer.

UARTx_Read_Text: This function is used to read the text from UART. A delimiter and number of attempts to detect the delimiter should be specified to identify the end of data.

UARTx_Write: This function writes a byte to UART.
UARTx_Write_Text: This function is used to write zero-terminated text to the UART.
UART_Set_Active: Specify the UART to be used in processors having more than one UART.

Testing the Program

The program can be tested using a terminal emulator software such as **Hyperterminal**, which is distributed free of charge with the Windows operating systems. The steps to test the program are given below (these steps assume that serial port COM15 is used. If you are not sure which serial port is available on your PC, go to **Control Panel → System → Device Manager** and look for available Ports. Serial ports have identifiers COM, followed by a number):

- Connect the RS232 output from the microcontroller to the serial input port of a PC (e.g. COM15).
- Start Hyperterminal terminal emulation software and give a name to the session.
- Select File → New connection → Connect using and select COM15.
- Select the Baud rate as 9600, data bits as 8, no parity bits, 1 stop bit, and Flow Control None.
- Reset the microcontroller.

An example output from the Hyperterminal screen is shown in Figure 6.81.

Alternatively, the terminal emulator software included inside the mikroC Pro for PIC compiler can be used. The steps are given below (assuming that COM15 will be used):

- Click **Tools → USART Terminal**.
- Select COM15, 9600 Baud, 1 Stop bit, No parity, Flow control None, data Format ASCII (Figure 6.82).
- Click **Connect**. You should see the data being displayed on your terminal (Figure 6.83).

Using USB-RS232 Converter Cable

Most PCs nowadays do not have RS232 serial ports; instead, they are equipped with universal serial bus (USB) ports. If this is the case, you should be able to purchase and use a USB-RS232 converter cable. This is a special cable with built-in electronics to convert RS232 signals into USB and vice versa. When such a cable is connected to a PC, it creates a virtual serial communications port on your PC. You should go to **Control Panel → System → Device Manager** to see the name of the created COM port, and then use a terminal emulator software such as the ones given above, that is, Hyperterm or the mikroC Pro for PIC built-in terminal emulator.

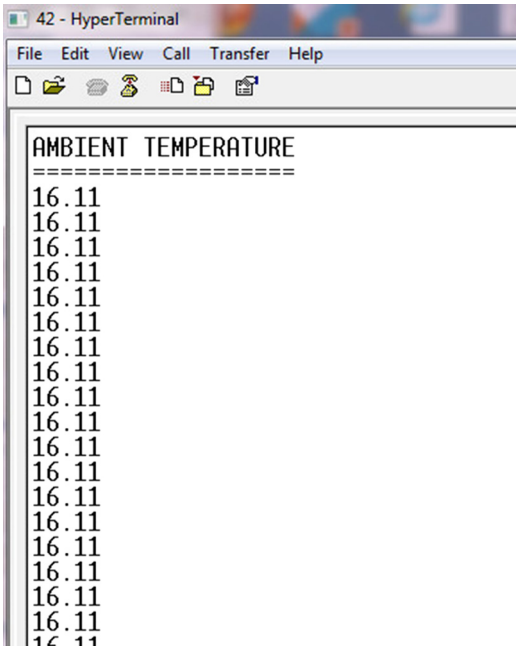


Figure 6.81: Hyperterminal Screen.

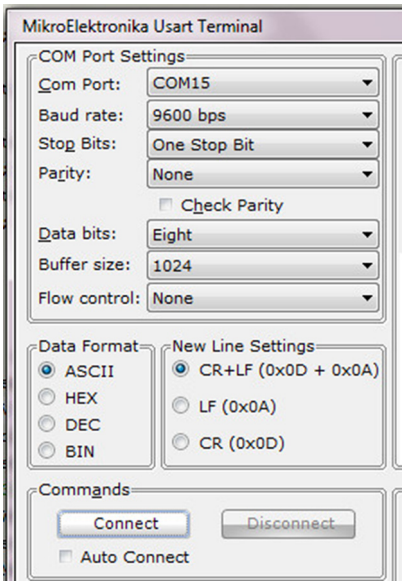


Figure 6.82: Use of the mikroC Pro for the PIC Built-in Terminal Emulator.

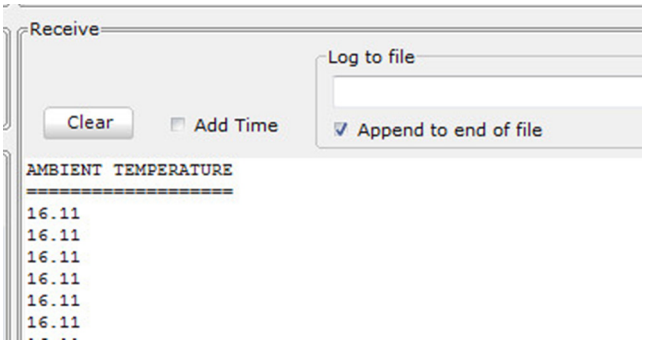


Figure 6.83: Displaying Data Using the mikroC Pro for the PIC Terminal Emulator.

Using the USB UART Port

The EasyPIC V7 development board has an on-board USB-RS232 converter module that enables you to connect the development board to the USB port on your PC. The steps to use this port are as follows:

- Set Jumper J3 to USB UART.
- Set Jumper J4 to USB UART.
- Set DIL switch SW1, switch 1 to ON position (to RC7).
- Set DIL switch SW2, switch 1 to ON position (to RC6).
- Connect the development board USB UART port to your PC via a USB cable.

Figure 6.84 shows the EasyPIC V7 development board configured for USB UART operation.

Note that the USB UART module on the development board uses the FT232RL chip to convert the signals. This chip requires the FTDI driver (known as VCP_DRIVERS) to be installed on your PC before you can use the USB UART communication. This driver is available on the EasyPIC V7 product DVD. Alternatively, it can be downloaded free of charge from the Internet.

MPLAB XC8

The mikroC Pro for PIC program listing of the project is shown in Figure 6.85 (MIKROC-RS232-1.C). Some of the important MPLAB XC8 hardware UART library functions are given below ("x" is either 1 or 2 and specifies the UART to be used in processors having more than one UART):

BusyxUSART: This function returns 1 if the UART transmitter is busy.
ClosexUSART: This function disables the UART.

