

# *Advanced PIC18 Projects*

## **Chapter Outline**

### **Project 7.1—Bluetooth Serial Communication—Slave Mode 333**

RN41 Bluetooth Module 334

Project Hardware 336

SA, <value> 339

SM, <value> 340

SN, <value> 340

SO, <value> 340

SP, <string> 340

SU, <value> 340

D 340

C, <address> 341

K 341

R,1 341

I, <value1>, <value2> 341

SR, <hex value> 341

Project PDL 346

Project Program 346

mikroC Pro for PIC 346

MPLAB XC8 355

### **Project 7.2—Bluetooth Serial Communication—Master Mode 361**

Project Hardware 361

Project PDL 361

Project Program 363

mikroC Pro for PIC 363

### **Project 7.3—Using the RFID 369**

Radiofrequency Identification 369

Project Hardware 371

CR95HF Operational Modes 374

CR95HF Startup Sequence 374

UART Communication 374

Passive RFID Tags 375

Project PDL 377

Project Program 379

mikroC Pro for PIC 379

MPLAB XC8 387

### **Project 7.4—RFID Lock 387**

Project Hardware 388

Project PDL 389

Project Program 389

*mikroC Pro for PIC* 389

**Project 7.5—Complex SPI Bus Project 389**

The Master Synchronous Serial Port Module 389

MSSP in the SPI Mode 396

SPI Mode Registers 397

*SSPxSTAT* 398

*SSPxCON1* 398

Operation in the SPI Mode 398

Sending Data to the Slave 400

Receiving Data From the Slave 400

Configuration of MSSP for the SPI Master Mode 400

Data Clock Rate 400

Clock Edge Mode 401

Enabling the SPI Mode 401

TC72 Temperature Sensor 402

TC72 Read/Write Operations 404

Internal Registers of the TC72 405

Control Register 405

LSB and MSB Registers 405

Manufacturer ID 405

Project Hardware 406

The Program 407

*MPLAB XC8* 407

Displaying Negative Temperatures 413

Displaying the Fractional Part 414

**Project 7.6—Real-Time Clock Using an RTC Chip 414**

Project Hardware 425

Project PDL 427

Project Program 427

*mikroC Pro for PIC* 427

*MPLAB XC8* 435

**Project 7.7—Real-Time Alarm Clock 436**

Project Hardware 438

Project PDL 438

Project Program 439

*mikroC Pro for PIC* 439

**Project 7.8—SD Card Projects—Write Text To a File 439**

Operation of the SD Card in the SPI Mode 449

Reading Data 451

Writing Data 451

Project Description 452

Project Hardware 452

Project PDL 453

---

Project Program	453
<i>mikroC Pro for PIC</i>	453
<i>MPLAB XC8</i>	457
Setting the Configuration Files	462
MPLAB XC8 MDD Library Functions	463
Library Options	464
Microcontroller Memory Usage	464
Sequence of Function Calls	465
Reading From an Existing File	465
Writing to an Existing File	465
Deleting an Existing File	466

**Project 7.9—SD Card-Based Temperature Data Logger 466**

Hardware Description	468
Project Program	469
<i>mikeoC Pro for PIC</i>	469
<i>MPLAB XC8</i>	476

**Project 7.10—Using Graphics LCD—Displaying Various Shapes 477**

The 128×64 Pixel GLCD	478
Operation of the GLCD	480
mikroC Pro for PIC GLCD Library Functions	481
<i>Glcd_Init</i>	481
<i>Glcd_Set_Side</i>	482
<i>Glcd_Set_X</i>	482
<i>Glcd_Set_Page</i>	482
<i>Glcd_Write_Data</i>	482
<i>Glcd_Fill</i>	482
<i>Glcd_Dot</i>	482
<i>Glcd_Line</i>	483
<i>Glcd_V_Line</i>	483
<i>Glcd_H_Line</i>	483
<i>Glcd_Rectangle</i>	483
<i>Glcd_Rectangle_Round_Edges</i>	484
<i>Glcd_Rectangle_Round_Edges_Fill</i>	484
<i>Glcd_Box</i>	484
<i>Glcd_Circle</i>	485
<i>Glcd_Circle_Fill</i>	485
<i>Glcd_Set_Font</i>	485
<i>Glcd_Set_Font_Adv</i>	486
<i>Glcd_Write_Char</i>	486
<i>Glcd_Write_Char_Adv</i>	486
<i>Glcd_Write_Text</i>	486
<i>Glcd_Write_Text_Adv</i>	487
<i>Glcd_Write_Const_Text_Adv</i>	487
<i>Glcd_Image</i>	487

Project Hardware 487

Project Program 487

*mikroC Pro for PIC* 487

**Project 7.11—Barometer, Thermometer and Altimeter Display on a GLCD 490**

Project Hardware 493

Project PDL 494

Project Program 494

*mikroC Pro for PIC* 494

**Project 7.12—Plotting the Temperature Variation on the GLCD 501**

Project Description 501

Block Diagram 501

Circuit Diagram 501

Project PDL 502

Project Program 502

*mikroC Pro for PIC* 502

**Project 7.13—Using the Ethernet—Web Browser-Based Control 508**

Ethernet Connectivity 509

Embedded Ethernet Controller Chips 510

Embedded Ethernet Access Methods 510

*Using a Web Browser on the PC* 511

*Using a HyperTerminal* 511

*Embedded System Sending E-mail* 512

*Using Custom Application* 512

*Example Ethernet-Based Embedded Control Project* 513

Project Hardware 513

The Construction 515

Project PDL 516

Project Software 516

*mikroC Pro for PIC* 516

**Project 7.14—Using the Ethernet—UDP-Based Control 521**

The Hardware 521

The PC Program 521

The Microcontroller Program 522

*mikroC Pro for PIC* 522

**Project 7.15—Digital Signal Processing—Low Pass FIR Digital Filter Project 522**

The Filter Structure 525

The Hardware 528

Project PDL 528

Project Program 529

*mikroC Pro for PIC* 529

**Project 7.16—Automotive Project—Local Interconnect Network Bus Project 535**

The LIN Protocol 536

Project Description 538

Project Hardware 538

Project PDL 541

Project Program 541

---

<i>mikroC Pro for PIC</i>	541
<i>Get_Response Function</i>	546
<b>Project 7.17—Automotive Project—Can Bus Project</b>	<b>550</b>
Data Frame	553
<i>Start of Frame</i>	554
<i>Arbitration Field</i>	554
<i>Control Field</i>	554
<i>Data Field</i>	554
<i>CRC Field</i>	555
<i>ACK Field</i>	555
Remote Frame	555
Error Frame	555
Overload Frame	555
Bit Stuffing	556
Nominal Bit Timing	556
PIC Microcontroller CAN Interface	558
PIC18F258 Microcontroller	559
Configuration Mode	561
Disable Mode	561
Normal Operation Mode	561
Listen-only Mode	561
Loop-back Mode	561
Error Recognition Mode	562
CAN Message Transmission	562
CAN Message Reception	562
Calculating the Timing Parameters	563
mikroC Pro for PIC CAN Functions	564
CAN Bus Programming	568
CAN Bus Project Description—Temperature Sensor and Display	568
The COLLECTOR Processor	569
The DISPLAY Processor	569
DISPLAY Program	570
COLLECTOR Program	574
<b>Project 7.18 Multitasking</b>	<b>578</b>
Cooperative Scheduling	579
Round-robin Scheduling	579
Preemptive Scheduling	580
<b>Project 1—Using Cooperative Multitasking</b>	<b>581</b>
Project Hardware	582
Project PDL	582
Project Program	582
<i>mikroC Pro for PIC</i>	582
<b>Project 2—Using Round-Robin Multitasking With Variable CPU Time Allocation</b>	<b>590</b>
Project Description	591
Project Hardware	591

Project Program 594

*mikroC Pro for PIC* 594

**Project 7.19—Stepper Motor Control Projects—Simple Unipolar Motor Drive 598**

Unipolar Stepper Motors 598

*One-phase Full-step Sequencing* 598

*Two-phase Full-step Sequencing* 598

*Two-phase Half-step Sequencing* 599

Bipolar Stepper Motors 600

Project Description 600

Project Hardware 601

Project PDL 602

Project Program 602

*mikroC Pro for PIC* 602

**Project 7.20—Stepper Motor Control Projects—Complex Control Of A Unipolar Motor 604**

Project Hardware 604

Project Program 604

*mikroC Pro for PIC* 604

**Project 7.21—Stepper Motor Control Project—Simple Bipolar Motor Drive 608**

Project Description 608

Project Hardware 609

Project Program 611

*mikroC Pro for PIC* 611

MPLAB XC8 611

**Project 7.22—DC Motor Control Projects—Simple Motor Drive 613**

Project Description 616

Project Hardware 617

Project Program 618

*mikro Pro for PIC* 618

**Project 7.23—A Homemade Optical Encoder For Motor Speed Measurement 619**

Project Hardware 620

Project Program 621

*mikroC Pro for PIC* 621

**Project 7.24—Closed-Loop DC Motor Speed Control—On/Off Control 624**

Project Hardware 626

Project Program 626

*mikroC Pro for PIC* 626

In this chapter, we will be developing advanced projects using various peripheral devices and protocols such as Bluetooth, radiofrequency identification (RFID), WiFi, Ethernet, controller area network (CAN) bus, secure digital (SD) cards, universal serial bus (USB), and motor control. As in the previous chapter, the project description, hardware design, program description language (PDL), full program listing, and description of the program for each project will be given in detail.

## Project 7.1—Bluetooth Serial Communication—Slave Mode

In this project, we shall be using a Bluetooth module to communicate with a personal computer (PC). The program will receive a text message from the PC and will display it on an liquid crystal display (LCD). In this project, our Bluetooth device is used as a slave device, and the PC is used as a master device.

Before going into the details of the project, it is worthwhile to review how Bluetooth devices operate.

The bluetooth is a form of digital communication standard for exchanging data over short distances using short-wavelength radiowaves in the industrial, scientific and medical (ISM) band from 2.402 to 2.489 GHz. The Bluetooth was originally conceived in 1994 as an alternative to the RS232 serial communications. Bluetooth communication occurs in the form of packets where the transmitted data are divided into packets, and each packet is transmitted using one of the designated Bluetooth channels. There are 79 channels, each with a 1-MHz bandwidth, starting from 2.402 GHz. The channels are hopped 1600 times per second using an adaptive frequency hopping algorithm. Because the communication is based on radiofrequency (RF), the devices do not have to be in the line of sight of each other in order to communicate.

Each Bluetooth device has a Media Access Control (MAC) address where communicating devices can recognize and establish a link if required.

Bluetooth communication operates in a master—slave structure, where one master can communicate with up to seven slaves. All the devices share the master's clock. Bluetooth devices in contact with each other form a *piconet*. At any time, data can be transferred between a master and a slave device. The master can choose which slave to communicate to. In the case of multiple slaves, the master switches from one slave to the next.

Bluetooth is a secure way to connect and exchange data between various devices such as mobile phones, laptops, PCs, printers, faxes, global positioning system (GPS) receivers, and digital cameras.

Bluetooth's main characteristics can be summarized as follows:

- There are two classes of Bluetooth standards. The communication range is up to 100 m for Class 1, up to 10 m for Class 2, and up to 1 m for Class 3 devices.
- Class 1 devices consume 100 mW of power, Class 2 devices consume 2.5 mW, and Class 3 devices consume only 1 mW.
- The data rate is up to 3 Mbps.

The effective communication range depends on many factors, such as the antenna size and configuration, battery condition, and attenuation from walls.

Further information about Bluetooth communication standards can be obtained from many books, from the Internet, and from the Bluetooth Special Interest Group.

### **RN41 Bluetooth Module**

In this project, the popular RN41 Bluetooth module, manufactured by Roving Networks, will be used. RN41 (Figure 7.1) is a Class 1 Bluetooth module delivering up to 3-Mbps data rate for distances up to 100 m. The module has been designed for easy interface to embedded systems. The basic features of the RN41 are as follows:

- Low-power operation (30 mA when connected, 250 µA in the sleep mode);
- Support for Universal Asynchronous Receiver Transmitter (UART) and USB data connection interfaces;
- On-board ceramic chip antenna;
- A 3.3-V operation;
- Baud rate from 1200 bps up to 921 kbps;
- A 128-bit encryption for secure communication;
- Error correction for guaranteed packet delivery.

RN41 is a 35-pin device. When operated using a UART interface, the following pins are of importance:

- 1: GND
- 3: GPIO6 (Set Bluetooth mode. 1 = auto master mode)
- 4: GPIO7 (Set Baud rate. 1 = 9600 bps, 0 = 115-kbps, or firmware setting)
- 5: Reset (Active low)
- 11: VDD (3.3-V supply)
- 12: GND
- 13: UART\_RX (UART receive input)
- 14: UART\_TX (UART transmit output)



**Figure 7.1: RN41 Bluetooth Module.**

- 15: UART\_RTS (UART RTS, goes high to disable host transmitter)
- 16: UART\_CTS (UART CTS, if set high, it disables transmitter)
- 20: GPIO3 (autodiscovery = 1)
- 28: GND
- 29: GND

In low-speed interfaces, RTS and CTS pins are not used. Pin 3 is set to 1 for the auto master mode, pin 4 is set to 1 for 9600 Baud, Pin 5 is set to 1 for normal operation, and pin 20 is set to 1 for autodiscovery, GND and V<sub>DD</sub> pins are connected to the ground and 3.3-V power supply lines. Thus, the module requires only two pins (pin 13 and pin 14) for interfacing to a microcontroller.

The PIC18F45K22 microcontroller operates with a +5-V power supply. The output logic high level from an I/O pin is minimum at +4.3 V. Similarly, the minimum input voltage to be recognized as logic high is +2.0 V.

The RN41 module operates with a +3.3-V power supply. The minimum output logic high level is V<sub>DD</sub> – 0.2 = +3.1 V. Similarly, the maximum input logic high level is V<sub>DD</sub> + 0.4 = +3.7 V.

The RN41 module cannot be connected directly to the PIC18F45K22 microcontroller. Although there is no problem with low logic levels, the output high voltage of +4.3 V of the microcontroller is much larger than the maximum allowable input high voltage of +3.7 V of RN41. Similarly, the minimum output high voltage of +3.1 V of the RN41 is just enough to provide the minimum high-level voltage required by the microcontroller inputs (minimum +2.0 V). In practice, +3.3- to +5.0-V and +5.0- to +3.3-V voltage converter chips are used in between the microcontroller and the RN41 module. A simple voltage converter circuit can be designed using a pair of transistors as switches to give both the required voltage level and also the correct logic polarity. Figures 7.2 and 7.3 show +5.0- to 3.3-V level converter and +3.3- to +5.0-V level converter circuits, respectively.

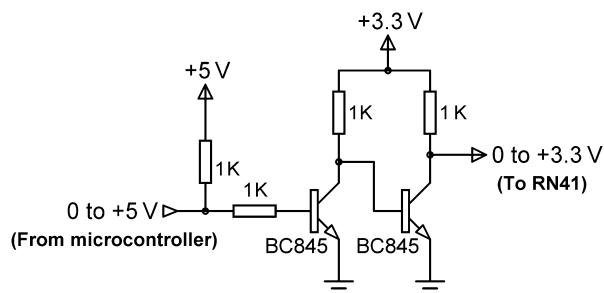


Figure 7.2: The +5- to +3.3-V Level Converter Circuit.

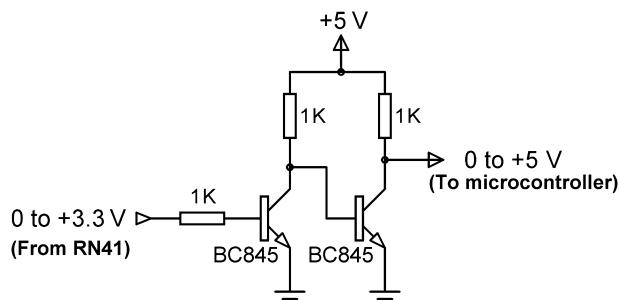


Figure 7.3: The +3.3- to +5.0-V Level Converter Circuit.

### Project Hardware

The project uses the EasyBluetooth board (HW Rev. 1.02) manufactured by mikroelektronika ([www.mikroe.com](http://www.mikroe.com)). This is a small plug-in board having the following specifications:

- RN41 Bluetooth module,
- Power select jumper (+5 or +3.3 V),
- A +5- to +3.3-V power regulator,
- Pull-up jumpers for pins GPIO3, GPIO4, GPIO6, and GPIO7,
- Voltage level converter transistor circuits,
- Dual in-line package switch for selecting signals for the microcontroller,
- IDC10 connector.

The RN41 Bluetooth module operates in two modes: data mode and command mode. The data mode is the default mode, and in this mode, when the module receives data, it simply strips the protocol headers and trailers and passes the raw data to the UART port.

Similarly, data to be sent out are framed by the addition of protocol headers and trailers and are passed to the UART for transmission. Thus, the process of data communication is transparent to the microcontroller.

The default configuration of the RN41 Bluetooth module is as follows:

- Bluetooth in the slave mode;
- Pin code: 1234;
- Serial port 115,200 (it is set to 9600 in this project by pulling-up pin GPIO7), eight data bits, no parity, one stop bit;
- No flow control.

The module can be configured after putting it into the command mode and sending appropriate ASCII characters. There are two ways to put the module into the command

mode: Local communication with the module via the UART port and via the Bluetooth link. The new configuration takes effect after a reboot of the module.

In this book, we shall see how to configure the Bluetooth module via its UART port. This process requires the module ideally to be connected to a PC via its UART port and the use of a terminal emulator program (e.g. Hyperterm or mikroC pro for a PIC built-in terminal emulator). Once the module is rebooted, commands must be sent within 60 s (this can be changed if required).

If you are using the EasyPIC V7 development board, the Bluetooth module can be configured via the USB UART module of the development board. The steps are as follows:

- Configure the EasyBluetooth board jumpers as given below:

DIL switch SW1: Set 1 to ON (this connects RN41 RX to UART TX)

DIL switch SW1: Set 4 to ON (this connects RN41 TX to UART RX)

DIL switch SW1: Set 7 to ON (this connects RN41 RESET to RC1)

Set J1 to 5 V (EasyPIC V7 operates at +5 V)

Set PI03 (GPIO3) to Pull-Up

Set PI07 (GPIO7) to Pull-Up (9600 baud)

- Configure the EasyPIC V7 development board as follows:

SET Jumper J3 to USB UART

Set Jumper J4 to USB UART

DIL switch SW1: Set 1 to ON (RC7 to RX)

DIL switch SW2: Set 1 to ON (RC6 to TX)

J17 to GND (pin becomes low when button pressed)

Insert a jumper at “Disable Protect” jumper position (near Button Press Level)

- When using the USB UART, the RX and TX pins should be reversed, and as a result, it is not possible to directly connect the EasyBluetooth board to PORTC of the EasyPIC V7 development board.

Connect an Insulation Displacement Connector (IDC) cable with two female connectors to PORTC and then make the following connections with wires between one end of the IDC cable and the EasyBluetooth board connector:

Connect RC6 to pin P7 of EasyBluetooth board

Connect RC7 to pin P6 of EasyBluetooth board

Connect RC1 to pin P1 of EasyBluetooth board

Connect VCC to pin VCC of EasyBluetooth

Connect GND to pin GND of EasyBluetooth board

- Connect the USB UART port of the EasyPIC V7 development board to the PC USB port.
- Start the terminal emulator program on the PC and select the serial port, 9600 baud, 8 bits, no parity, 1 stop bit, and no flow control.

- Enter characters “\$\$\$” on the terminal emulator window with no additional characters (e.g. no carriage return). The Bluetooth module should respond with characters “CMD”. If there is no response from the Bluetooth module, reset the module by pressing button RC1 on the EasyPIC V7 development board and send the “\$\$\$” characters again.

Figure 7.4 shows the response from the Bluetooth module, using the mikroC Pro for PIC terminal emulator. Note here that “Append New Line” box is disabled so that any other characters are not sent after the “\$\$\$”.

Now, enable the “Append New Line” by clicking the box and set the “New Line Setting” to CR (0x0D). Now, Enter character “h” to get a list of default configuration settings.

Figure 7.5 shows a part of the display.

When a valid command is entered, the module returns string “AOK”. If an invalid command is entered, the module returns “ERR”, and “?” character for unrecognized commands. Some of the useful commands are given below. Command “—“ exits from the command mode and returns to the data mode.

The various return codes have the following numeric values:

There are three types of commands: Set commands, Get commands, Change commands, Action commands, and General Purpose Input Output (GPIO) commands.

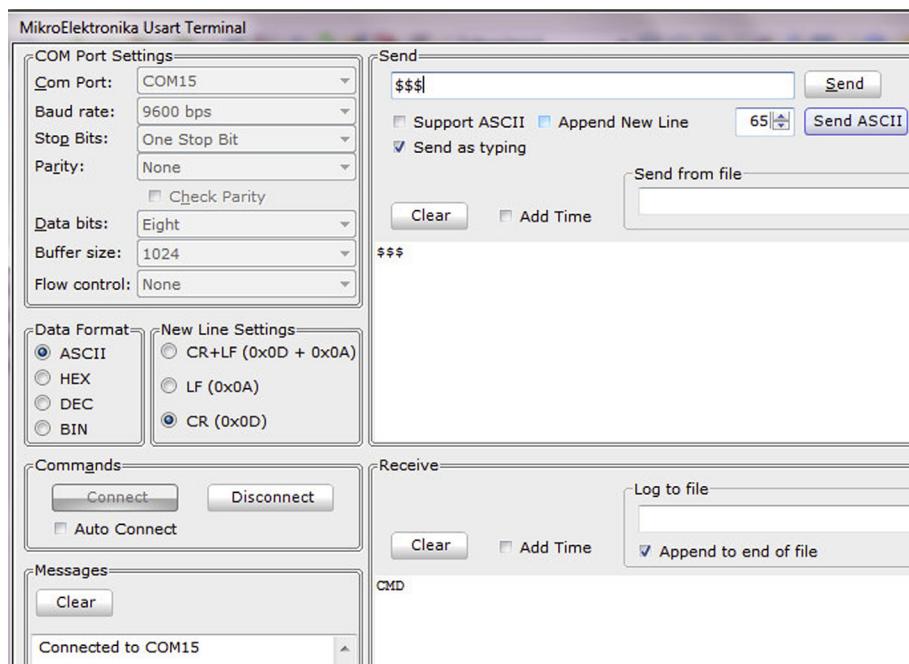
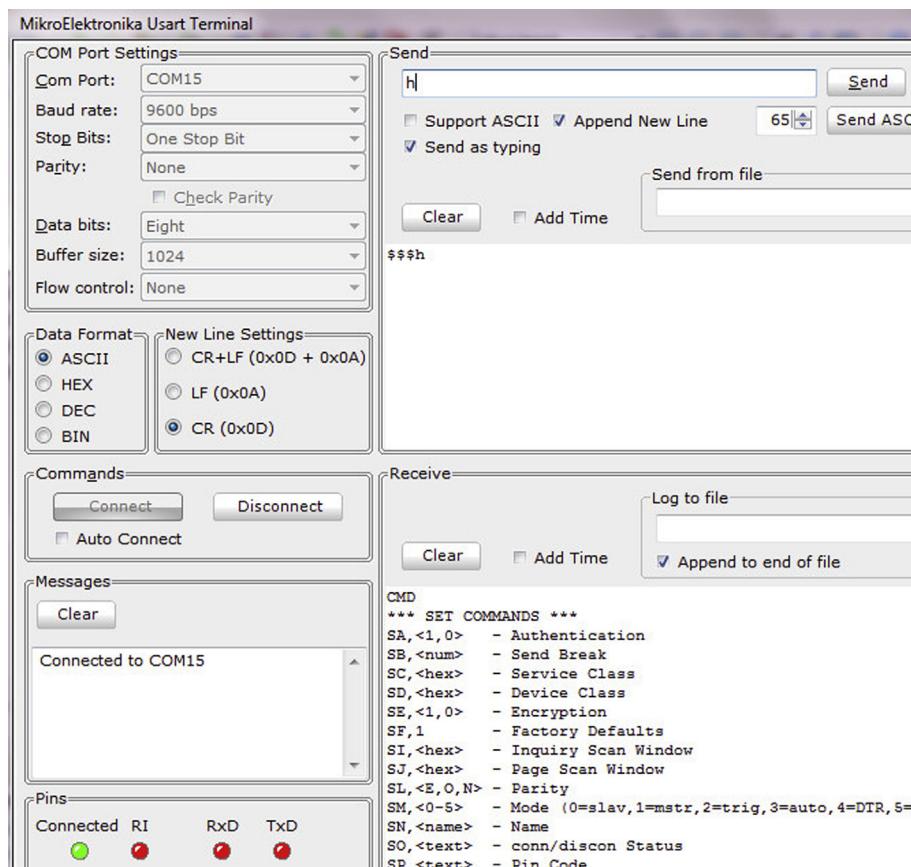


Figure 7.4: “CMD” Response from the Bluetooth Module.



**Figure 7.5: Part of the Default Configuration Settings.**

**Set commands:** These commands store information to the flash memory. Changes take effect after a power cycle or reboot.

**Get commands:** These commands retrieve and display the stored information.

**Change commands:** These commands temporarily change the values of various settings.

**Action commands:** These commands perform action commands such as connections.

**GPIO commands:** These commands configure and manipulate the GPIO signals.

Some of the commonly used command examples are given below (details of the full command list can be obtained from the manufacturer's data sheet):

**SA, <value>**

This command forces authentication when a remote device attempts to connect.

The <value> can be 0, 1, 2, or 4. The default value is 1 where the remote host

receives a prompt to pair. The user should press OK or YES on the remote device to authenticate.

*SM, <value>*

This command sets the operation mode. The options are

- 0: Slave mode
- 1: Master mode
- 2: Trigger mode
- 3: Autoconnect master mode
- 4: Autoconnect DTR mode
- 5: Autoconnect any mode
- 6: Pairing mode

The default value is 0.

*SN, <value>*

This command sets the device name. In the following example, the device name is set to Micro-Book:

```
SN, Micro-Book
```

*SO, <value>*

This command sets the extended status string (up to eight characters long). When set, two status messages are sent to the local serial port: When a Bluetooth connection is established, the device sends the string <string>CONNECT. Also, when disconnecting, the device sends the string <string>DISCONNECT.

*SP, <string>*

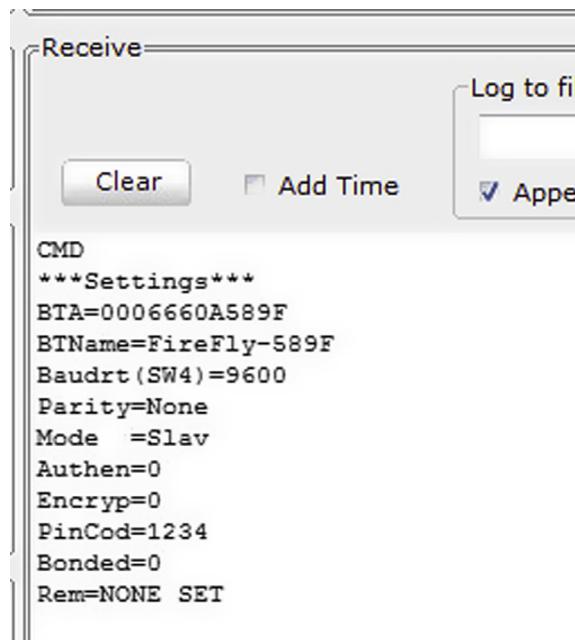
This command sets the security pin code. The string can be up to 20 characters long. The default value is 1234.

*SU, <value>*

This command sets the Baud rate. Valid values are 1200, 2400, 4800, 9600, 19.2, 28.8, 38.4, 57.6, 115, 230, 460, or 921 K. The default value is either 115 K or 9600, set by the GPIO7 pin.

*D*

This command displays basic settings, such as the address, name, and pin code. [Figure 7.6](#) shows an example.



**Figure 7.6: Displaying the Basic Settings with the D Command.**

**C, <address>**

This command forces the device to connect to the specified remote address, where the address must be specified in hexadecimal format.

**K**

This command disconnects the current connection.

**R, 1**

This command causes a reboot of the Bluetooth module.

**I, <value1>, <value2>**

This command performs a scan for a device. <value1> is the scan time, and 10 s is assumed if not specified. The maximum scan time is 48 s <value2> is the optional COD of the device.

**SR, <hex value>**

This command stores the remote address as a 12-digit hexadecimal code (6 bytes). Two additional characters can be specified with this command:

SR,I writes the last address obtained using the inquiry command. This option is useful when there is only one other Bluetooth device in the range.

SR,Z erases any stored addresses.

### Making a Connection

The RN41 Bluetooth module has several operating modes. By default, the RN41 Bluetooth module is a slave device, and the other device (e.g. PC) is the master. Assuming that the device is configured as the slave, connection to a master is as follows:

- Discovery: This phase is only available in the slave mode. When we turn on the device in the slave mode, it is automatically discoverable. In this phase, the Bluetooth device is ready to pair with other devices, and it broadcasts its name, profile, and MAC address. If the master is a PC, then the Bluetooth manager displays a list of discoverable devices (if there is more than one).
- Pairing: During this phase, the master and slave devices validate the pin code, and if the validation is successful, they exchange security keys, and a link key is established. Double clicking the Bluetooth manager on the PC will pair with the device and create a virtual serial COM port for the communication.
- Connecting: If the pairing is successful and the link key is established, then the master and the slave connect to each other.

If the device is configured as the master, then the connection to a slave is as follows:

- The module makes connections when a connect command is received. The Bluetooth address of the remote node can be specified in the command string. The master mode is useful when we want to initiate connections rather than receive connections. Note that in the master mode the device is neither discoverable nor connectable.

The RN41 Bluetooth module also supports Autoconnect modes where the module makes connections automatically on power up and reconnects when the connection is lost.

### Manual Connection Example

A manual connection example is given in this section to show the steps in connecting to a master device. Here, we assume that the master device is a PC, and the Bluetooth is enabled on the PC:

- Make sure that the Bluetooth on the PC is enabled to accept connections. Open Bluetooth Settings and enable the Discovery as shown in [Figure 7.7](#).
- Configure and connect the EasyBluetooth board to PORTC as described earlier, configure the EasyPIC V7 development board, connect a cable from the USB UART on the development board to the PC USB port. Start the terminal emulation software on the PC. Get into the command mode (Press button RC1 and enter \$\$\$ without any other characters). Name device as “Bluetooth-Slave” (optional, command SN), set extended



Figure 7.7: Enable the PC to Accept Connections.

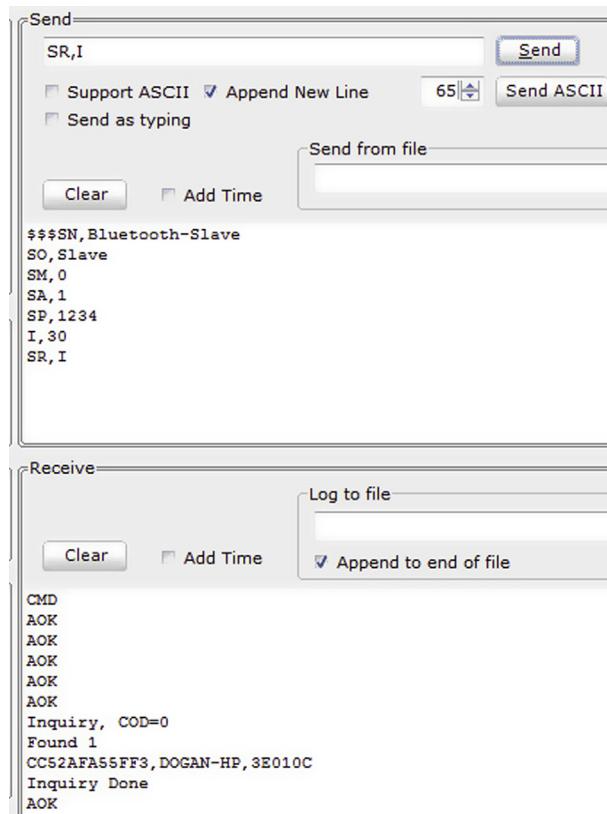


Figure 7.8: Steps in Finding a Bluetooth Device.

status string to “Slave” (optional, command SO), set the mode to slave (optional, command SM), enable authentication (optional, command SA), change the pass code to 1234 (command SP), look for Bluetooth devices (command I), and store the address of the found device (command SR). These steps are shown in [Figure 7.8](#).