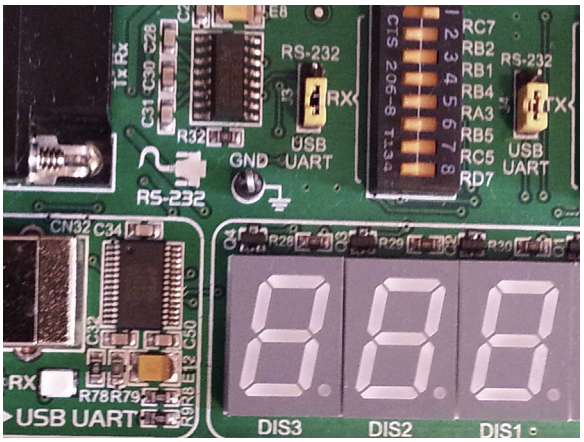


Figure 6.84: EasyPIC V7 Board Configured for the USB UART Operation. (For color version of this figure, the reader is referred to the online version of this book.)

DataRdyxUSART: This function returns 1 if data are available in the UART receive buffer.

getsxUSART: This function reads a fixed length of characters from UART. The number of characters to be read is specified in the function argument.

OpenxUSART: This function configures the specified UART. The function has two arguments. The first argument can be a bitwise AND of the following identifiers:

Interrupt on Transmission:

`USART_TX_INT_ON`	Transmit interrupt ON
`USART_TX_INT_OFF`	Transmit interrupt OFF

Interrupt on Receipt:

`USART_RX_INT_ON`	Receive interrupt ON
`USART_RX_INT_OFF`	Receive interrupt OFF

USART Mode:

`USART_ASYNCH_MODE`	Asynchronous Mode
`USART_SYNCH_MODE`	Synchronous Mode

```
/************************************************************************
                    THERMOMETER WITH RS232 SERIAL OUTPUT
                    ====================================

In this project a PC is connected to a PIC18F452K2 microcontroller. The project reads the
ambient temperature using an LM35DZ type analog temperature sensor. The temperature is
then sent to a PC over the RS232 communications line. The PC displays the temperature on the
screen.

This program uses the built in USART of the microcontroller. The USART is configured to operate
with 9600 Baud rate.

The serial TX pin is RC6 and the serial RX pin is RC7.

Author:   Dogan Ibrahim
Date:     September 2013
File:     XC8-RS232-1.C
************************************************************************/
#include <xc.h>
#include <string.h>
#include <plib/usart.h>
#include <plib/adc.h>
//#include <strings.h>
#include <stdlib.h>
#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

//
// This function creates seconds delay. The argument specifies the delay time in seconds.
//
void Delay_Seconds(unsigned char s)
{
  unsigned char i,j;

  for(j = 0; j < s; j++)
  {
    for(i = 0; i <  100; i++)__delay_ms(10);
  }
}


void Newline()
{
  Write1USART(0x0D);                    // Send carriage return
  while(Busy1USART());                  // Wait while UART is busy
  Write1USART(0x0A);                    // Send line feed
}


//
// Start of MAIN program
```

Figure 6.85: MPLAB XC8 Program.

```
//
void main()
{
   char *Txt;
   unsigned char msg1[] = "AMBIENT TEMPERATURE";
   unsigned char msg2[] = "====================";
   unsigned int temp;
   float mV;
   char *res;
   int status;

   ANSELA = 1;                                   // Configure RA0 as analog
   ANSELC = 0;                                   // Configure PORTC as digital
   TRISA = 1;                                    // Configure RA0 as input

   Open1USART( USART_TX_INT_OFF    &
         USART_RX_INT_OFF    &
         USART_ASYNCH_MODE       &
         USART_EIGHT_BIT     &
         USART_CONT_RX           &
         USART_BRGH_LOW,
         12);

   putrs1USART(msg1);                            // Display heading
   Newline();                                    // Newline
   putrs1USART(msg2);
   Newline();
//
// Configure A/D converter
//
  OpenADC(ADC_FOSC_2 & ADC_RIGHT_JUST & ADC_12_TAD,
       ADC_CH0 & ADC_INT_OFF,
       ADC_TRIG_CTMU & ADC_REF_VDD_VDD & ADC_REF_VDD_VSS);


   for(;;)                                       // Endless loop
   {
     SelChanConvADC(ADC_CH0);                    // Select channel 0 and start conversion
     while(BusyADC());                           // Wait for completion
     temp = ReadADC();                           // Read converted data
     mV = temp * 5000.0/1024;                    // Convert to millivolt
     mV = mV/10.0;                               // Convert to Degrees Centigrade
     Txt = ftoa(mV, &status);                    // Convert to string

     res = strchr(Txt, '.');                     // Locate "."
     *(res+3) = '\0';                            // Insert NULL character 2 digits after "."
     putrs1USART(Txt);
     Newline();
     Delay_Seconds(1);                           // Wait 1 s
   }
}
```

Figure 6.85
cont'd

Transmission Width:

`USART_EIGHT_BIT`	8-Bit transmit/receive
`USART_NINE_BIT`	9-Bit transmit/receive

Slave/Master Select (synchronous mode only):

`USART_SYNC_SLAVE`	Synchronous Slave mode
`USART_SYNC_MASTER`	Synchronous Master mode

Reception mode:

`USART_SINGLE_RX`	Single reception
`USART_CONT_RX`	Continuous reception

Baud rate:

`USART_BRGH_HIGH`	High baud rate
`USART_BRGH_LOW`	Low baud rate

The second argument specifies the value to be written to the Baud rate generator register for the required Baud rate. The formula used to determine the Baud rate is (F_{OSC} is the oscillator frequency):

Asynchronous mode, high speed:

```
FOSC/(16 * (spbrg + 1))
```

Asynchronous mode, low speed:

```
FOSC/(64 * (spbrg + 1))
```

Synchronous mode:

```
FOSC/(4 * (spbrg + 1))
```

putrsxUSART: This function writes a string of characters to the UART, including the NULL character.

ReadxUSART: This function reads 1 byte from the UART.

WritexUSART: This function writes 1 byte to the UART.

Note that in Figure 6.85 the header file usart.h must be included at the beginning of the program. The UART module is initialized by using the following identifiers:

```
Open1USART(USART_TX_INT_OFF   &
           USART_RX_INT_OFF   &
           USART_ASYNCH_MODE  &
           USART_EIGHT_BIT    &
           USART_CONT_RX      &
           USART_BRGH_LOW,
           12);
```

The second argument specifies the Baud rate. Using a low speed with the required 9600 Baud and clock rate of 8 MHz (F_{OSC}) $= 8 \times 10^6$ Hz, we have

$$9600 = F_{OSC}/(64 * (spbrg + 1))$$

or

$$spbrg = [F_{OSC}/(64 * 9600)] - 1 = 12$$

Project 6.11—Microcontroller and a PC-Based Calculator

Project Description

In this project, a PC is connected to the microcontroller using an RS232 cable. The project operates as a simple integer calculator where numbers and operation to be performed are sent to the microcontroller via the PC keyboard, and the results are displayed on the PC screen.