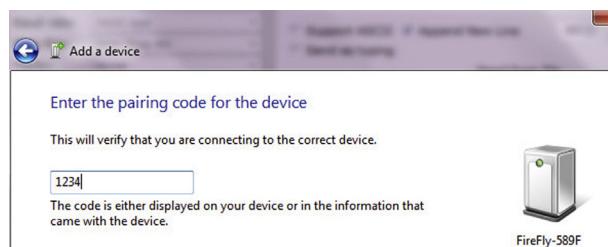


Figure 7.9: Entering the Pass Code.

- Connect (command C). Enter the pass code (default 1234) as shown in Figure 7.9. You should get a message to say that the device has been added to the computer (Figure 7.10).

You can check whether the device is added to the computer by clicking the “Bluetooth Devices” and then selecting “Show Bluetooth Devices” (Figure 7.11. The device name is “Bluetooth-Slave”).

- Exit from the command menu by entering characters “—“. The Bluetooth device should respond with string “END”.

Figure 7.12 shows the circuit diagram of the project. The RX, TX, and RESET pins of the RN41 Bluetooth module are connected to pin RC7 (UART RX), pin RC6 (UART TX), and pin RC1 of the microcontroller, respectively.

If you are using the EasyBluetooth board with the EasyPIC V7 development board, you should directly plug in the EasyBluetooth board to the PORTC connector (no pin reversal necessary) of the development board and then configure the following jumpers on the EasyBluetooth board:

- DIL switch SW1: Set 1 to ON (this connects RN41 RX to UART TX)
- DIL switch SW1: Set 4 to ON (this connects RN41 TX to UART RX)
- DIL switch SW1: Set 7 to ON (this connects RN41 RESET to RC1)

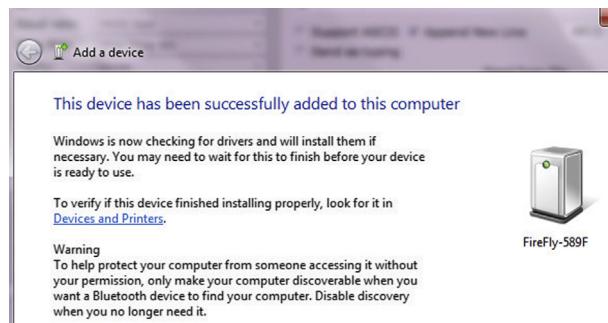


Figure 7.10: Device Added to the Computer.

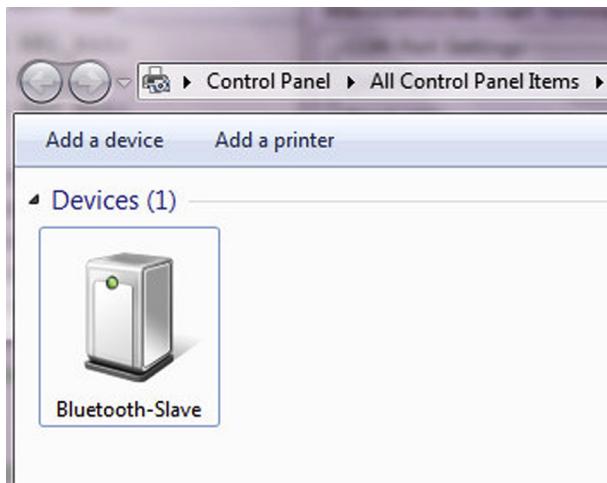


Figure 7.11: Checking whether the New Device is Added to the Computer.

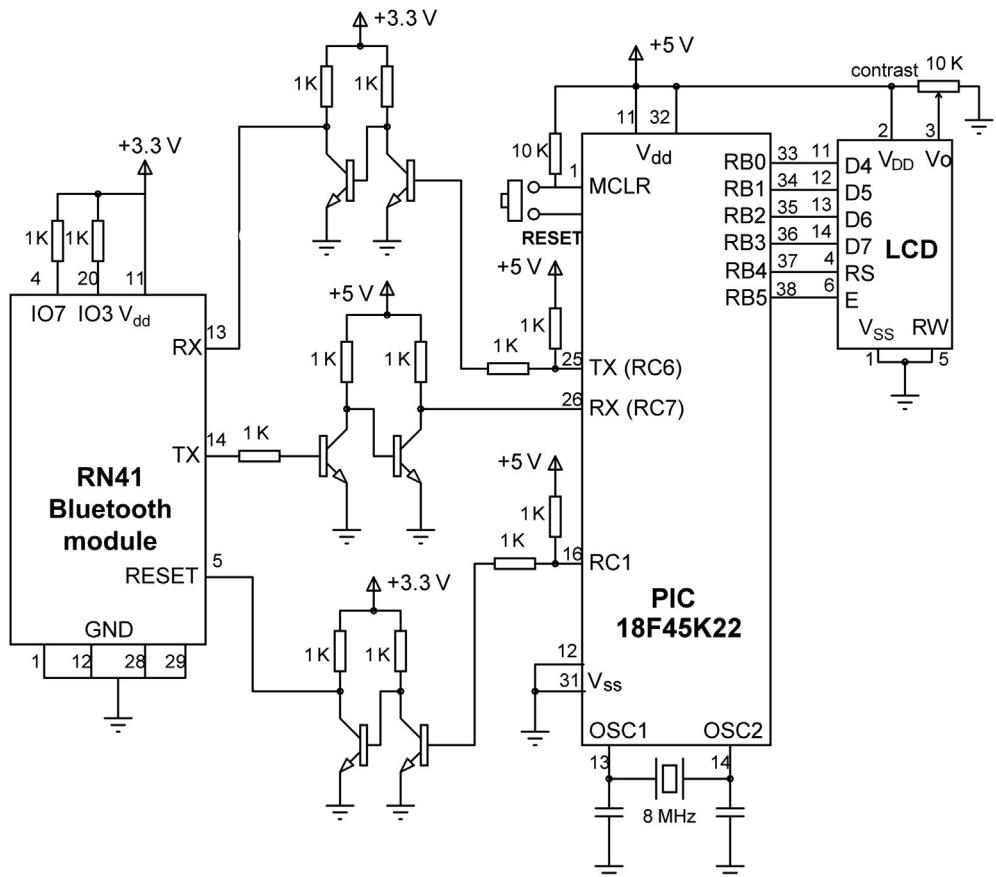
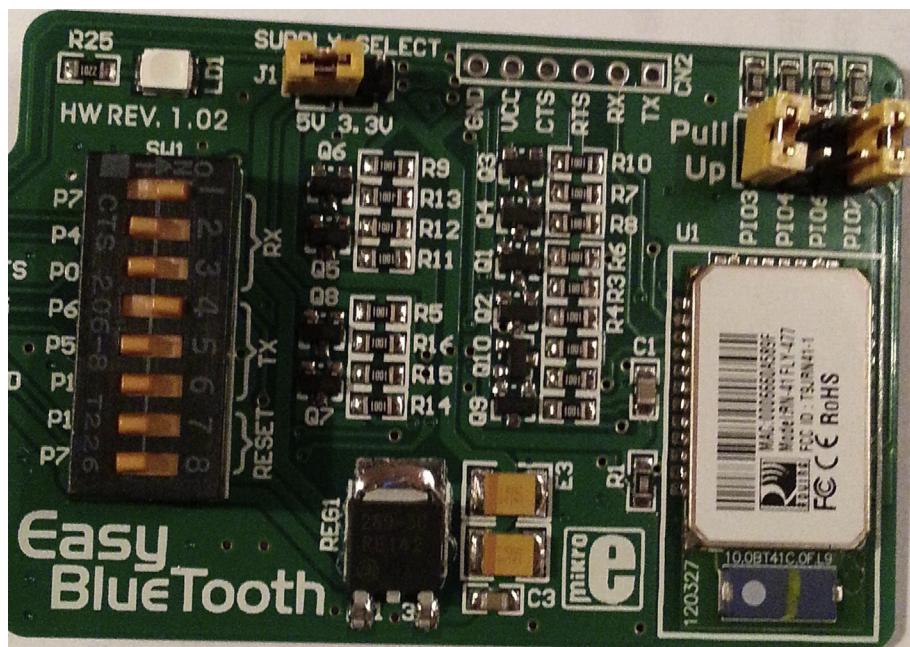


Figure 7.12: Circuit Diagram of the Project.



**Figure 7.13:** EasyBluetooth Board Jumper Settings.

Set J1 to 5 V (EasyPIC V7 operates at +5 V)

Set PI03 (GPI03) to Pull-Up

Set PI07 (GPIO7) to Pull-Up (9600 baud)

CTS and RTS pins are connected to a pad (CN2) on the EasyBluetooth board and are not used in his project.

Figure 7.13 shows the jumper settings on the EasyBluetooth board.

*Project PDL*

The project PDL is shown in Figure 7.14.

## *Project Program*

*mikroC Pro for PIC*

The mikroC pro for the PIC program listing is shown in Figure 7.15 (MIKROC-BLUE1.C). The following sequence describes the operations performed by the program:

- Get into command mode (command \$\$\$)

Main Program

```

BEGIN
    Define the connection between the LCD and microcontroller
    Configure PORTB and PORTC as digital
    Configure RC1 as output
    Enable UART interrupts
    Initialize LCD
    Initialize UART to 9600 baud
    Reset the Bluetooth device
    Get into command mode
    Set device name
    Set extended status
    Set slave mode
    Enable authentication
    Set pass code
    Wait until connected by the master
DO FOREVER
    IF message received flag is set
        Display message on the LCD
    ENDIF
ENDDO
END

```

```

BEGIN/INTERRUPT
    IF this is UART receive interrupt
        IF command mode
            Return command response
        ELSE
            Get message
            Set message received flag
        ENDIF
        Clear UART interrupt flag
    ENDIF
END/INTERRUPT

```

```

BEGIN/SEND_COMMAND
    DO WHILE correct response not received
        Send command to Bluetooth module
        Send carriage-return character
        Wait 500ms
    ENDDO
END

```

Figure 7.14: Project PDL.

- Configure the device by sending the following commands (these are optional and the defaults can be used if desired):
  - Device name (command SN),
  - Extended status (command SO),

```
*****
BLUETOOTH COMMUNICATION
=====

This project is about using Bluetooth communication in a project. In this project a Bluetooth module is used in slave mode. A PC is used in master mode. Messages sent to the slave module are displayed on an LCD.

The Easy Bluetooth board (www.mikroe.com) is used in this project, connected to PORTC of an EasyPIC V7 development board. An LCD is connected to PORTB of the microcontroller as in the previous projects.

Author: Dogan Ibrahim
Date: September 2013
File: MIKROC-BLUE1.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 RESET PORTC.RC1

const CMD = 1;
const AOK = 2;
const CONN = 3;
const END = 4;

unsigned char Command_Mode, temp, Response, Data_Received;
unsigned char Buffer[6], Txt[16];
unsigned char Cnt = 0;
unsigned char i = 0;
//
// UART receive interrupt handler. In Command Mode we get the following responses:
// CMD, AOK, END, CONN. The interrupt handler returns responses in command mode
// and also stores the received data from the master
//
void interrupt(void)
{
    if(PIR1.RC1IF == 1)                                // Is this a UART receive interrupt ?
```

**Figure 7.15: mikroC Pro for the PIC Program.**

```

{
    temp = UART1_Read();                                // Read the received character
    if(Command_Mode == 1 && temp != 0x0)
    {
        Buffer[Cnt] = temp;
        Cnt++;
        if(Cnt == 4)
        {
            if(Buffer[0] == 'C' && Buffer[1] == 'O' && Buffer[2] == 'N' && Buffer[3] == 'N')
            {
                Response = CONN;
                Cnt = 0;
            }
        }

        if(Cnt == 5)
        {
            Cnt = 0;
            Response = 0;
            if(Buffer[0] == 'C' && Buffer[1] == 'M' && Buffer[2] == 'D' && Buffer[3] == 0x0D &&
               Buffer[4] == 0x0A)Response = CMD;
            if(Buffer[0] == 'A' && Buffer[1] == 'O' && Buffer[2] == 'K' && Buffer[3] == 0x0D &&
               Buffer[4] == 0x0A)Response = AOK;
            if(Buffer[0] == 'E' && Buffer[1] == 'N' && Buffer[2] == 'D' && Buffer[3] == 0x0D &&
               Buffer[4] == 0x0A)Response = END;
        }
    }
    else
    {
        if(temp == 0x0D)                                // If END of data
        {
            Data_Received = 1;                          // End of data received flag
            Txt[i] = 0x0;                             // Terminate data with NULL
        }
        else
        {
            Txt[i] = temp;                           // Store received data
            i++;                                  // Increment for next character
        }
    }
    PIR1.RC1IF = 0;                                // Clear UART interrupt flag
}
}

//
// Send a command to the Bluetooth Module. The first argument is the command string
// to be sent to the Bluetooth module. The second argument is the Response expected
// from the module (can be CMD, AOK, END, or CONN)
//
void Send_Command(char *msg, unsigned char Resp)

```

**Figure 7.15**  
cont'd

```

{
do
{
    UART1_Write_Text(msg);
    UART1_Write(0x0D);
    Delay_Ms(500);
}while(Response != Resp);
}

// Start of MAIN program
//
void main()
{
    ANSELB = 0;                                // Configure PORTB as digital
    ANSELC = 0;                                // Configure PORTC as digital
    TRISC1_bit = 0;                            // Configure RC1 as an output
//
// Enable UART receive interrupts
//
    PIE1.RC1IE = 1;                           // Clear UART1 interrupt flag
    INTCON.PEIE = 1;                           // Enable UART1 interrupts
    INTCON.GIE = 1;                            // Enable global interrupts

    LCD_Init();                                // Initialize LCD
    Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD
    Lcd_Cmd(_LCD_CURSOR_OFF);                  // Cursor off
    Lcd_Out(1,1,"Cmd Mode");                  // Display message

    Uart1_Init(9600);                         // Initialzie UART to 9600 Baud

    Command_Mode = 1;                          // We are getting into Command Mode
    RESET = 0;                                 // Reset the Bluetooth module
    Delay_Ms(100);
    RESET = 1;
    Delay_Ms(1000);                           // End of Resetting the Bluetooth module

    do
    {
        UART1_Write_Text("$$$");
        Delay_Ms(1000);
    }while(Response != CMD);

    Send_Command("SN,Bluetooth-Testing",AOK);   // Set device name
    Send_Command("SO,Slave", AOK);              // Set extended status
    Send_Command("SM,0", AOK);                 // Set into slave mode
    Send_Command("SA,1", AOK);                 // Enable authentication
    Send_Command("SP,1234", AOK);              // Set pass code
    Send_Command("---", END);                  // Exit command mode
}

```

**Figure 7.15**  
cont'd

```

Lcd_Out(1,1,"Connecting");           // Display message
while(Response != CONN);           // Wait until connected

Command_Mode = 0;                   // Now we are in Data mode
Data_Received = 0;

Lcd_Out(1,1,"Connected ");

for(;;)                           // Display received messages on the LCD
{
    i = 0;
    while(Data_Received == 0);     // Wait until data up to CR is received
    Data_Received = 0;             // Clear data received flag
    Lcd_Cmd(_LCD_CLEAR);          // Clear LCD
    Lcd_Out(1,1,"Received Data:"); // Display "Received Data:" on first row
    Lcd_Out(2,1,Txt);             // Displayed received data
    for(i = 0; i < 15; i++)Txt[i] = 0; // Clear buffer for next time
}
}

```

**Figure 7.15**  
cont'd

Slave mode (command SM),  
 Enable authentication (command SA),  
 Set pass code (command SP),

- Exit command mode (command “—“).
- Wait for connection request from the master (PC).
- Read data from the master and display on the LCD.

At the beginning of the program, the connections between the LCD and the microcontroller are defined, symbol RESET is assigned to port pin RC1, and the various module responses are defined.

The main program configures PORTB and PORTC as digital, and RC1 pin is configured as an output pin. The LCD is initialized, the UART module is initialized to operate at 9600 baud, and the Bluetooth module is reset.

The program then puts the Bluetooth module into the command mode by sending characters “\$\$\$” and waiting for the response string “CMD” to be received. This process is repeated until a response is received from the Bluetooth module. Once the correct response is received, commands are sent to the module to set the device name, extended status, mode, etc. Function Send\_Command is used to send commands to the module. This function consists of the following code:

```

void Send_Command(char *msg, unsigned char Resp)
{
    do
    {
        UART1_Write_Text(msg);

```

```
    UART1_Write(0x0D);
    Delay_Ms(500);
}while(Response != Resp);
```

The first argument is the command string, while the second argument is the expected response from the module. The last command to send is the “—”, which exits from the command mode and waits for a connection from the master. Once a connection is established, the program waits until data are received from the master (until variable Data\_Received becomes 1). The received text message in character array Txt is displayed on the second row of the LCD. Txt is then cleared, ready for then next message to be received.

Commands and data are received inside the UART data receive interrupt service routine (ISR). At the beginning of the ISR, the program checks to see whether or not the cause of the interrupt is actually the UART data reception. If this is the case, the received data are stored in variable temp. If we are in the command mode (Command\_Mode = 1), then the received character must form part of the command response, and this character is stored in character array Buffer. After receiving four characters, the program checks to see if the received response is “CONN”, and if so, the Response is set to CONN; otherwise, two more characters are received and the program checks to see if the response is one of “CMD”, “AOK”, or “END”, followed by carriage-return (0x0D) and line-feed (0x0A) characters. The correct response is returned to the main program. If we are in the data mode (Command\_Mode = 0), then the received character in temp is copied to character array Txt. If the end of data is detected (0x0D), then a NULL character is inserted at the end of Txt to turn it into a string.

The responses to various commands are as follows:

Command	Response
\$\$\$	CMD<cr><lf>
SN<cr>	AOK<cr><lf>
SO<cr>	AOK<cr><lf>
SM<cr>	AOK<cr><lf>
SA<cr>	AOK<cr><lf>
SP<cr>	AOK<cr><lf>
-<cr>	END<cr><lf>

### Testing the Program

In this section, we shall see how we can connect to a PC master device and receive a message from the PC and then display this message on the LCD. In this test, a Windows 7 PC is used.

- Make sure the Bluetooth adapter on the PC is turned on and the PC is set so that Bluetooth device can find the computer (open “Bluetooth Devices” in the hidden icons in the status bar and then “Open Settings” and configure as necessary if this is not the case).
- Compile and run the program. You should see the message “Connecting” on the LCD.

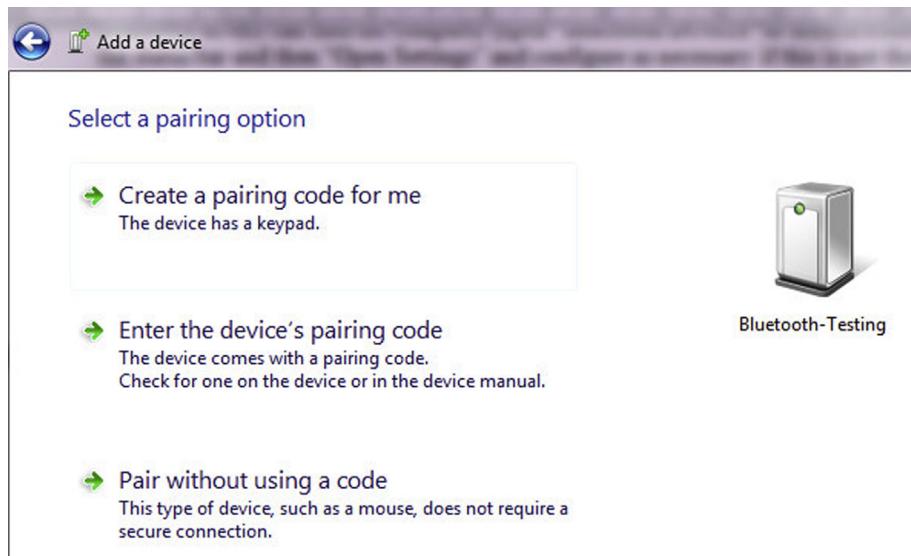


**Figure 7.16:** Device “Bluetooth Testing”.

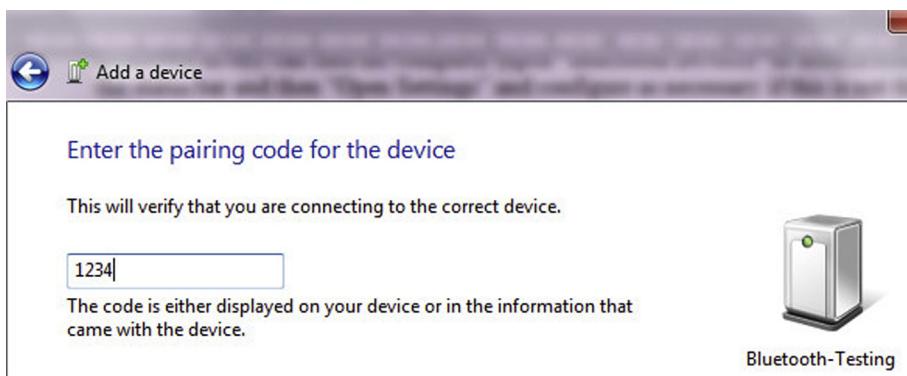
- Open “Bluetooth Devices” on the PC. Select “Add a Device”. You should see the device name “Bluetooth Testing” ([Figure 7.16](#)). Double click on this device, and the program will ask you to enter the pairing code ([Figure 7.17](#)). Enter “1234” ([Figure 7.18](#)). The master should now connect to our slave device.

You are now ready to send messages to the slave device. The messages can be sent by finding the COM port that the Bluetooth is using on the PC for outgoing data. Open “Bluetooth Devices” on the PC. Then, “Open Settings”, click “COM Ports” and see which port is assigned for outgoing data. In [Figure 7.19](#), COM17 is the required port.

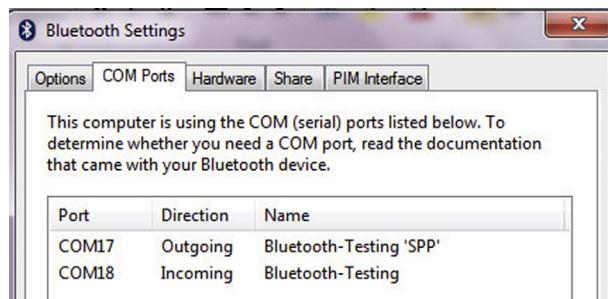
Now, we can send data via COM17 to our Bluetooth module. Start the Hyperterm terminal emulation software and type COM17 for “Connect using”. Select 9600 Baud, 8 bits, no



**Figure 7.17:** Asking the Pairing Code.



**Figure 7.18: Entering the Pairing Code.**



**Figure 7.19: COM17 is Assigned for Outgoing Data.**



**Figure 7.20: Example Message Displayed on the LCD.**

parity, no flow control. Type a message, followed by the Enter key. You should see the message displayed on the LCD. Note that by default the characters you type on the Hyperterm window are not echoed. You can turn the echo ON by selecting and enabling File → Properties → Settings → ASCII Setup → Echo Typed Characters Locally. An example message is shown in Figure 7.20.

The Bluetooth connection can be broken, and the device can be removed from the PC easily. Open “Bluetooth Devices”, click “Show Bluetooth Devices”, right click on the

device, and select “Remove Device”. Next time you connect you will need to enter the pass code again.

### Modifications

The program in [Figure 7.20](#) can be used for remote control applications. For example, the following modification shows how a message can be sent to the Bluetooth module to turn ON required light emitting diodes (LEDs) of PORTD. Here, for simplicity, we will assume that the message format is

```
PD = nnn
```

Where nnn is the three-digit decimal data (000–255) to be sent to PORTD.

Insert to the beginning of the program:

```
ANSEL0 = 0; // Configure PORTD as digital
TRISD = 0; // Configure PORTD as output
PORTD = 0; // Clear PORTD to start with
```

Modify last part of the program as follows:

```
for(); // Display received messages on the LCD
{
    i = 0;
    while(Data_Received == 0); // Wait until data up to CR is received
    Data_Received = 0; // Clear data received flag
    Lcd_Cmd(_LCD_CLEAR); // Clear LCD
    Lcd_Out(1,1,"Received Data:");
    Lcd_Out(2,1,Txt); // Displayed the received data
    //
    // Check for command PD = nnn and set PORTD accordingly
    //
    if(Txt[0] == 'P' && Txt[1] == 'D' && Txt[2] == '=')
    {
        i = 100*(Txt[3]-‘0’) + 10*(Txt[4] - ‘0’) + Txt[5] - ‘0’;
        PORTD = i;
    }
    for(i = 0; i < 15; i++)Txt[i] = 0; // Clear buffer for next time
}
```

As an example, entering command PD = 067 will turn ON LEDs 6, 1, and 0 of PORTD.

### MPLAB XC8

The MPLAB XC8 program is similar, and the full program listing is shown in [Figure 7.21](#) (XC8-BLUE1.C). Note that the LCD connections are different in the MPLAB XC8 version of the program as described in earlier MPLAB XC8 projects. In addition, since the MPLAB XC8 UART function putrsUSART sends the NULL character as well to the

```
*****
                    BLUETOOTH COMMUNICATION
=====

This project is about using Bluetooth communication in a project. In this project a Bluetooth module is used in slave mode. A PC is used in master mode. Messages sent to the slave module are displayed on an LCD.

The Easy Bluetooth board (www.mikroe.com) is used in this project, connected to PORTC of an EasyPIC V7 development board. An LCD is connected to PORTB of the microcontroller as in the previous MPLAB XC8 projects.

Author: Dogan Ibrahim
Date: September 2013
File: XC8-BLUE1.C
*****/




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

#define RST PORTCbits.RC1

unsigned char CMD = 1;
unsigned char AOK = 2;
unsigned char CONN = 3;
unsigned char END = 4;

unsigned char Command_Mode, temp, Response, Data_Received;
unsigned char Buffer[6], Txt[16];
unsigned char Cnt = 0;
unsigned char i = 0;
// UART receive interrupt handler. In Command Mode we get the following responses:
// CMD, AOK, END, CONN. The interrupt handler returns responses in command mode
// and also stores the received data from the master
//
void interrupt isr(void)
{
    if(PIR1bits.RC1IF == 1)                                // Is this a UART receive interrupt ?
    {
        temp = getc1USART();                                // Read the received character
        if(Command_Mode == 1 && temp != 0x0)
        {
            Buffer[Cnt] = temp;
            Cnt++;
            if(Cnt == 4)
            {

```

**Figure 7.21: MPLAB XC8 Program.**

```

if(Buffer[0] == 'C' && Buffer[1] == 'O' && Buffer[2] == 'N' && Buffer[3] == 'N')
{
    Response = CONN;
    Cnt = 0;
}
}

if(Cnt == 5)
{
    Cnt = 0;
    Response = 0;
    if(Buffer[0] == 'C' && Buffer[1] == 'M' && Buffer[2] == 'D' && Buffer[3] == 0x0D &&
       Buffer[4] == 0xA)Response = CMD;
    if(Buffer[0] == 'A' && Buffer[1] == 'O' && Buffer[2] == 'K' && Buffer[3] == 0x0D &&
       Buffer[4] == 0xA)Response = AOK;
    if(Buffer[0] == 'E' && Buffer[1] == 'N' && Buffer[2] == 'D' && Buffer[3] == 0x0D &&
       Buffer[4] == 0xA)Response = END;
}
else
{
    if(temp == 0xD)                                // If END of data
    {
        Data_Received = 1;                          // End of data received flag
        Txt[i] = 0x0;                             // Terminate data with NULL
    }
    else
    {
        Txt[i] = temp;                            // Store received data
        i++;                                    // Increment for next character
    }
    PIR1bits.RC1IF = 0;                           // Clear UART interrupt flag
}
}

// 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);
    }
}

//

```

**Figure 7.21**  
cont'd

```
// This function creates milliseconds delay. The argument specifies the delay time in ms
//
void Delay_Ms(unsigned int ms)
{
    unsigned int i;

    for(i = 0; i < ms; i++)__delay_ms(1);
}

//
// 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);
}
```

**Figure 7.21**  
cont'd

```

// 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( BusyLCD() );
    SetDDRamAddr( ddaddr );
}

// This function sends commands to the Bluetooth module. The response is checked
// after sending a command
//
void Send_Command(const char msg[], unsigned char Resp)
{
    unsigned char i;

    do
    {
        i = 0;
        do
        {
            while(Busy1USART());
            putc1USART(msg[i]);                                // Check if UART is busy
            i++;                                              // Send to UART
        }while(msg[i] != 0x00);                             // Until NULL terminator detected

        while(Busy1USART());                                // Send carriage-return at the end
        putc1USART(0x0D);
        Delay_Ms(500);
    }while(Response != Resp);
}

// Start of MAIN program
//
void main()
{
    ANSELB = 0;                                         // Configure PORTB as digital
    ANSELC = 0;                                         // Configure PORTC as digital
    TRISCbits.RC1 = 0;                                   // Configure RC1 as an output

    // Enable UART receive interrupts
    PIE1bits.RC1IE = 1;                                 // Clear UART1 interrupt flag
    INTCONbits.PEIE = 1;                               // Enable UART1 interrupts
    INTCONbits.GIE = 1;                               // Enable global interrupts
}

```

**Figure 7.21**  
cont'd

```

// Configure the LCD to use 4-bits, in multiple display mode
//
Delay_Seconds(1);
OpenXLCD(FOUR_BIT & LINES_5X7);

Open1USART( USART_TX_INT_OFF  &           // Initialize UART
            USART_RX_INT_ON  &
            USART_ASYNCH_MODE &
            USART_EIGHT_BIT   &
            USART_CONT_RX    &
            USART_BRGH_LOW,
            12);
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 LCD
LCD_Move(1,1);
putrsXLCD("Cmd Mode");                    // Display message

Command_Mode = 1;                          // Getting into Command Mode
RST = 0;                                   // Reset the Bluetooth module
Delay_Ms(100);
RST = 1;
Delay_Seconds(1);                         // End of Reset

do                                         // Get into command mode
{
    putc1USART('$'); // Get into command mode
    while(Busy1USART());
    putc1USART('$');
    while(Busy1USART());
    putc1USART('$');
    Delay_Seconds(1);
}while(Response != CMD);

Send_Command("SN,BLUETOOTH2",AOK);          // Set device name
Send_Command("SO,SLAVE",AOK);                // Set extended status
Send_Command("SM,0",AOK);                   // Set into slave mode
Send_Command("SA,1",AOK);                   // Enable authentication
Send_Command("SP,1234",AOK);                // Set pass code
Send_Command("---", END);                  // Exit command mode
LCD_Move(1,1);
putrsXLCD("Connecting");                  // Display message
while(Response != CONN);                  // Wait until connected

Command_Mode = 0;                          // Now we are in Data mode
Data_Received = 0;

```

**Figure 7.21**  
cont'd

```

LCD_Move(1,1);
putrsXLCD("Connected ");
//
// The received message is displayed inside this loop
//
for(;;)                                // Endless loop
{
    i = 0;
    while(Data_Received == 0);           // Wait until CR is received
    Data_Received = 0;                  // Clear data received flag
    LCD_Clear();                      // Clear LCD
    LCD_Move(1,1);
    putrsXLCD("Received Data:");
    LCD_Move(2,1);
    putrsXLCD(Txt);
    for(i = 0; i < 15; i++)Txt[i] = 0; // Display the received data
                                         // Clear buffer for next time
}
}

```

**Figure 7.21**  
cont'd

UART, we have used the putcUSART function to send characters to the Bluetooth module with no additional bytes.

### **Project 7.2—Bluetooth Serial Communication—Master Mode**

In this project, we shall be using a Bluetooth module to communicate with a PC. The program will connect to a PC and then send a text message to the PC, which will be displayed on the screen. In this project, our Bluetooth device is used as a master device and the PC is used as a slave device.

This project is very similar to the previous project. Here, after a connection is established, the message “Bluetooth Test” will be sent to the master device every second. The LCD is used to display various messages and also the text sent to the slave device.

#### **Project Hardware**

The circuit diagram and the hardware configuration of this project is the same as the one given in the previous project.

#### **Project PDL**

The project PDL is given in [Figure 7.22](#).