A sample calculation is as follows:

```
CALCULATOR PROGRAM

 Enter First Number: 12
Enter Second Number: 2
    Enter Operation: +
            Result = 14
```

Project Hardware

The circuit diagram of the project is shown in Figure 6.86. The serial communication lines of the microcontroller (RC6 and RC7) are connected to an MAX232 voltage translator chip and then to the serial input port (COM1) of a PC using a nine-pin connector.

Project PDL

The PDL of the project is shown in Figure 6.87. The main program receives two numbers and the operation to be performed from the PC keyboard. The numbers are echoed on the PC monitor. The result of the operation is also displayed on the monitor.

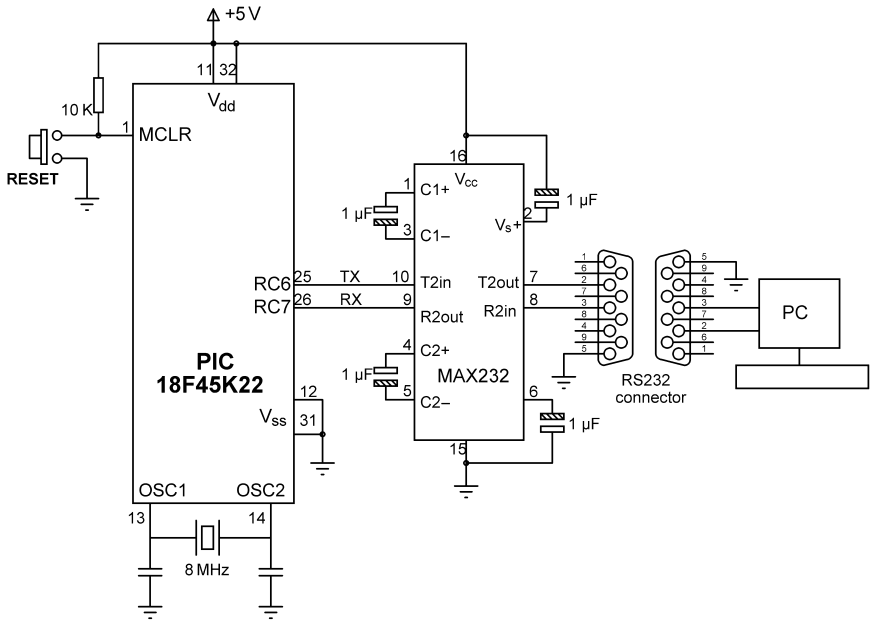


Figure 6.86: Circuit Diagram of the Project.

BEGIN/NEWLINE
Send carriage return to USART
Send line feed to USART
END/NEWLINE

Main program:

BEGIN
Configure USART to 9600 Baud
DO FOREVER
Display "CALCULATOR PROGRAM"
Display "Enter First Number:"
Read first number
Display "Enter Second Number:"
Read second number
Display "Operation:"
Read operation
Perform required operation
Convert result into string
Remove leading spaces from the result
Display "Result="
Display the result
ENDDO
END

Figure 6.87: Project PDL.

Project Program

The program listing of the project is shown in Figure 6.88 (MIKRO-RS232-2.C). At the beginning of the program, various messages used in the program are defined as **msg1** to **msg5**. The USART is then initialized to the 9600-Baud using mikroC library routine **Uart1_Init**. Then, the heading **CALCULATOR PROGRAM** is displayed on the PC monitor. The program reads the first number from the keyboard using the library function **Uart1_Read**. Function **Uart1_Data_Ready** checks when a new data byte is ready before reading it. Variable **Op1** stores the first number. Similarly, another loop is formed, and the second number is read into variable **Op2**. The program then reads the operation to be performed (+, −, *, /). The required operation is performed inside a switch statement, and the result is stored in variable **Calc**. The program then converts the result into string format by calling library function **LongToStr**. Leading blanks are removed from this string using the **Ltrim** built-in function, and the final result is stored in character array **op** and sent to the UART to display on the PC screen.

Testing the Program

The program can be tested by using a terminal emulator program on the PC as described in the previous project. If you are using the mikroC Pro for a PIC terminal emulator, you should make the following settings before clicking the Connect button:

New Line Settings: CR (0x0D)
Append New Line

Keyboard data should be entered at the top of the terminal emulator window and the Send button should be clicked.

Figure 6.89 shows a sample run of the program using the Hyperterm terminal emulator software.

Project 6.12—GPS with an LCD Output

Project Description

This project is about designing a global positioning system-based system to display the latitude and longitude of current position on an LCD display.

The GPS is a satellite navigation system that provides time and location information anywhere on the Earth, 24 h a day, and in all weather conditions. Currently, the system is heavily used by motorists, ships, and in the air. The system is maintained by the US government and is freely available to anyone who has a GPS receiver.

```
/*************************************************************************************
                                CALCULATOR WITH PC INTERFACE
                                ============================

In this project a PC is connected to a PIC18F45K22 microcontroller. The project is a simple integer
calculator. User enters the numbers using the PC keyboard.  Results are displayed on the PC
monitor.

The following operations can be performed:

  + – * /

This program uses the built in USART of the microcontroller. The USART is  configured to operate
with 9600 Baud rate.

The serial TX pin is RC6 and the serial RX pin is RC7.

Author:   Dogan Ibrahim
Date:     September 2013
File:     MIKRO-RS232-2.C
*************************************************************************************/

#define Enter 13
#define Plus '+'
#define Minus '–'
#define Multiply '*'
#define Divide '/'

//
// This functions send carriage-return and line-feed to USART
//
void Newline()
{
  Uart1_Write(0x0D);                          // Send carriage return
  Uart1_Write(0x0A);                          // Send line feed
}


void main()
{
   unsigned char MyKey, i,j,kbd[12];
   unsigned char *op;
   unsigned long Calc, Op1, Op2,Key;
   unsigned char msg1[] = "CALCULATOR PROGRAM";
   unsigned char msg2[] = "    Enter First Number: ";
   unsigned char msg3[]=  "  Enter Second Nummber: ";
   unsigned char msg4[] = "      Enter Operation: ";
   unsigned char msg5[] = "            Result = ";
```