**MAIN PROGRAM:****BEGIN**

Define connection between LCD and microcontroller  
Configure PORTB and PORTC as digital outputs  
Enable UART receive interrupts  
Initialize LCD  
Initialize UART to 9600 Baud  
Get into command mode  
Set Device name (command SN)  
Extended status (command SO)  
Slave mode (command SM)  
Enable authentication (command SA)  
Set pass code (command SP)  
Look for Bluetooth devices (command I)  
Store the address of the found Bluetooth device (command SR)  
Connect to the slave device (command C)

**DO FOREVER**

Send text "Bluetooth Test" to the slave device  
Display text "Bluetooth Test" on the LCD  
Wait 1 second  
Clear LCD

**ENDDO****END****Interrupt Service Routine:****BEGIN**

**IF** this is a UART receive interrupt  
    Read received character  
**ELSE IF** command mode  
    Return CMD  
**ELSE IF** AOK  
    Return AOK  
**ELSE IF** Done  
    Return IDone  
**ELSE IF** CONNECT  
    Return CONN  
**ENDIF**

**ENDIF****END****BEGIN/ Send\_Command**

Send text to UART  
Send carriage-return character to UART  
Wait 500ms

**END/Send\_Command****Figure 7.22: Project PDL.**

## **Project Program**

### *mikroC Pro for PIC*

The mikroC pro for PIC program listing is shown in [Figure 7.23](#) (MIKROC-BLUE2.C). The following sequence describes the operations performed by the program:

- Get into command mode (command \$\$\$).
- Configure the device by sending the following commands (these are optional and the defaults can be used if desired):

Device name (command SN);  
 Extended status (command SO);  
 Slave mode (command SM);  
 Enable authentication (command SA);  
 Set pass code (command SP);  
 Look for Bluetooth devices (command I);  
 Store the address of the found Bluetooth device (command SR);  
 Connect to the slave device (command C);  
 Send text “Bluetooth Test” to the slave device every second. Display the sent message on the LCD.

At the beginning of the program, the connections between the LCD and the microcontroller are defined, symbol RESET is assigned to port pin RC1, and the various module responses are defined.

The main program configures PORTB and PORTC as digital, and the RC1 pin is configured as an output pin. The LCD is initialized, the UART module is initialized to operate at 9600 baud, and the Bluetooth module is reset.

The program then puts the Bluetooth module into the command mode by sending characters “\$\$\$” and waiting for the response string “CMD” to be received. This process is repeated until a response is received from the Bluetooth module. Once the correct response is received, commands are sent to the module to set the device name, extended status, mode, etc. Function `Send_Command` is used to send commands to the module as in the previous project. Command “I,30” looks for Bluetooth devices, and if a device is found, it gets its address. Command “SR,I” stores the address of the device just found. Command “C” connects to a slave device. The program then enters in a loop and sends message “Bluetooth Test” to the slave device every second.

Commands and data are received inside the UART data receive ISR. The ISR implemented here is slightly different from the one given in [Figure 7.15](#). At the beginning of the ISR, the program checks to see whether or not the cause of the

```
*****
BLUETOOTH COMMUNICATION
=====

This project is about using Bluetooth communication in a project. In this project a Bluetooth module is used in master mode. A PC is used in slave mode. Text "Bluetooth Test" is sent to the slave device every second.

The Easy Bluetooth board (www.mikroe.com) is used in this project, connected to PORTC of an EasyPIC V7 development board. An LCD is connected to PORTB of the microcontroller as in the previous projects.

Author: Dogan Ibrahim
Date: September 2013
File: MIKROC-BLUE2.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 RESET PORTC.RC1

const CMD = 1;
const AOK = 2;
const CONN = 3;
const IDone = 4;

unsigned char Response, temp;
unsigned char Buffer[60];
unsigned char i,Cnt = 0;
//
// UART receive interrupt handler. In Command Mode we get the following responses:
// CMD, AOK, Done, CONN. The interrupt handler returns responses in command mode
// and also stores the received data from the master
//
void interrupt(void)
{
    if(PIR1.RC1IF == 1) // Is this a UART receive interrupt ?
    {
```

**Figure 7.23: mikroC Pro for the PIC Program.**

```

temp = UART1_Read();           // Read the received character
if(temp != 0x0)
{
    if(temp == 0x0A)
    {
        if(Buffer[Cnt-4] == 'C' && Buffer[Cnt-3] == 'M' && Buffer[Cnt-2] == 'D' &&
           Buffer[Cnt-1] == 0x0D)
        {
            Response = CMD;
            Cnt = 0;
        }
        if(Buffer[Cnt-4] == 'A' && Buffer[Cnt-3] == 'O' && Buffer[Cnt-2] == 'K' &&
           Buffer[Cnt-1] == 0x0D)
        {
            Response = AOK;
            Cnt = 0;
        }
        if(Buffer[Cnt-5] == 'D' && Buffer[Cnt-4] == 'o' && Buffer[Cnt-3] == 'n' &&
           Buffer[Cnt-2] == 'e' && Buffer[Cnt-1] == 0x0D)
        {
            Response = IDone;
            Cnt = 0;
        }
        if(Buffer[Cnt-8] == 'C' && Buffer[Cnt-7] == 'O' && Buffer[Cnt-6] == 'N' &&
           Buffer[Cnt-5] == 'N' && Buffer[Cnt-4] == 'E' && Buffer[Cnt-3] == 'C' &&
           Buffer[Cnt-2] == 'T')
        {
            Response = CONN;
            Cnt = 0;
        }
        Cnt=0;
    }
    else
    {
        Buffer[Cnt] = temp;
        Cnt++;
    }
}
PIR1.RC1IF = 0;           // Clear UART interrupt flag
}

// Send a command to the Bluetooth Module. The first argument is the command string
// to be sent to the Bluetooth module. The second argument is the Response expected
// from the module (can be CMD, AOK, END, or CONN)
//
void Send_Command(char *msg, unsigned char Resp)
{
    do
    {

```

**Figure 7.23**  
cont'd

```

        UART1_Write_Text(msg);
        UART1_Write(0x0D);
        Delay_Ms(500);
    }while(Response != Resp);
}

// Start of MAIN program
//
void main()
{
    ANSELB = 0;                                // Configure PORTB as digital
    ANSELC = 0;                                // Configure PORTC as digital
    TRISC1_bit = 0;                            // Configure RC1 as an output
//
// Enable UART receive interrupts
//
    PIE1.RC1IE = 1;                           // Clear UART1 interrupt flag
    INTCON.PEIE = 1;                           // Enable UART1 interrupts
    INTCON.GIE = 1;                            // Disable global interrupts

    LCD_Init();                                // Initialize LCD
    Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD
    Lcd_Cmd(_LCD_CURSOR_OFF);                  // Cursor off
    Lcd_Out(1,1,"Cmd Mode");                  // Display message
    Uart1_Init(9600);                         // Initialize UART to 9600 Baud

    RESET = 0;                                 // Reset the Bluetooth module
    Delay_Ms(100);
    RESET = 1;
    Delay_Ms(1000);                          // End of Resetting the Bluetooth module

    do                                         // Send $$$ for command mode
    {
        UART1_Write_Text("$$$");
        Delay_Ms(1000);
    }while(Response != CMD);

    Lcd_Out(1,1,"Send Commands");
    Send_Command("SN,Bluetooth-Master",AOK);   // Set device name
    Send_Command("SO,Master", AOK);             // Set extended status
    Send_Command("SM,1", AOK);                  // Set into master mode
    Send_Command("SA,1", AOK);                  // Enable authentication
    Send_Command("SP,1234", AOK);               // Set pass code
    Lcd_Out(1,1,"Look For Devices");
    Lcd_Out(2,1,"Wait 30 secs");

    UART1_Write_Text("I,30");                   // Look for Bluetooth devices (wait 30 s)
    UART1_Write(0x0D);                        // Send carriage-return
}

```

**Figure 7.23**

cont'd

```

while(Response != IDone);           // Wait up to 30 s

Lcd_Cmd(_LCD_CLEAR);
Lcd_Out(1,1,"Device Found ");
Send_Command("SR,I", AOK);         // Display message
                                    // Store address of the device just found

UART1_Write_Text("C");
UART1_Write(0x0D);
while(Response != CONN);          // Connect to the slave just found

for(;;)
{
    Uart1_Write_Text("Bluetooth Test");
    Uart1_Write(0x0D);
    LCD_Out(1,1,"Sent Message:");
    Lcd_Out(2,1,"Bluetooth Test");
    Delay_Ms(1000);
    Lcd_Cmd(_LCD_CLEAR);
}
}

```

**Figure 7.23**  
cont'd

interrupt is actually the UART data reception. The command responses are terminated with the line-feed character 0x0A. The program stores the received characters in array Buffer until the line-feed character is received. Then, the program checks to see what type of response this is and sets variable Response accordingly. Response is set to CMD, AOK, Done, or CONNECT.

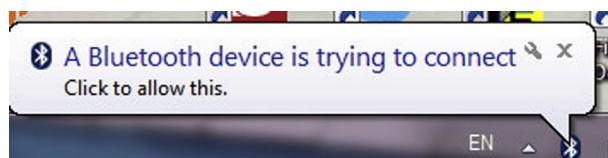
The Bluetooth device returns the following responses when a command is sent:

Command	Response
\$\$\$	CMD<cr><lf>
SN<cr>	AOK<cr><lf>
SO<cr>	AOK<cr><lf>
SM<cr>	AOK<cr><lf>
SA<cr>	AOK<cr><lf>
SP<cr>	AOK<cr><lf>
-<cr>	END<cr><lf>
I<cr>	Inquiry, COD=0<cr><lf> Found n<cr><lf> <address>,<name>,<COD>,<serial port><cr><lf>
SR<cr>	Inquiry Done<cr><lf> AOK<cr><lf>
C<cr>	TRYING<cr><lf> MasterCONNECT<cr><lf>

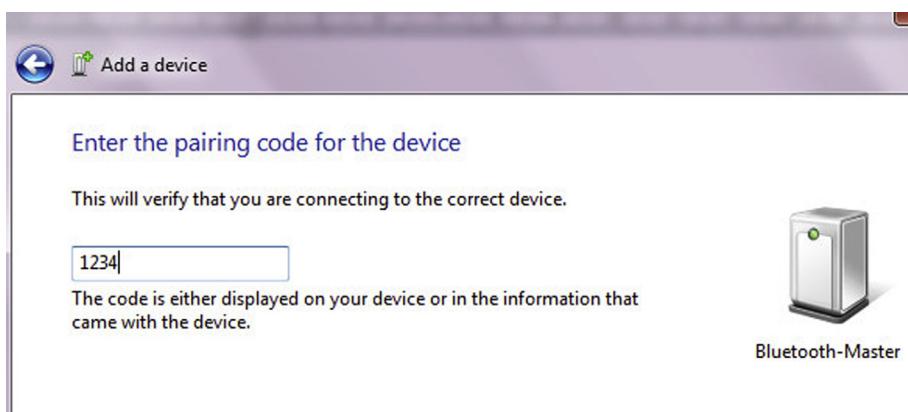
### Testing the Program

In this section, we shall see how we can connect to a PC slave device and send a message from our Bluetooth device. In this test, a Windows 7 PC is used.

- Make sure the Bluetooth adapter on the PC is turned on and the PC is set so that the Bluetooth device can find the computer (open “Bluetooth Devices” in hidden icons in the status bar and then “Open Settings” and configure as necessary if this is not the case). You should also enable the option “Alert me when a new Bluetooth device wants to connect”.
- Compile and run the program.
- You should see these messages on the LCD: “CMD Mode”, “Send Commands”, “Look For Devices”, “Wait 30 s”. The program will look for Bluetooth devices. This may take up to 30 s, and you should wait until the message “Device Found” is displayed on the LCD.
- At this stage, the PC will display an alert message to say that a new Bluetooth device wants to connect ([Figure 7.24](#)). Click the message and then enter the pass code 1234 ([Figure 7.25](#)). Open a terminal emulation session and enter the COM port number (in Bluetooth settings), and connect with 9600 baud, 8 bits, no parity, no flow control. The two devices will connect, and you should see the message “Bluetooth Test” displayed on the terminal emulation window.



**Figure 7.24:** Alert from the PC.



**Figure 7.25:** Entering the Pass Code.

You should now try as an exercise to use two EasyBluetooth boards, one configured as the master and one as the slave, and connect them to each other to exchange data.

### ***Project 7.3—Using the RFID***

In this project, we shall be using an RFID receiver to read the unique identifier (UID) number stored on an RFID tag.

Before going into the details of the project, it is worthwhile to review the basic principles of the RFID.

#### ***Radiofrequency Identification***

RFID involves the use of RF electromagnetic waves to transfer data without making any contact to the source of the data. Generally, an RFID system consists of two parts: A Reader and one or more transponders (also known as Tags) that carry the data. RFID systems evolved from barcode labels as a means of identifying an object. A comparison of RFID systems with barcodes reveals the following:

- Barcodes are read only. Most RFID systems are read write.
- Barcodes are based on optical technology and may be affected from environmental lighting, making the reliable reading distance not more than several feet. RFID systems are based on RF waves and are not affected by environmental lighting. The reading distance of an RFID system can be  $\geq 20\text{--}40$  ft.
- Barcode images are normally printed on paper, and their readability is affected by aging and the state of the paper, for example, dirt on the paper and torn paper. RFID systems do not suffer from environmental lighting, but their operation may be affected if attached to metals.
- Barcodes can be generated and distributed electronically, for example, via e-mail and mobile phone. For example, boarding passes with barcodes can be printed.
- Barcode reading is more labor intensive as it requires the light beam to be directed onto them. RFID readers on the other hand can read the data wirelessly and without touching the tags.
- Barcodes are much cheaper to produce and use than RFID systems.

Some common uses of RFID systems are as follows:

- Tracking of goods,
- Tracking of persons and animals,
- Inventory systems,
- Public transport and airline tickets,
- Access management,

- Passports,
- Hospitals and healthcare,
- Libraries,
- Museums,
- Sports,
- Defense,
- Shoplifting detection.

There are three types of RFID tags: Active, Passive, or Semipassive. Passive tags have no internal power sources, and they draw their power from the electromagnetic field generated by the RFID reader (Figure 7.26). Passive tags have no transmitters; they simply alter the electromagnetic field emitted by the reader such that the reader can detect. Because the reader has no transmitter, the range of passive tags is limited to several feet. Passive tags have the advantage that their cost is low compared to other types of tags.

Active tags have their own power sources (small batteries). These tags also have their own transmitters. As a result, active RFID systems have much greater detection ranges, usually a few hundred feet. To extend the battery life, these tags are normally in a low-power state until they detect the electromagnetic waves transmitted by the RFID receiver. After they leave the electromagnetic field, they return back to the low-power mode.

Semipassive tags have their own power sources (small batteries). But the power source is used just to power the microchip embedded inside the tag. Like the passive tags, these tags do not have any transmitters, and they rely on the same principle as the passive tags for transferring their data. Semipassive tags have greater detection ranges than the passive tags.

RFID tags can be read only where a fixed serial number is written on the tag, and this number is used by the reader to identify the tag, or they may be read write where data can be written onto the tag by the user. Some tags are write once, where the user can only write once onto the tag but read as many times as required.

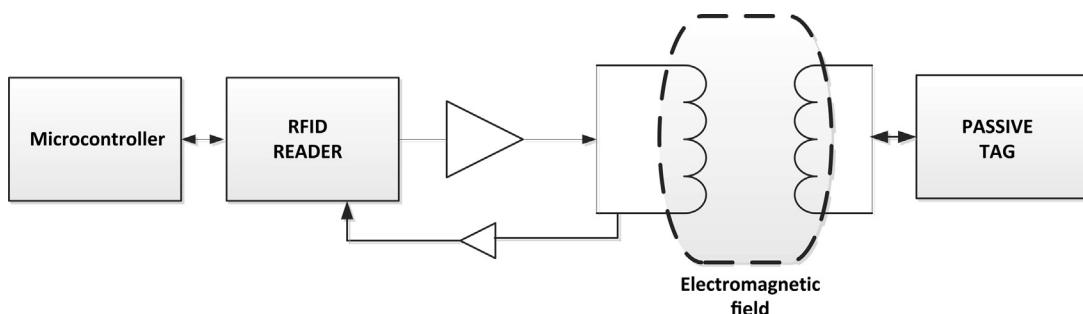
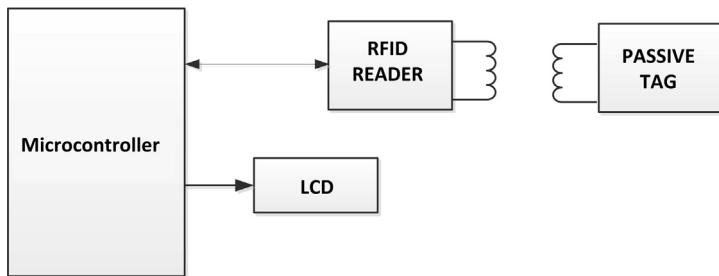


Figure 7.26: Passive RFID System.



**Figure 7.27:** Block Diagram of the Project.

The data on the tag are stored in a nonvolatile memory. This may be a small memory that, for example, stores just a serial number, or larger memory capable of storing, for example, product-related information, or a person's details.

An RFID tag contains at least two parts: an antenna for receiving (and transmitting on some tags) the signal and a microchip with memory for controlling all operations of the tag and for storing the data.

There are tags that operate in the LF, HF, or the UHF bands. Usually, every country sets its own rules and the allocation of frequencies. LF tags are generally passive, operating in the 120- to 150-kHz band, and their detection range is not  $>10$  cm. The 13.56-MHz ISM HF band is very popular for passive tags, offering good detection ranges up to 1 m. The 433-MHz ultra high frequency (UHF) band tags are active, offering detection ranges of over several hundreds of meters but having higher costs. The 865–868 UHF ISM band tags are passive with ranges up to 10 m, and having very low costs. The microwave band tags are active with high data rates and usually high ranges, but their cost is high.

There are several standards that control and regulate the design and development of RFID based products. It is important that the tag we use is compatible with the RFID receiver we are using. Also, the receivers generally support various standards and the selection of a particular standard is generally programmable. We should also make sure that the receiver and the tag use the same standard.

The block diagram of the project is shown in [Figure 7.27](#). An RFID receiver chip is connected to the microcontroller. The microcontroller reads the serial number (UID) on the tag and displays on the LCD.

### **Project Hardware**

The project uses the RFID Click board manufactured by mikrolektronika ([Figure 7.28](#)). This is a small mikroBUS compatible board, featuring the CR95HF 13.56-MHz RFID

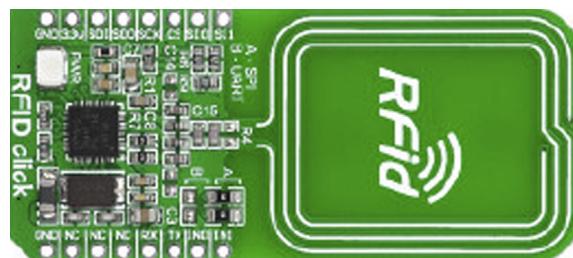


Figure 7.28: The RFID Click Board.

transceiver as well as the trace antenna. The board is designed to operate with a 3.3-V supply voltage, and it can communicate with the microcontroller using one of several busses, such as UART and serial peripheral interface (SPI). The basic specifications of the CR95HF chip are as follows:

- Support for reading and writing;
- A 13.56-MHz frequency, supporting the following standards:  
ISO/IEC 14443 Types A and B  
ISO/IEC 15693  
ISO/IEC 18092
- Host interface for UART, SPI, and INT;
- A 32-pin VFQFPN package;
- A 3.3-V operation.

In this project, we shall be using an RFID tag compatible with the standard ISO/IEC 14443 Type A. We shall be using the UART interface for simplicity. The CR95HF pins used while operating in the UART mode are as follows:

Pin	Pin name	Pin Description
1	TX1	Driver output to the coil
2	TX2	Driver output to the coil
5	RX1	Receiver input from the coil
6	RX2	Receiver input from the coil
8	GND	Ground
12	UART_RX/IRQ_IN	UART receive pin + Interrupt input
13	VPS	Power supply
14	UART_TX/IRQ_OUT	UART transmit pin + interrupt output
19	SSI_0	Select comms interface
20	SSI_1	Select comms interface
22	GND	Ground
29	XIN	Crystal input
30	XOUT	Crystal output
31	GND	Ground
32	VPS_TX	Power supply

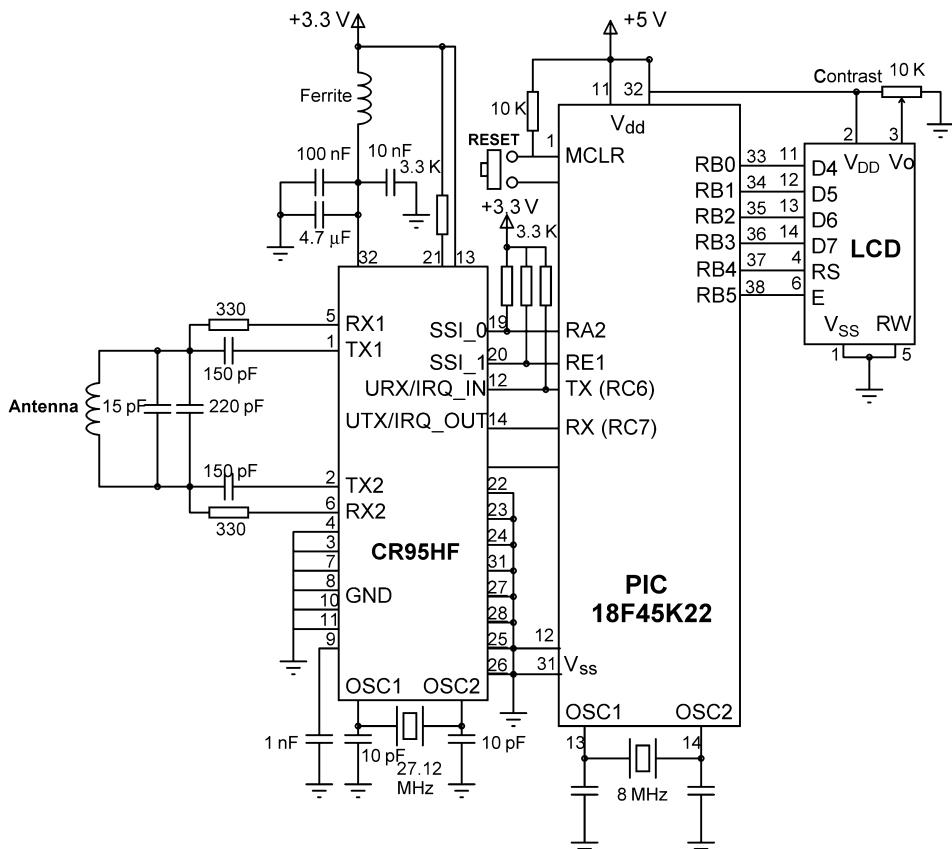
The CR95HF chip requires a 27.12-MHz crystal to be connected between its XIN-XOUT pins together with a pair of 10-pF capacitors. These are included on the RFID Click board.

**Figure 7.29** shows the circuit diagram of the project. The UART pins of the CR95HF chip, UART\_RX and UART\_TX are connected to UART pins RC6 (TX) and RC7 (RX) of the microcontroller, respectively. The antenna is connected to pins TX1-TX2 and RX1-RX2" of the CR95HF. An LCD is connected to PORTB as in the earlier projects.

If you are using the RFID Click board together with the EasyPIC V7 development board, then configure the RFID board as follows (see the RFID Click board schematic):

Solder the jumpers in position B (to use the UART interface).

Connect the RFID board to mikroBUS 1 connector on the development board.



**Figure 7.29: Circuit Diagram of the Project.**

The connections between the microcontroller and the RFID board are as follows (except the power pins):

RFID Board	Microcontroller
UART_RX/IRQ_IN	RC6
UART_TX/IRQ/OUT	RC7
SSI_0	RA2
SSI_1	RE1

### *CR95HF Operational Modes*

The CR95HF chip operates in two modes: Wait For Event (WFE) mode and Active mode. The WFE mode includes four low-power states: Power-up, Hibernate, Sleep, and Tag Detector. The chip cannot communicate with the external host (e.g. a microcontroller), while in one of these states, it can only be woken-up by the host. In the Active mode, the chip communicates actively with a tag or an external host.

Power-up mode: This mode is entered after power is applied to the chip.

Hibernate mode: In this mode, the chip consumes the lowest power, and it has to be woken up to communicate.

Sleep mode: The chip can be woken up from this state by the Timer, IRQ\_IN pin, or the SPI\_SS pin.

Tag Detector mode: The chip can be woken up from this state by the Timer, IRQ\_IN pin, SPI\_SS pin, or by tag detection.

### *CR95HF Startup Sequence*

After applying power to the chip, the IRQ\_IN pin should be raised after a minimum time of 10 ms. The IRQ\_IN pin should stay high for a minimum of 100 µs. The chip then waits for a low pulse (minimum 10 µs) on the IRQ\_IN pin. The IRQ\_IN pin should then go high for a minimum of 10 ms before the type of communications interface to be used (SPI or UART) is selected and the device is ready to receive commands from the host.

To select the UART interface, both the SSI\_0 and SSI\_1 pins must be low.

### *UART Communication*

The CR95HF default Baud rate is 57,600 bps, although it can be changed by a command if desired. The host sends commands to the CR95HF and waits for replies. A command consists of the following bytes. <CMD> and <LEN> are always 1 byte long, but <DATA> can be from 0 to 255 bytes long:

<CMD><LEN><DATA.....DATA>

**Table 7.1: List of Commands.**

<b>Code</b>	<b>Command</b>	<b>Description</b>
0x01	IDN	Request short information and revision of the CR95HF
0x02	PROTOCOL_SELECT	Select the required RF protocol and its parameters
0x04	SEND_RECV	Send data and receive tag response
0x07	IDLE	Switch CR95HF into the low-power WFE mode, specify the wake-up source, and wait for an event to exit to ready state
0x08	RD_REG	Read wake-up event register or the analog ARC_B register
0x09	WR_REG	Write to ARC_B, Timer window, or the AutoDetect filter enable register (for ISO/IEC 18092 tags)
0x0A	BAUD_RATE	Write the UART baud rate
0x55	ECHO	CR95HF returns echo response (0x55)

Where CMD is the command type, LEN is the length of the command, and DATA are the data bytes. If the LEN field is zero, no data will be sent or received.

The response from the CR95HF is in the following format:

<ResponseCode><LEN><Data....DATA>

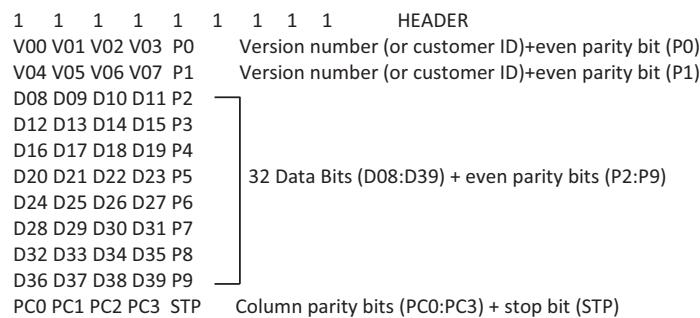
A list of valid commands is given in [Table 7.1](#) (see the CR95HF Data Sheet for further information and command examples). The use of some of these commands is given later in the programming section.

#### *Passive RFID Tags*

Passive RFID tags are available in many shapes, forms, frequencies, and capacities. The format of a very simple popular read-only passive tag, known as EM4100, is given here for reference (this is not the one used in this project). One form of the EM4100 card is shown in [Figure 7.30](#), although they are also available in credit-card shape.



**Figure 7.30: Sample EM4100 RFID Tag.**



**Figure 7.31:** EM4100 Tag Data Format.

The EM4100 tag consists of a 64-bit Read-Only Memory. This means that the tag is configured during the manufacturing process and the data on the tag cannot be changed.

The format of the EM4100 tag is as follows:

- The first 9 bits is all 1 s, and this is the header field.
  - Next, we have 11 groups of 5 bits of data. In the first 10 groups, the first 4 bits are the data bits, while the last bit is the even parity bit. In the last group, the first 4 bits are the column parity bits, while the last bit is the stop bit. The first 8 bits are the version number (or customer ID).

The format of the data is shown in Figure 7.31.

An example tag string with its decoding is given below:

11111111	Header
00000	0
11110	F
00000	0
00000	0
00011	1
00011	1
01010	5
01010	5
01100	6
10100	A
11000	(column parities and stop-bit)

The version number (or customer ID) = 0x0F.

Data string = 0x0011556A.

The tag used in this project is called the MIFARE MF1ICS50, manufactured by NXP. This is the card supplied by mikroElektronika for use with their RFID Click board.

This tag has the following specifications (see manufacturers' data sheet for further information):

- A 13.56-MHz operating frequency,
- A 106-kbps data transfer rate,
- Up to a 100-mm detection range,
- A 1-kbyte electrically erasable programmable read-only memory (EEPROM), organized in 16 sectors with four blocks of 16 bytes each (one block consists of 16 bytes).

The memory on the tag is organized as shown in [Figure 7.32](#). Block 0, Sector 0, is also known as the Manufacturer Block, and the tag serial number is stored in the first 5 bytes of this block (4-byte serial number + Check byte) as shown in [Figure 7.33](#).

### **Project PDL**

The project PDL is given in [Figure 7.34](#).

		Byte Number within a Block															Description
Sector	Block	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	3																Sector Trailer 15
	2																Data
	1																Data
	0																Data
14	3																Sector Trailer 14
	2																Data
	1																Data
	0																Data
:	:																
:	:																
:	:																
1	3																Sector Trailer 1
	2																Data
	1																Data
	0																Data
0	3																Sector Trailer 0
	2																Data
	1																Data
	0																Manufacturer Block

**Figure 7.32: MF1ICS50 Tag Memory Organization.**

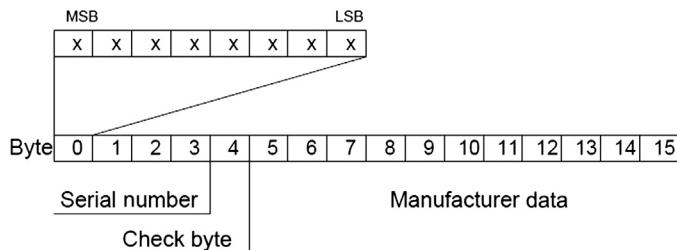


Figure 7.33: Serial Number is the First 4 Bytes.

**Main Program:****BEGIN**

- Define LCD to microcontroller connections
- Configure PORTA,B,C,E as digital
- Configure RA2, RC6, RE1 as outputs
- Initialize LCD
- Configure CR95HF to operate in UART mode
- CALL Initialize\_CR95HF**
- Initialize UART to 57600 Baud
- Display message "RFID" on LCD
- CALL Request\_Short\_Info** to get CR95HF ID
- CALL Select\_Protocol** to select protocol 14443-A
- CALL RAQ** to send RAQ and receive AQTA
- CALL UID** to get tag serial number

**END****BEGIN/Initialize\_CR95HF**

- Send startup initialization sequence to CR95HF chip

**END/Initialize\_CR95HF****BEGIN/Request\_Short\_Info**

- Send command 0x01 to CR95HF
- Get the ID and display on the LCD

**END/Request\_Short\_Info****BEGIN>Select\_Protocol**

- Send command 0x02 to CR95HF to set ISO/IEC 14443-A protocol

**END>Select\_Protocol****BEGIN/RAQ**

- Send command 0x04 to CR95HF with RAQ parameters
- CALL Get\_Data** to receive and display the data

**END/RAQ****BEGIN/UID**

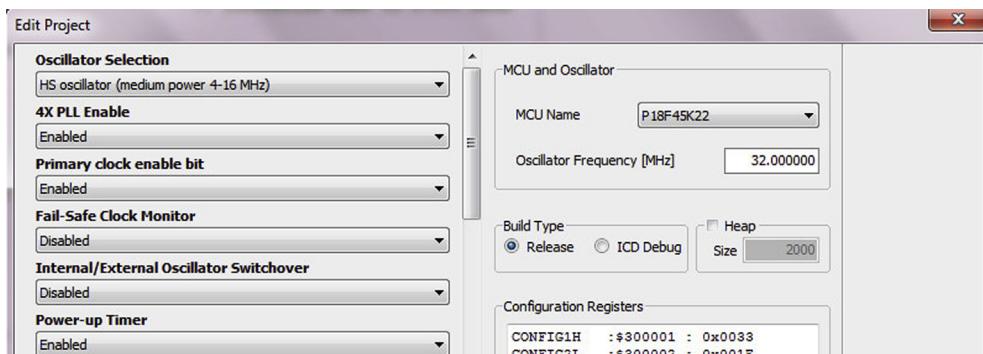
- Send command 0x04 to CR95HF to read the serial number
- CALL Get\_Data** to receive and display the data

**END/UID****BEGIN/Get\_Data**

- Get the data length
- Display the data on the LCD in hex format

**END/Get\_Data**

Figure 7.34: Project PDL.