Figure 6.88: mikroC Pro for PIC Program.

```
ANSELC = 0;
//
// Configure the USART
//
   Uart1_Init(9600);                          // Baud = 9600
//
// Program loop
//
    for(;;)                                   // Endless loop
    {
      MyKey = 0;
      Op1 = 0;
      Op2 = 0;

      Newline();                              // Send newline
      Newline();                              // Send newline
      Uart1_Write_Text(msg1);                        // Send TEXT
      Newline();                              // Send newline
      Newline();                              // Send newline

//
// Get the first number
//
      Uart1_Write_Text(msg2);                 // Send TEXT to USART
      do                                      // Get first number
      {
        if(Uart1_Data_Ready())                // If a character ready
        {
          MyKey = Uart1_Read();               // Get a character
          if(MyKey == Enter)break;            // If ENTER key
          Uart1_Write(MyKey);                 // Echo the character
          Key = MyKey – '0';
          Op1 = 10*Op1 + Key;                 // First number in Op1
        }
      }while(1);

      Newline();

//
// Get the second character
//
      Uart1_Write_Text(msg3);                 // Send TEXT to USART
      do                                      // Get second number
      {
        if(Uart1_Data_Ready())
        {
          MyKey = Uart1_Read();               // Get a character
          if(Mykey == Enter)break;            // If ENTER key
          Uart1_Write(MyKey);                 // Echo the character
          Key = MyKey – '0';
          Op2 = 10*Op2 + Key;                 // Second number in Op2
        }
       }while(1);
```

Figure 6.88
cont'd

```
      Newline();
//
// Get the operation
//
      Uart1_Write_Text(msg4);
      do
      {
        if(Uart1_Data_Ready())
        {
          MyKey = Uart1_Read();                    // Get a character
          if(MyKey == Enter)break;                 // If ENTER key
          Uart1_Write(MyKey);                      // Echo the character
          Key = MyKey;
        }
      }while(1);

//
// Perform the operation
//
     Newline();
     switch(Key)                                   // Calculate
     {
      case Plus:
           Calc = Op1 + Op2;                       // If ADD
           break;
      case Minus:
           Calc = Op1 – Op2;                       // If Subtract
           break;
      case Multiply:
           Calc = Op1 * Op2;                       // If Multiply
           break;
      case Divide:
           Calc = Op1/Op2;                         // If Divide
           break;
     }

     LongToStr(Calc, kbd);                         // Convert to string
     op = Ltrim(kbd);
     Uart1_Write_Text(msg5);
     Uart1_Write_Text(op);                         // Display result

   }
}
```

Figure 6.88
cont'd

The GPS system consists of 24 satellites orbiting the Earth in six orbital planes at an altitude of 20,000 km. The orbital period is about 12 h, so each satellite orbits the earth twice a day. The orbits are organized such that normally six satellites are visible from any point on the Earth. To calculate the position, at least three satellites are required. The height can also be calculated if the fourth satellite is available. There are many books on the theory and operation of the GPS system. Also, there is a vast amount of information on the Internet on this topic.

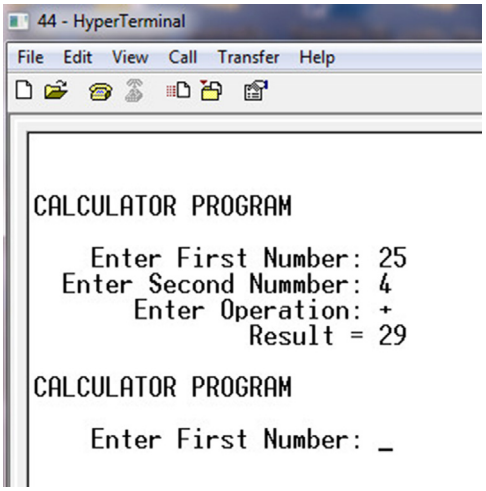


Figure 6.89: Sample Run of the Program.

GPS receivers are small handheld devices that receive information from the satellites and report the position (latitude and longitude) as well as the height of the user on a graphical display. GPS receivers usually provide ASCII output data to indicate the position and time information. These data, called the NMEA sentences, consist of a number of text, each one separated by a newline. We can read and decode these sentences to get information about our current coordinates.

In this project, we will be using a small GPS receiver board with an antenna and display our longitude and the latitude on an LCD display.

The block diagram of the project is shown in Figure 6.90.

Project Hardware

The project is based on the GPS Click board, manufactured by mikroElektronika (www.mikroe.com). Click boards are small peripheral modules designed for the

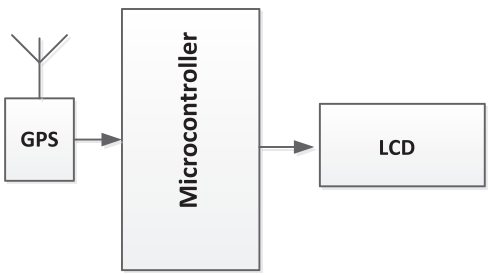


Figure 6.90: Block Diagram of the Project.

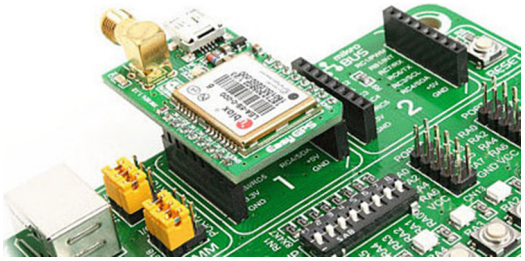


Figure 6.91: Connecting the GPS Click Board to EasyPIC V7 Development Board. (For color version of this figure, the reader is referred to the online version of this book.)

mikroBUS interface. There are Click boards available for many peripheral devices, such as LED, A/D converter, DAC, temperature sensor, GPS, relay, pressure sensor, light sensor, accelerator, WiFi, Ethernet, GSM, compass, and many more.

The mikroBUS is a 2 × 8-pin female header that provides interface signals, enabling many "Click Boards" to be connected to this bus. The mikroBUS provides interface signals for the following:

Analog pin,
Reset pin,
SPI Bus pins (CS, clock, and data I/O),
+3.3 V, +5 V, GND,
PWM output line,
Hardware interrupt pin,
UART (TX and RX) pins,
I^2C Bus pins (clock and data).

The EasyPIC V7 development board contains two identical mikroBUS connectors, enabling up to two Click boards to be connected directly to the board. Figure 6.91 shows the GPS Click board connected to one of the mikroBUS connectors on the EasyPIC V7 development board.

The GPS Click board uses the LEA-6S high performance GPS chip. The board can be interfaced with a microcontroller through the UART, I^2C bus, or through a USB connection. In this project, the UART interface is used for simplicity. An active or a passive antenna can be connected to the board to increase its sensitivity. Operation is with a 3.3 V power supply, and a power regulator should be used to provide 3.3 V if the board is used outside the EasyPIC V7 development board.

Figure 6.92 shows the circuit diagram of the project. In this diagram, only the UART interface to the GPS Click board is shown (see mikrolektronika web site for full circuit diagram of this Click board). The UART TX (RC6) and RX (RC7) pins of the microcontroller are connected directly to the GPS chip. PORTB is connected to an LCD as

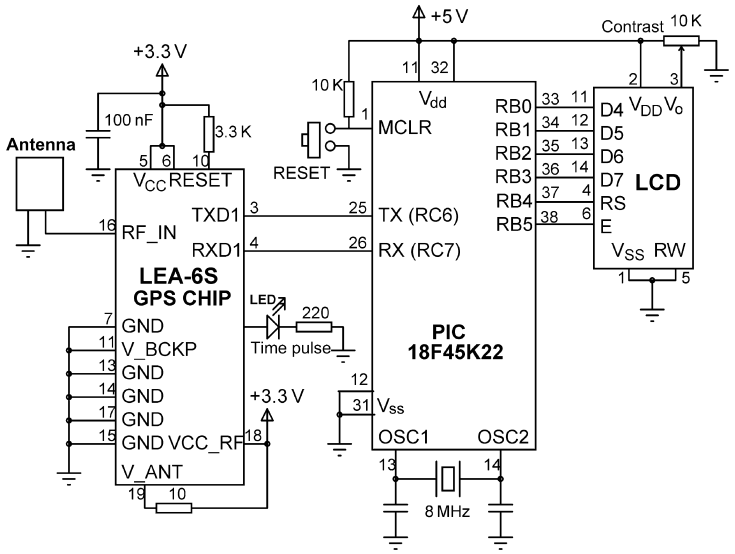


Figure 6.92: Circuit Diagram of the Project.

in the previous projects. An LED is connected to the Time_Pulse output of the GPS chip to see the device working. An antenna is connected to the RF_IN pin of the GPS chip.

Project PDL

The NMEA sentences generated by the GPS receivers start with the "$" character, followed by the name of the sentence and then its parameters. Each parameter is separated by a comma. A checksum byte is used at the end.

A sample of the NMEA sentences generated by the LEA-6 GPS every second are given below (different GPS chips may generate different sentences):

```
$GPRMC,101241.00,A,5127.36070,N,0003.12726,,0.118,,28813,,,A*7F
$GPVTG,,T,M,0.118,N0.218,K,A*0
$GPGGA,01241.00,527.36070,N00003.1272,E,1,06,1.0,40.8,M,4.4,M,,*63
$GPGSA,A,329,02,31,1,25,14,,,,,,2.57,1.6,2.01*0C
$GPGSV,,2,09,24,0,142,,25,7,072,21,2967,189,34,1,52,298,3*79
$GPGLL,517.35744,N,0003.13373E,103109.00,A,A*65
$GPRMC,101242.00,A,5127.36058,N,0003.12714,E0.326,,2806,1.6,40.8,M,454,M,,*6C
```

The above sentence list was obtained on a PC screen by the following program code after the microcontroller was connected to a PC and by running the terminal emulator software on the PC. See Project 6.11:

```
Uart1_Init(9600);
for(;;)
```

```
BEGIN
    Define connection between LCD and microcontroller
    Configure PORTB and PORTC as digital output
    Initialize LCD
    Initialize UART to 9600 Baud
    DO FOREVER
        Read until string "$GPGLL" is detected
        Read until character "*" is detected
        Extract latitude
        Extract longitude
        Display latitude and longitude
        Wait 2 s
    ENDDO
END
```

Figure 6.93: Project PDL.

```
{
   if(Uart1_Data_Ready() == 1)
   {
      c = Uart1_Read();
      Uart1_Write(c);
   }
}
```

The coordinates of the current position can be obtained from the $GPGLL sentence. This sentence decodes as follows:

$GPGLL,517.35744,N,0003.13373,E,103109.00,A,A*65

5127.35744,N	Latitude 51 deg. 27.35744 min. North
00003.1276,E	Longitude 000 deg. 03.1276 min. East
10124100	Fix taken at 10:31:09 UTC
A	Data valid (V = data invalid)
A	Mode (A = autonomous, D = differential)
*65	Checksum

The PDL of the program is given in Figure 6.93. The program initially searches for characters $GPGLL, and then the remainder of the text is read until the newline character is obtained. The latitude and longitude are then extracted and displayed on the LCD.

Project Program

microC Pro for PIC

The mikroC Pro for the PIC program listing is shown in Figure 6.94 (MIKROC-GPS.C). At the beginning of the program, the connection between the LCD and the microcontroller is defined, and PORTB and PORTC are configured as digital. The LCD is then initialized, and message INVALID DATA is displayed to start with. Then, the UART is initialized, and the remainder of the program is executed in an endless loop.

```
/*************************************************************************
                                GPS WITH LCD OUTPUT
                                ===================

In this project a GPS is connected to a PIC18F45K22 microcontroller. The coordinates (latitude
and longitude) of the current location are read and displayed on an LCD.

The GPS Click Board (www.mikroe.com) is used in this project together with the EasyPIC V7
development board. The GPS board is connected to one of the mikroBUS connectors on the
development board.

The LCD is connected to PORTB of the microcontroller as in the previous projects.

The communications between the GPS and the microcontroller is by using the serial port.
TX pin (RC6) and the RX pin (RC7) of the microcontroller are connected to the corresponding
 serial pins of the GPS.

The latitude and longitude are determined by decoding the NMEA sentence $GPGLL.  An example
is given below:

     $GPGLL,5127.35744,N,00003.13373,E,103109.00,A,A*65

     5127.35744,N       Latitude 51 deg. 27.35744 min. North
     00003.1276,E       Longitude 000 deg. 03.1276 min. East
     103109.00          Fix taken at 10:31:09 UTC
     A                  Data valid
     A                  Data autonomous
     *65                Checksum

The program assumes fixed length NMEA sentence.

Author:   Dogan Ibrahim
Date:     September 2013
File:     MIKRO-GPS.C
*************************************************************************/
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections
```

Figure 6.94: mikroC Pro for PIC Program.

```
void main()
{
   unsigned char buffer[50];
   unsigned char i,flag,c;
   unsigned char Lat[13], Lon[13];
   unsigned char gps[]="$GPGLL,";

   ANSELB = 0;
   ANSELC = 0;

   LCD_Init();                                  // Initialize LCD
   Lcd_Cmd(_LCD_CURSOR_OFF);                    // Disable cursor
   Lcd_Out(1,1,"INVALID DATA");                 // Display INVALID DATA to start with

   Uart1_Init(9600);                            // Baud = 9600

   for(;;)                                      // Endless loop
   {
     for(i = 0; i < 50; i++)buffer[i] = 0;      // Clear the buffer

     i = 0;
     flag = 0;
//
// Read until "$GPGLL," is detected
//
     while(flag == 0)
     {
       if(Uart1_Data_Ready() == 1)
       {
        c = Uart1_Read();
        if(c == gps[i])
        {
          i++;
          if(i == 7)flag=1;
        }
        else i = 0;
       }
     }
//
// We come to this point when the string "$GPGLL," has been detected
//
     Uart1_Read_Text(buffer,"*",255);           // Read until "*" detected
     if(buffer[37] == 'A')                      // If the sentence is valid
     {                                          // Get latitude Degrees
       Lat[0] = buffer[0];
       Lat[1] = buffer[1];
       Lat[2] = 178;                            // Degree character
       Lat[3] = ' ';
       Lat[4] = ' ';
       for(i = 0; i < 6; i++)Lat[5+i] = buffer[2+i];  // Get latitude minutes
       Lat[11] = buffer[11];                    // Get latitude direction
       Lat[12] = 0x0;                           // Terminate the string
```