**Figure 7.35:** Setting the Microcontroller Clock Frequency to 32 MHz.

## Project Program

### mikroC Pro for PIC

The PIC18F45K22 microcontroller is operated from an 8-MHz crystal, but the PLL is enabled and the clock frequency is set to 32 MHz via the Project → Edit Project menu option for faster speed ([Figure 7.35](#)).

The mikroC Pro for PIC program is given in [Figure 7.36](#) (MIKROC-RFID1.C). At the beginning of the program, the connections between the LCD and the microcontroller are defined, symbols IRQ\_IN, SSI\_0 and SSI\_1 are assigned to port pins RC6, RA2 and RE1, respectively. Ports A, B, C, and E are then configured as digital, pins RA2, RC6, RE1 configured as outputs, and the LCD is initialized.

The program then configures the RFID reader to operate in the UART mode by clearing both SSI\_0 and SSI\_1 pins. Function Initialize\_CR95HF is called to initialize the chip as explained earlier in the Startup Sequence. The UART is initialized to operate at 57,600 Baud, which is the default Baud rate of the RFID receiver. The LCD is cleared, and message RFID is displayed at the top row.

Function Request\_Short\_Info requests the ID of the CR95HF by sending command 1 with zero data length. Thus, the command sent to the chip is

0x01	Chip ID request command
0x00	Length of command

The chip returns result code and length of data. The result code must be 0x00 for success. The received result code is checked for validity, and if this is the case, ID data are received from the chip and displayed on the LCD as shown in [Figure 7.37](#). The display consists of the following letters: “NFC FS2JAST2”, which means read only memory (ROM) code revision 2. This text is followed by Cyclic Redundancy Check (CRC) bytes that are not ASCII and cannot be displayed on the LCD.

```
*****  
RFID  
=====  
  
This project is about using an RFID receiver and a passive RFID tag. The project reads the information on the tag and then displays it on the LCD.  
  
An 13.56 MHz CR95HF type RFID reader chip is used in the design. The chip is operated in accordance with the standard ISO/IEC 14443 Type A. The communication between the chip and the microcontroller is established using a standard UART interface. The CR95HF chip is clocked from a 27.12 MHz crystal.  
  
A compatible passive RFID tag is used in the design. The UID of the tag is read, formatted, and then displayed on an LCD. The LCD is connected as in the previous LCD projects.  
  
An 8 MHz crystal is used for the PIC18F45K22 microcontroller in the project. The PLL is enabled so that the effective microcontroller clock rate is X4. i.e. 32 MHz.  
  
Author: Dogan Ibrahim  
Date: October 2013  
File: MIKROC-RFID1.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  
  
#define IRQ_IN LATC.RC6 // IRQ_IN is also the URT TX pin  
#define SSI_0 LATA.RA2 // SSI_0 pin  
#define SSI_1 LATE.RE1 // SSI_1 pin  
  
unsigned char Info[50];  
unsigned char CMD, LEN;  
//  
// This function initializes the chip after power-up so that it is in Ready state. See the  
// CR95HF Data sheet for the delays  
//  
void Initialize_CR95HF()  
{  
    Delay_Ms(10); // t4 delay time
```

Figure 7.36: mikroC Pro for the PIC Program.

```

IRQ_IN = 1;                                // Set IRQ_IN = 1
Delay_us(100);                             // t0 delay time
IRQ_IN = 0;                                 // Lower IRQ_IN
delay_us(500);                            // t2 delay time (typical 250)
IRQ_IN = 1;                                 // IRQ_IN high
Delay_Ms(10);                            // t3 delay time. The CR95HF is Ready now
}

// This function waits until the UART is ready and sends a bytes to it
//
void WriteUart(unsigned char c)
{
    while(Uart1_Tx_Idle() == 0);
    Uart1_Write(c);
}

// This function waits until the UART is ready and reads a byte from it
//
unsigned char ReadUart()
{
    unsigned char c;

    while(Uart1_Data_Ready() == 0);
    c = Uart1_Read();
    return c;
}

// This function requests short info (IDN) about the CR95HF chip. The received information
// is stored in character array called Info. Received short Info is displayed
//
void Request_Short_Info()
{
    unsigned char i, Cnt, flag, c;

    WriteUart(0x01);                           // Send IDN command
    WriteUart(0x00);                           // Send length
    c = ReadUart();                            // Read response
    if(c != 0x00)
    {
        Lcd_Out(1,1,"Error");                // Error, abort
        while(1);
    }

    i = 0;
    flag = 0;
    Cnt = 0;
}

```

**Figure 7.36**  
cont'd

```

while(flag == 0)                                // Read the data bytes
{
    if(Uart1_Data_Ready() == 1)
    {
        Info[i] = Uart1_Read();
        if(i == 0)
        {
            LEN = Info[0];
            Cnt = LEN + 1;
        }
        i++;
        if(i == Cnt)flag = 1;                      // All data read, terminate loop
    }
}

LCD_Cmd(_LCD_FIRST_ROW);                         // To first row of the LCD
for(i=1; i < Cnt; i++)Lcd_Chr_Cp(Info[i]);      // Write short Info to LCD
for(i=0; i < Cnt; i++)Info[i] = 0;                // Clear buffer
}

// This function selects protocol 14443 Type A
//
void Select_Protocol_14443()
{
    unsigned char c;

    WriteUart(0x02);                            // Send Select Protocol command
    WriteUart(0x02);                            // Send length
    WriteUart(0x02);                            // Send protocol 14443 Type A
    WriteUart(0x00);

    // Get result code (must be 0x00)
    //
    c = ReadUart();                            // Get result code
    LEN = ReadUart();                          // get length

    if(c != 0x00)                                // If error, abort
    {
        Lcd_Out(1,1,"Protocol Error");
        while(1);
    }
    else
    {
        Lcd_Out(1,1,"Protocol Selected");
    }
}

// This function reads the data length and then reads all the data and displays on the
// LCD in hexadecimal format

```

**Figure 7.36**  
cont'd

```

// 
void Get_Data()
{
    unsigned char i, Cnt, flag;
    unsigned char Txt[3];

    i = 0;
    flag = 0;
    Cnt = 0;

    while(flag == 0)                                // Read data
    {
        if(Uart1_Data_Ready() == 1)                  // If UART has a character
        {
            Info[i] = Uart1_Read();                 // Get the character
            if(i == 0)
            {
                LEN = Info[0];                      // Get the data length
                Cnt = LEN + 1;
            }
            i++;
            if(i == Cnt)flag = 1;                  // If no more data, exit the loop
        }
    }

    LCD_Cmd(_LCD_FIRST_ROW);                         // Goto first row of LCD
    for(i=1; i < Cnt; i++)
    {
        ByteToHex(Info[i], Txt);                  // Convert to hex
        Lcd_Out_CP(Txt);                         // Display in hex
    }
}

// 
// This function sends RAQ to the card and receives ATAQ response. The response is
// displayed on the LCD
//
void RAQ()
{
    unsigned char i, Cnt, flag, c;
    unsigned char Txt[3];

    WriteUart(0x04);                                // Send IDN command
    WriteUart(0x02);                                // Send length
    WriteUart(0x26);                                // Send length
    WriteUart(0x07);                                // Send length

    c = ReadUart();                                  // Read result code
    if(c != 0x80)
    {
}

```

**Figure 7.36**  
cont'd

```
Lcd_Out(1,1,"Read Error");
while(1);
}

Get_Data();                                // Get data and display it
}

// This function receives the UID of the card and displays on the LCD
//
void UID()
{
    unsigned char i, Cnt, flag, c;
    unsigned char Txt[3];

    WriteUart(0x04);                         // Send IDN command
    WriteUart(0x03);                         // Send length
    WriteUart(0x93);                         // Send length
    WriteUart(0x20);
    WriteUart(0x08);

    c = ReadUart();                           // Get result code
    if(c != 0x80)
    {
        c = ReadUart();                      // Read length
        while(1);
    }

    Get_Data();                                // Get data and display it
}

void main()
{
    ANSELA = 0;                             // Configure PORTA as digital
    ANSELB = 0;                             // Configure PORTB as digital
    ANSELC = 0;                             // Configure PORTC as digital
    ANSELE = 0;                             // Configure PORTE as digital
    TRISA.RA2 = 0;                          // Configure RA2 as output
    IRQ_IN = 1;
    TRISC.RC6 = 0;                          // Configure RC6 as output
    TRISE.RE1 = 0;                          // Configure RE1 as output

    Lcd_Init();                             // Initialize LCD
    Delay_ms(10);

    SSI_0 = 0;                            // Configure CR95HF to use UART
    SSI_1 = 0;                            // Configure CR95HF to se UART
    Initialize_CR95HF();                  // Ready mode after power-up

    UART1_Init(57600);                    // Initialize UART to 57600 Baud
```

**Figure 7.36**  
cont'd

```

Lcd_Cmd(_LCD_CURSOR_OFF);           // Disable cursor
Lcd_Cmd(_LCD_CLEAR);              // Clear LCD
Lcd_Out(1,8,"RFID");             // Display heading on LCD
Delay_Ms(1000);                  // Wait 1 s for the display
Lcd_Cmd(_LCD_CLEAR);
//
// Request short Info (IDN) about the CR95HF chip
//
Request_Short_Info();            // Request short information
Delay_Ms(2000);                  // Wait 2 s
Lcd_Cmd(_LCD_CLEAR);
//
// Select the proocol used (14443 Type A)
//
Select_Protocol_14443();
Delay_Ms(2000);                  // Wait 2 s
Lcd_Cmd(_LCD_CLEAR);
//
// Send RAQ and get ATAQ
//
RAQ();
Delay_ms(1000);
Lcd_Cmd(_LCD_CLEAR);
//
// Read the Tag UID
//
UID();
for(;;)                          // Wait here forever
{
}
}

```

**Figure 7.36**

cont'd

**Figure 7.37: Displaying ID of the CR95HF Chip.**

The next step is to select the RF communication protocol to be used. Function Select\_Protocol\_14443 is called for this purpose. In this example, ISO/IEC 14443-A protocol is used. The command to set a protocol is 0x02. The protocol code for the required protocol is 0x02, and the parameter is 0x00, which is the recommended setting to

send and receive at 106 kbps (see the CR95HF data sheet). Thus, the command sent to the CR95HF is

0x02	Protocol select command
0x02	Command length is 2
0x02	Select ISO/IEC 14443-A
0x00	Command parameter

The chip returns the result code and length of data. The result code must be 0x00 if the required protocol is successfully selected. Selecting a protocol automatically turns ON the electromagnetic field around the RFID board so that tags can be detected.

The next step is to send an RAQ request command and receive the ATQA response from the chip. Command 0x04 (SEND\_RECV) is used for this purpose. The command to send the RAQ is as follows:

0x04	SEND_RECV command
0x02	Command length 2
0x26	RAQ command
0x07	No of significant bits (RAQ is coded on 7 bits)

The tag must be near the receiver when this command is issued. The result code of 0x80 corresponds to success, and if this is the case, the command length and the data bytes (ATAQ) are returned. The card used by the author displayed the following data in hexadecimal format:

0400280000

This data decodes to the following (see ISO/IEC 14443-A specifications):

0400 - ATQA response

Twenty eight in binary is “0001 1000”. This translates as

Bits 0: 3 = Number of significant bits in the first byte (here, 8),

Bit 4: 1 = Parity error, 0 = No parity error (here, no parity error),

Bit 5: 1 = CRC error, 0 = No CRC error (here this bit has no meaning since there is no CRC in AFTA response),

Bit 6: Not used,

Bit 7: 1 = Collision between multiple tags, 0 = No collision (here, no collision)

00 00 - Indexes to where collision detected (no collision here)



**Figure 7.38: Data Returned by the Receiver.**

In the next step, we send a command to read the manufacturers' block, which also contains the serial number. Command 0x04 (SEND\_RECV) is used for this purpose. The command is as follows:

0x04	SEND_RECV command
0x03	Command length 3
0x93	Cascade Level 1 (CL1) command to get the serial number
0x20	Command tail
0x08	Command tail

The result code of 0x80 corresponds to success, and if this is the case, the command length and the data bytes are returned. The card used by the author displayed the following data in hexadecimal format ([Figure 7.38](#)):

AD3D910706280000

The serial number is the first 5 bytes (40 bits), that is, AD3D910706, which corresponds to the binary number “1010 1101 0011 1101 1001 0001 0000 0111 0000 0110”.

#### MPLAB XC8

It is left as an exercise for the reader to convert the program to compile under the MPLAB XC8 compiler.

### **Project 7.4—RFID Lock**

In this project, we shall be using an RFID system to create a security lock. The lock will be based on a relay that will operate when the correct tag is placed near the reader.

The block diagram of the project is shown in [Figure 7.39](#).

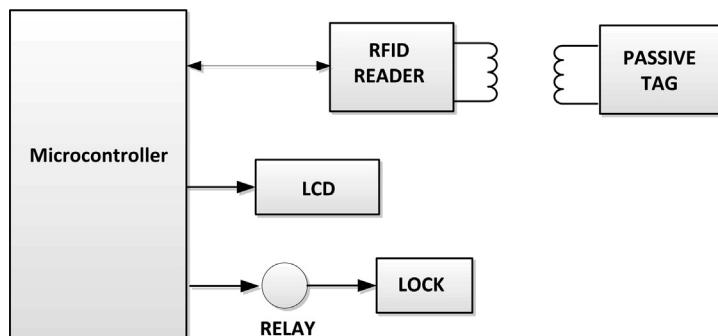


Figure 7.39: Block Diagram of the Project.

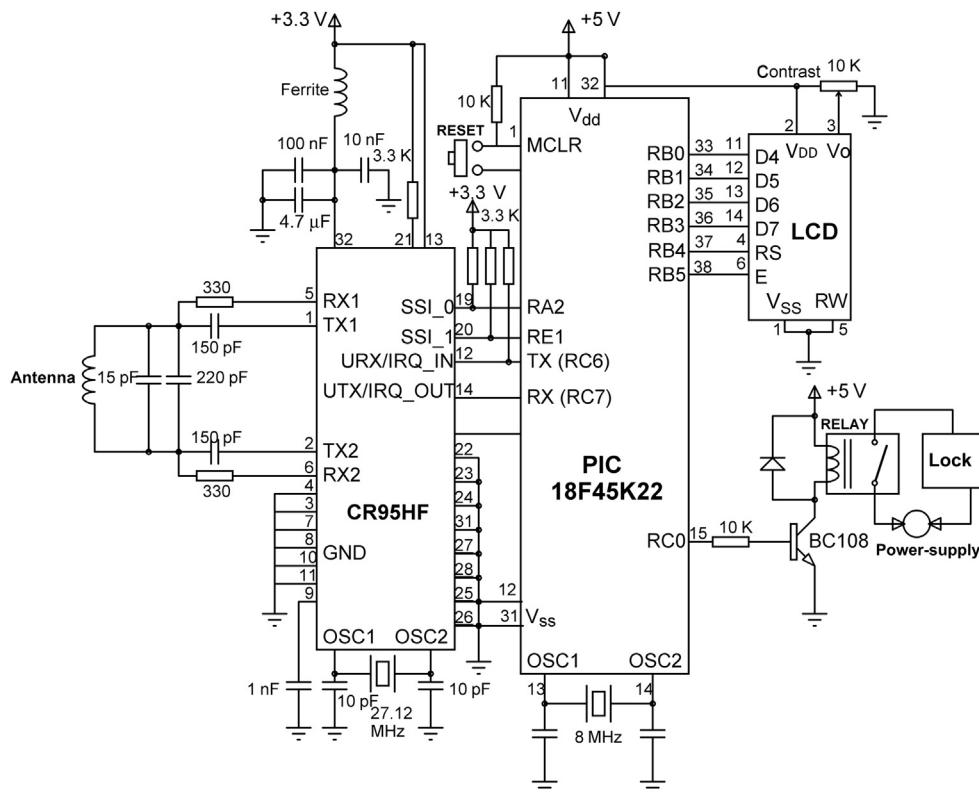


Figure 7.40: Circuit Diagram of the Project.