Figure 6.94
cont'd

```
        Lon[0] = buffer[13];                              // Get longitude Degrees
        Lon[1] = buffer[14];
        Lon[2] = buffer[15];
        Lon[3] = 178;                                     // Degree character
        Lon[4] = ' ';
        for(i = 0; i < 6; i++)Lon[5+i] = buffer[16+i];    // Get longitude minutes
        Lon[11] = buffer[25];                             // Get longitude direction
        Lon[12] = 0x0;                                    // Terminate the string
        Lcd_Cmd(_LCD_CLEAR);                              // Clear LCD
        Lcd_Out(1,1,"LAT=");                              // Display LAT=
        Lcd_Out_Cp(Lat);                                  // Display the latitude
        Lcd_Out(2,1,"LON=");                              // Display LON=
        Lcd_Out_Cp(Lon);                                  // Display the longitude
      }
      else                                                // If invalid data
      {
       Lcd_Cmd(_LCD_CLEAR);                               // Clear LCD
       Lcd_Out(1,1,"INVALID DATA");                       // Display INVALID DATA
      }
      Delay_Ms(2000);                                     // Wait 2 s
    }
}
```

Figure 6.94
cont'd

At the beginning of the loop, the buffer that will hold the GPS data is cleared. The program then reads data from the GPS and looks for the string "$GPGLL,". When this string is detected, the Uart1_Read_Text function is used to read data from the GPS until the delimiting character "*" is detected (this character is at the end of the $GPGLL sentence just before the checksum). The buffer at this point contains all the parameters of the sentence $GPGLL, starting from the latitude parameter. The remainder of the program extracts the latitude and longitude parameters and loads into two string arrays called LAT and LON, respectively. These arrays are then displayed on the LCD after adding the degree sign and spaces at appropriate points.

Note that the code to detect the NMEA sentence "GPGLL," could have been done using the following two lines. Although the code seems to be smaller, it requires a very large buffer size (e.g. $\geq$1000 characters) since all the generated NMEA sentences will be read until the "$GPGLL," is detected as the delimiter:

```
Uart1_Read_Text(buffer, "$GPGLL,", 255);   // Read until "$GPGLL," detected
Uart1_Read_Text(buffer, "*",255);          // Read until "*" detected
```

The program checks the NMEA sentence validity every second, and if the sentence is not valid, the message INVALID DATA is displayed. The program assumes that the width of the parameters in the "$GPGLL" sentence have fixed sizes. Although this is the case with

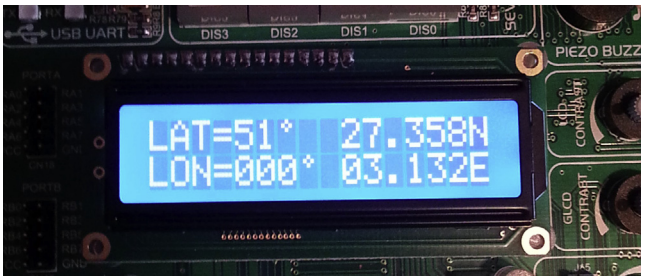


Figure 6.95: Sample Display. (For color version of this figure, the reader is referred to the online version of this book.)

most GPS receivers, it may be necessary to locate the commas and then extract the parameters if this is not the case.

A sample display is shown in Figure 6.95.

Project 6.13—ON−OFF Temperature Control

Project Description

This project is about designing an ON−OFF type temperature control system for a small plant. Figure 6.96 shows the block diagram of the system to be designed.

The desired temperature setting is entered using a keypad. The temperature of the plant is measured using an analog temperature sensor. The microcontroller reads the temperature

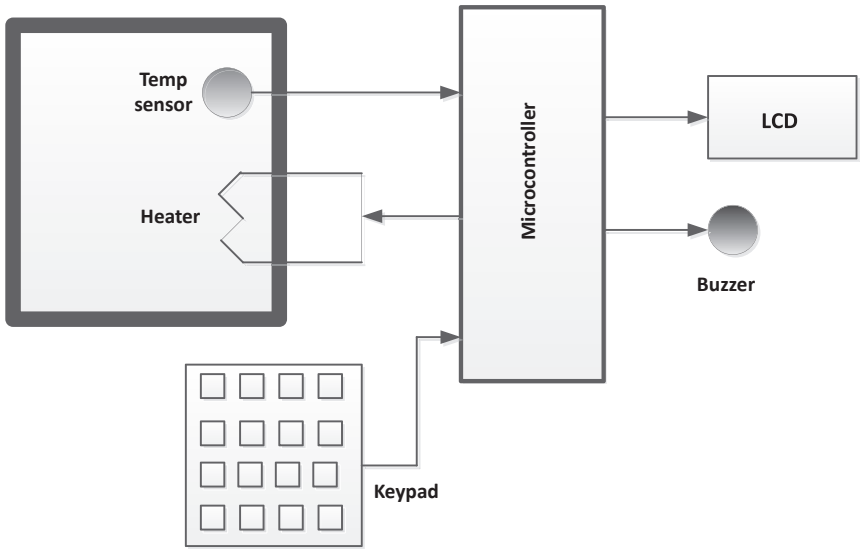


Figure 6.96: Block Diagram of the System.

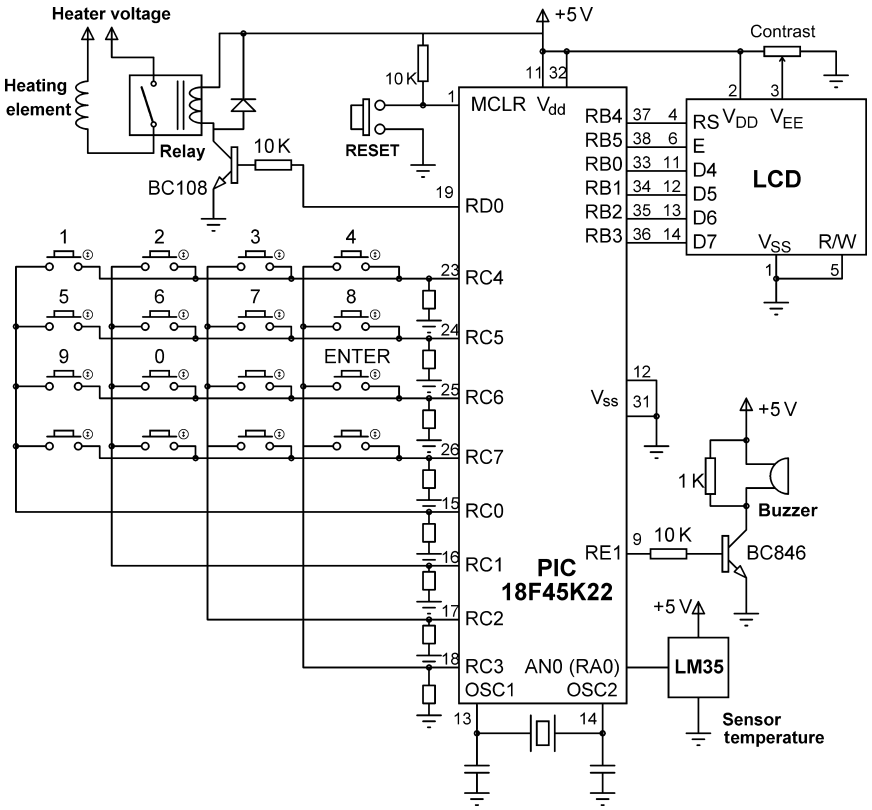


Figure 6.97: Circuit Diagram of the Project.

every 5 s and compares it with the desired value. If the desired value is higher than the measured value, then the heater is turned ON. If on the other hand the measured value is higher than the desired value, then the heater is turned OFF. An LCD display shows the measured temperature continuously. If the absolute difference between the desired value and the measured value is $>2\ ^{\circ}C$, then the buzzer sounds every 5 s as an alarm and remains ON as long as the temperature is high or low. The buzzer will turn OFF when the difference between the desired value and the actual value is $<2\ ^{\circ}C$.

Project Hardware

Figure 6.97 shows the circuit diagram of the project. The LCD is connected to PORTC as in the previous projects. An LM35 analog temperature sensor chip is connected to the analog input pin AN0 (RA0). A 4×4 keypad is connected to PORTC. The buzzer is connected to pin RE1. The heater is controlled using a transistor and a relay connected to pin RD0 of the microcontroller.

```
BEGIN
    Define LCD  to microcontroller connections
    Define Keypad port
    Assign symbols BUZZER, HEATER, ON, OFF to port pins
    Configure PORTB, PORTC, PORTD, PORTE as digital
    Configure RA0 as input, RD0, RE1 and RC0:RC3 as output
    Initialize Keypad and Sound libraries
    Display heading "ON–OFF CONTROL"
    Wait 2 s
    Read desired temperature
    Wait until ENTER is pressed
    DO FOREVER
        Read plant temperature from channel 0
        Convert temperature to Degrees Centigrate
        Display the Set Point and actual pant temperatures
        IF Set Point > Plant temperature
            Turn HEATER ON
            IF Set Point – Plant temperature > 2
                Sound alarm
            ENDIF
        ELSE
            Turn HEATER OFF
            IF Set Point – Plant temperature > 2
                Sound alarm
            ENDIF
        ENDIF
        Wait 5 s
    ENDDO
END
```

Figure 6.98: Project PDL.

Project PDL

The project PDL is shown in Figure 6.98.

Project Program

mikroC Pro for PIC

The mikroC Pro for the PIC program listing is shown in Figure 6.99 (MIKROC-ON-OFF.C). At the beginning of the program, the connections between the LCD and the microcontroller are defined, the keypad port is defined, BUZZER, HEATER, and ON and OFF are defined and assigned as symbols to port pins. PORTB, PORTC, PORTD, and PORTE are configured as digital. RA0 is configured as an input pin. RD0, RE1, PORTB, and half of PORTC are configured as output pins. The program then initializes the Keypad and the Sound libraries. The LCD is cleared and message "ON−OFF CONTROL" is displayed for 2 s.

```
/*************************************************************************
                        ON-OFF TEMPERATURE CONTROL
                        ==========================

In this project the temperature of a plant is controlled using simple ON–OFF type controller.

The plant temperature is measured using an LM35DZ type analog temperature sensor. An
LCD helps to enter the set point temperature, and also displays the set point as well as the
actual plant temperature in real-time.

 A 4×4 keypad is used to set the desired temperature. If the set temperature is below the
actual plant temperature then the heater relay is turned ON. If on the other hand the set
temperature is above the actual plant temperature then the relay is turned OFF. A buzzer
sounds if the absolute difference between the desired and the actual temperatures is more
than 2 oC

        The LCD is connected to PORTB
        The buzzer is connected to pin RE1 through a transistor switch
        The keypad is connected to PORTC
        The heater relay is connected to pin RD0 through a transistor switch
        The LM35DZ temperature sensor is connected to pin AN0 (RA0)

        The program uses the built-in keypad library functions.

The control action is taken every 5 s

Author:   Dogan Ibrahim
Date:     September 2013
File:     MIKROC-ON-OFF.C
*************************************************************************/
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

// Keypad module connections
char keypadPort at PORTC;
// End of keypad module connections

#define BUZZER PORTE.RE1
```

Figure 6.99: mikroC Pro for PIC Program.

```
#define HEATER PORTD.RD0
#define Enter 12
#define ON 1
#define OFF 0
//
// Start of MAIN program
//
void main()
{
   unsigned char MyKey,Txt[14];
   unsigned int SetPoint;
   unsigned char *op;
   unsigned int temp;
   float mV, PlantTemp;

   ANSELB = 0;                                  // Configure PORTB as digital
   ANSELC = 0;                                  // Configure PORTC as digital
   ANSELD = 0;                                  // Configure PORTD as digital
   ANSELE = 0;                                  // Configure PORTE as digital
   TRISA0_bit = 1;                              // Configure AN0 (RA0) as input
   TRISB = 0;                                   // PORTB are outputs (LCD)
   TRISC = 0xF0;                                // RC4–RC7 are inputs
   TRISD0_bit = 0;                              // RD0 is output
   TRISE1_bit = 0;                              // RE1 is output

   Keypad_Init();                               // Initialize keypad library
   Sound_Init(&PORTE,1);                        // Initialize Sound library
//
// Configure LCD
//
    Lcd_Init();                                 // Initialize LCD
    Lcd_Cmd(_LCD_CLEAR);
    Lcd_Out(1,1,"ON–OFF CONTROL");              // Display CALCULATOR
    Delay_ms(2000);                             // Wait 2 s
    Lcd_Cmd(_LCD_CLEAR);                        // Clear display

    BUZZER = OFF;                               // TURN OFF buzzer to start with
    HEATER = OFF;                               // Turn OFF heater to start with
//
// On startup read the set point temperature from the keypad
//
    Lcd_Out(1,1,"Enter Set Point");
    setPoint = 0;

    Lcd_Out(2,1,"SP: ");
    while(1)                                    // Get first no
    {
      do
        MyKey = Keypad_Key_Click();
      while(!MyKey);
      if(MyKey == Enter)break;                  // If ENTER pressed
```

Figure 6.99
cont'd

```
      if(MyKey == 10)MyKey = 0;                    // If 0 key pressed
      Lcd_Chr_Cp(MyKey + '0');
      SetPoint = 10*SetPoint + MyKey;              // First number in Op1
     }

     Lcd_cmd(_LCD_CLEAR);                          // Clear LCD
     Lcd_Out(1,1,"SP = ");                         // Display SP=
     IntToStr(SetPoint,Txt);                       // Convert to string
     op = Ltrim(Txt);                              // Remove leading spaces
     Lcd_Out_CP(op);
     Lcd_Out(2,1,"ENTER to cont.");
//
// Wait until ENTER is pressed
//
     MyKey = 0;
     while(MyKey != Enter)
     {
      do
         MyKey = Keypad_Key_Click();
      while(!MyKey);
     }


// Program loop
//
   for(;;)                                         // Endless loop
   {
//
// Display the SetPoint and the Actual temperatures
//
     temp = ADC_Read(0);                           // Read from AN0 (RA0)
     mV = temp*5000.0/1024.0;                      // Convert to mV
     PlantTemp = mV/10.0;                          // Convert to degrees C
     Lcd_Cmd(_LCD_CLEAR);
     IntToStr(SetPoint,Txt);                       // Convert to string
     op = Ltrim(Txt);
     Lcd_Out(1,1,"SP=");                           // Display SP=
     Lcd_Out_Cp(op);                               // Display Set Point
     Lcd_Chr_Cp(178);
     Lcd_Chr_Cp('C');                              // Display C
     Lcd_Out(2,1,"AC=");                           // Display AC=
     FloatToStr(PlantTemp, Txt);                   // Convert to string
     Lcd_Out_CP(Txt);
     Lcd_Chr_Cp(178);                              // Display Degree sign
     Lcd_Chr_Cp('C');                              // Display C character
//
// Implement the ON–OFF controller algorithm
//
     if(SetPoint > PlantTemp)                      // If SetPoint is bigger than actual
     {
       HEATER = ON;                                // Turn ON heater
```

Figure 6.99
cont'd

```
            if((SetPoint – PlantTemp) > 2.0)          // If outside range
              Sound_Play(1000,1000);                  // Turn ON BUZZER
          }
          else                                        // If Set Point is not bigger than actual
          {
            HEATER = OFF;                             // Turn OFF heater
            if((PlantTemp – SetPoint) > 2.0)          // Actual temp is bigger than setPoint
              Sound_Play(1000,1000);                  // Turn ON BUZZER
          }
          Delay_Ms(5000);                             // Wait 5 s and repeat
        }
    }
```

Figure 6.99
cont'd

The program then reads the desired temperature setting (Set Point temperature) from the keypad after displaying the following message and waiting for the user to enter the desired temperature:

Enter Set Point
SP:

After reading the desired temperature, the program displays a message to tell the user to press the ENTER key to continue. The remainder of the program is executed inside an endless loop that is repeated every 5 s.

Inside this loop, the program reads the plant temperature from analog channel 0 (AN0, or RA0) of the microcontroller and stores in the floating point variable PlantTemp in degrees centigrade. The Set Point and the actual plant temperature are then displayed in the following format (assuming the Set Point temperature is 25 °C, and the actual plant temperature is 20.45189 °C):

SP = 25 °C
AC = 20.45189 °C

The next part of the program implements the ON−OFF control algorithm. If the Set Point temperature is greater than the measured temperature, then the plant temperature is low, and HEATER is turned ON. If also the difference between the Set Point and the measured temperatures is >2 °C, then the BUZZER is sounded as an alarm. If the Set Point temperature is less than (or equal) to the measured temperature, then the HEATER is turned OFF. At the same time, if the difference between the Set Point and the measured values are >2 °C, then the BUZZER is sounded. The above process gets repeated every 5 s.

The program given in Figure 6.99 can be improved by the following modifications:

- The Set Point temperature can be stored in the electrically erasable programmable read-only memory of the microcontroller so that it does not need to be entered every time.

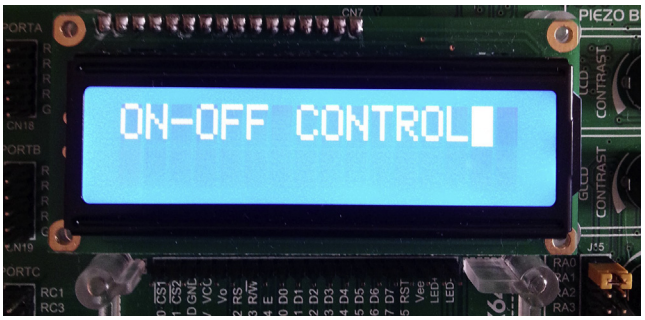


Figure 6.100: Sample Display 1. (For color version of this figure, the reader is referred to the online version of this book.)

Figure 6.101: Sample Display 2. (For color version of this figure, the reader is referred to the online version of this book.)

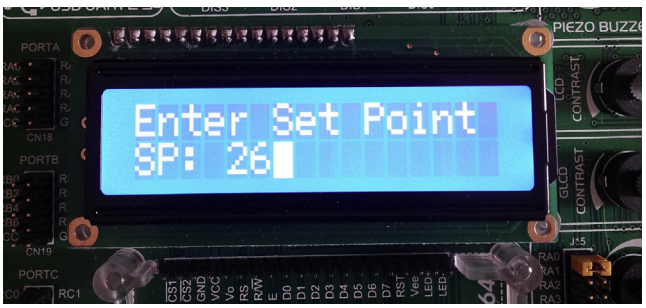


Figure 6.102: Sample Display 3. (For color version of this figure, the reader is referred to the online version of this book.)

Figure 6.103: Sample Display 4. (For color version of this figure, the reader is referred to the online version of this book.)

Figure 6.104: Sample Display 5. (For color version of this figure, the reader is referred to the online version of this book.)

- Instead of the simple ON−OFF, a more powerful control algorithm can be used (e.g. PID).
- Currently, the Set Point temperature must be an integer number. The keypad entry routine can be modified to accept floating point Set Point temperatures.

The program given in Figure 6.99 exceeds the free 2 K limit of the compiler, and users must have a licensed copy of the compiler to compile the program.

Figures 6.100−6.104 show sample displays from the project.