### Project Hardware

The circuit diagram of the project is very similar to the one given in Figure 7.26, except that here additionally a relay and a lock mechanism are connected to the RC0 pin of the microcontroller via a transistor switch. Figure 7.40 shows the complete circuit diagram.

## **Project PDL**

The project PDL is shown in [Figure 7.41](#).

## **Project Program**

### *mikroC Pro for PIC*

The program listing of the project is given in [Figure 7.42](#) (MIKROC-RFID2.C). In this project, it is assumed that the tag used in the previous project is used to open the lock. The program is similar to the one given in [Figure 7.36](#). Additionally, at the beginning of the program, symbol RELAY is assigned to port RC0. Also, there is no need to read and display the ID of the CR95HF chip.

The program selects the required protocol (ISO/IEC 14443-A), displays the message READY on the LCD, and then enters an endless loop formed by a *while* statement. Inside this loop, the RAQ request is sent with no display of the AQAT. The serial number of the tag is read and compared with what was read in the earlier project (i.e. AD3D9107). If this is the matching tag, then the relay is energized by setting RELAY = 1, message “Opened” is sent to the LCD, and the program stays in this state for 5 s. After this time, the relay is deenergized, and message “Ready...” is displayed on the LCD to inform the user that the system is ready again.

## **Project 7.5—Complex SPI Bus Project**

In Chapter 8, we have seen briefly how to use the SPI bus to generate various waveforms using a DAC. In this project, we will look at the operation of the SPI bus in greater detail as it is very important in the design of microcontroller-based systems. We will also develop a project to measure the temperature using an SPI-based temperature sensor. The ambient temperature will be measured and then displayed on an LCD.

### ***The Master Synchronous Serial Port Module***

The Master Synchronous Serial Port (MSSP) module is a serial interface module on PIC18F series of microcontrollers, used for communicating with other serial devices such as EEPROMs, display drivers, A/D converters, D/A converters, and SD cards. PIC18F45K22 microcontroller has two built-in MSSP modules.

The MSSP module can operate in one of two modes:

- SPI,
- Interintegrated Circuit ( $I^2C$ ).

**Main Program:**

```
BEGIN
    Define LCD to microcontroller connections
    Configure PORTA,B,C,E as digital
    ConfigureRA2, RC6, RE1,RC0 as outputs
    Initialize LCD
    Configure CR95HF to operate in UART mode
    CALL Initialize_CR95HF
    Initialize UART to 57600 Baud
    Display message "RFID LOCK" on LCD
    CALL Select_Protocol to select protocol 14443-A
    Display "READY" on LCD
    DO FOREVER
        CALL RAQ to send RAQ and receive AQTA
        CALL UID to get tag serial number
        IF the tag matches the serial number
            Energize the relay
            Display "Opened" on LCD
            Wait for 5 seconds
            De-energize the relay
        ELSE
            Display "Ready..." on LCD to try again
        ENDIF
    ENDDO
END
```

```
BEGIN/Initialize_CR95HF
    Send startup initialization sequence to CR95HF chip
END/Initialize_CR95HF
```

```
BEGIN>Select_Protocol
    Send command 0x02 to CR95hF to set ISO/IEC 14443-A protocol
END>Select_Protocol
```

```
BEGIN/RAQ
    Send command 0x04 to CR95HF with RAQ parameters
    CALL Get_Data to receive and display the data
END/RAQ
```

```
BEGIN/UID
    Send command 0x04 to CR95HF to read the serial number
    CALL Get_Data to receive and display the data
END/UID
```

```
BEGIN/Get_Data
    Get the data length
    Display the data on the LCD in hex format
END/Get_Data
```

**Figure 7.41: Project PDL.**

```
*****
RFID LOCK
=====

This project is about using an RFID receiver and a passive RFID tag to operate a relay to open a lock.

An 13.56 MHz CR95HF type RFID reader chip is used in the design, operated in accordance with the standard ISO/IEC 14443 Type A. The communication between the chip and the microcontroller is established using a standard UART interface. The CR95HF chip is clocked from a 27.12 MHz crystal.

A compatible passive RFID tag is used in the design. The UID of the tag is read and if it is an acceptable tag then the relay is operated to open the lock. The relay operates for 5 s to open the lock and then stops. The program then waits for other activations.

An LCD is used to display various messages about the operation of the lock. The relay is connected To RCO pin of the microcontroller via a transistor switch. The lock is assumed to operate when the Relay is energized.

An 8 MHz crystal is used for the PIC18F45K22 microcontroller in the project. The PLL is enabled so that the effective microcontroller clock rate is X4. i.e. 32 MHz.

Author: Dogan Ibrahim
Date: September 2013
File: MIKROCC-RFID2.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

#define IRQ_IN LATC.RC6 // IRQ_IN is also the URT TX pin
#define SSI_0 LATA.RA2 // SSI_0 pin
#define SSI_1 LATE.RE1 // SSI_1 pin
#define RELAY LATC.RCO // Relay

unsigned char Info[50];
unsigned char CMD, LEN, ErrorFlag, ID[30];
//
// This function initializes the chip after power-up so that it is in Ready state
// See the CR95HF Data sheet for the delays
```

**Figure 7.42: mikroC Pro for PIC Program.**

```
//  
void Initialize_CR95HF()  
{  
    Delay_Ms(10);                                // t4 delay time  
    IRQ_IN = 1;                                    // Set IRQ_IN = 1  
    Delay_us(100);                                // t0 delay time  
    IRQ_IN = 0;                                    // Lower IRQ_IN  
    delay_us(500);                                // t2 delay time (typical 250)  
    IRQ_IN = 1;                                    // IRQ_IN high  
    Delay_Ms(10);                                // t3 delay time. The CR95HF is Ready now  
}  
  
//  
// This function waits until the UART is ready and sends a byte to it  
//  
void WriteUart(unsigned char c)  
{  
    while(Uart1_Tx_Idle() == 0);  
    Uart1_Write(c);  
}  
  
//  
// This function waits until the UART is ready and reads a byte from it  
//  
unsigned char ReadUart()  
{  
    unsigned char c;  
  
    while(Uart1_Data_Ready() == 0);  
    c = Uart1_Read();  
    return c;  
}  
  
//  
// This function selects protocol 14443 Type A  
//  
void Select_Protocol_14443()  
{  
    unsigned char c;  
  
    WriteUart(0x02);                             // Send Select Protocol command  
    WriteUart(0x02);                             // Send length  
    WriteUart(0x02);                             // Send protocol 14443 Type A  
    WriteUart(0x00);  
  
//  
// Get result code (must be 0x00)  
//  
    c = ReadUart();                            // Get result code  
    LEN = ReadUart();                           // get length
```

**Figure 7.42**  
cont'd

```

if(c != 0x00)                                // If error, abort
{
    Lcd_Out(1,1,"Protocol Error");           // Error. Reset the device
    while(1);
}
else Lcd_Out(1,1,"Protocol Set ");
Delay_Ms(1000);
}

// This function reads the data length and then reads all the data and displays
// on the LCD in hexadecimal format
//
void Get_Data()
{
    unsigned char i, Cnt, flag;
    unsigned char Txt[3];

    i = 0;
    flag = 0;
    Cnt = 0;

    while(flag == 0)                         // Read data
    {
        if(Uart1_Data_Ready() == 1)           // If UART has a character
        {
            Info[i] = Uart1_Read();          // Get the character
            if(i == 0)
            {
                LEN = Info[0];              // Get the data length
                Cnt = LEN + 1;
            }
            i++;
            if(i == Cnt)flag = 1;           // If no more data, exit the loop
        }
    }

    ID[0] = 0x0;
    for(i=1; i < Cnt; i++)
    {
        ByteToHex(Info[i], Txt);          // Convert to hex
        strcat(ID, Txt);                 // Append to ID
    }
}

// This function sends RAQ to the card and receives ATAQ response. The response
// is displayed on the LCD
//

```

**Figure 7.42**  
cont'd

```
void RAQ()
{
    unsigned char i, Cnt, flag, c;
    unsigned char Txt[3];

    WriteUart(0x04);                                // Send IDN command
    WriteUart(0x02);                                // Send length
    WriteUart(0x26);                                // Send length
    WriteUart(0x07);                                // Send length

    c = ReadUart();                                  // Read result code
    ErrorFlag = 0;                                   // Assume no error

    if(c != 0x80)                                    // If error
    {
        c = ReadUart();                            // Read length
        ErrorFlag = 1;                            // Set error flag
        return;
    }

    Get_Data();                                     // Get data
}

// This function receives the UID of the card and displays on the LCD
//
void UID()
{
    unsigned char i, Cnt, flag, c;
    unsigned char Txt[3];

    WriteUart(0x04);                                // Send IDN command
    WriteUart(0x03);                                // Send length
    WriteUart(0x93);                                // Send length
    WriteUart(0x20);
    WriteUart(0x08);

    c = ReadUart();                                  // Get result code
    if(c != 0x80)
    {
        ErrorFlag = 1;                            // error flag
        c = ReadUart();                            // Read length
        return;
    }

    Get_Data();                                     // Get data and display it
}

void main()
{
```

**Figure 7.42**  
cont'd

```

ANSELA = 0;                                // Configure PORTA as digital
ANSELB = 0;                                // Configure PORTB as digital
ANSELC = 0;                                // Configure PORTC as digital
ANSELE = 0;                                // Configure PORTE as digital
TRISA.RA2 = 0;                             // Configure RA2 as output
IRQ_IN = 1;                                 // Configure RC6 as output
TRISC.RC6 = 0;                             // Configure RE1 as output
TRISE.RE1 = 0;                            // Configure RC0 as output
TRISC.RC0 = 0;                             // Relay OFF to start with
RELAY = 0;                                  // Initialize LCD
Delay_ms(10);

SSI_0 = 0;                                // Configure CR95HF to use UART
SSI_1 = 0;                                // Configure CR95HF to se UART
Initialize_CR95HF();                      // Ready mode after power-up

UART1_Init(57600);                         // Initialize UART to 57600 Baud

Lcd_Cmd(_LCD_CURSOR_OFF);                  // Disable cursor
Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD
Lcd_Out(1,1,"RFID LOCK");                // Display heading on LCD
Delay_Ms(2000);                           // Wait 1 s for the display
Lcd_Cmd(_LCD_CLEAR);
//
// Select the proocol used (14443 Type A)
//
Select_Protocol_14443();
Delay_Ms(2000);                           // Wait 2 s
Lcd_Cmd(_LCD_CLEAR);
Lcd_Out(1,1,"READY");                    // Display "READY" message

//
// This is the tag detection and relay energization part of the program
//
while(1)
{
    RAQ();                                  // Send RAQ and get ATAQ
    UID();                                  // get serial number

    if(ErrorFlag == 0)                      // If no errors in RAQ and UID
    {
        if(ID[0] == 'A' && ID[1] == 'D' && ID[2] == '3' && ID[3] == 'D' &&
           ID[4] == '9' && ID[5] == '1' && ID[6] == '0' && ID[7] == '7')      // If tag matches ?
        {
            Lcd_Cmd(_LCD_CLEAR);          // Display "Opened" message
            Lcd_Out(1,1,"Opened");
            RELAY = 1;                   // Energize the relay
            Delay_Ms(5000);             // Wait for 5 s
            RELAY = 0;                   // de-energize the relay
        }
    }
}

```

**Figure 7.42**

cont'd

```
        Lcd_Cmd(_LCD_CLEAR);           // Clear LCD
    }
}
else
{
    Lcd_Out(1,1,"Ready...");      // Error detected. Ready to try again
}
Delay_Ms(500);                  // Wait before re-try
}
}
```

**Figure 7.42**  
cont'd

Both SPI and I<sup>2</sup>C are serial bus protocol standards. The SPI protocol was initially developed and proposed by Motorola for use in microprocessor and microcontroller-based interface applications. In a system that uses the SPI bus, one device acts as the master and other devices act as slaves. The master initiates a communication and also provides clock pulses to the slave devices.

The I<sup>2</sup>C is a two-wire bus and was initially developed by Philips for use in low-speed microprocessor-based communication applications.

In this project, we shall be looking at the operation of the MSSP module in the SPI mode.

### ***MSSP in the SPI Mode***

The SPI mode allows 8 bits of data to be synchronously transmitted and received simultaneously. In the master mode, the device uses three signals, and in the slave mode, a fourth signal is used. In this project, we shall be looking at how to use the MSSP module in the master mode since in most microcontroller applications the microcontroller is the master device.

In the master mode, the following pins of the PIC18F45K22 microcontroller are used for SPI1 interface:

- Serial data out (SDO1) -Pin RC5
- Serial data in (SDI1) -Pin RC4
- Serial clock (SCK1) -Pin RC3

And for SPI2 interface:

- Serial data out (SDO2) -Pin RD4
- Serial data in (SDI2) -Pin RD1
- Serial clock (SCK2) -Pin RD0

Figure 7.43 shows the block diagram of the PIC18F45K22 microcontroller MSSP module when operating in the SPI mode.

### SPI Mode Registers

The MSSP module has three registers when operating in the SPI master mode, ‘x’ is 1 or 2 and corresponds to the SPI module used:

- MSSP control register (SSPxCON1),
- MSSP status register (SSPxSTAT),
- MSSP receive/transmit buffer register (SSPxBUF),
- MSSP shift register (SSPxSR, not directly accessible).

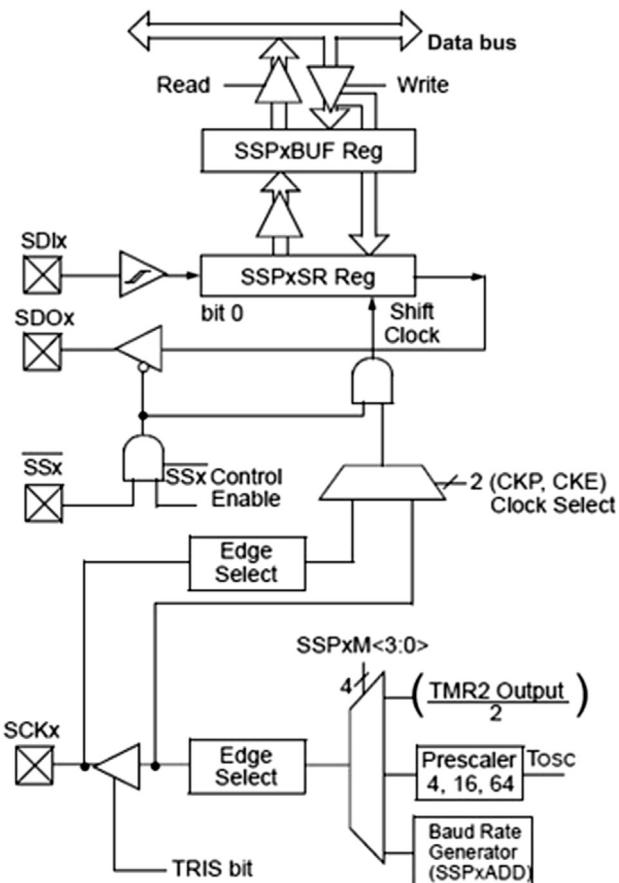


Figure 7.43: MSSP Module in the SPI Mode.

### SSPxSTAT

This is the status register with the lower 6 bits read only and the upper two bits read/write. [Figure 7.44](#) shows the bit definitions of this register. Only bits 0, 6, and 7 are related to operation in the SPI mode. Bit 7 (SMP) allows the user to select the input data sample time. When  $SMP = 0$ , input data are sampled at the middle of data output time, and when  $SMP = 1$ , the sampling is done at the end. Bit 6 (CKE) allows the user to select the transmit clock edge. When  $CKE = 0$ , transmit occurs on transition from the idle to active clock state, and when  $CKE = 1$ , transmit occurs on transition from the active to the idle clock state. Bit 0 (BF) is the buffer full status bit. When  $BF = 1$ , receive is complete (i.e. SSPxBUF is full), and when  $BF = 0$ , receive is not complete (i.e. SSPxBUF is empty).

### SSPxCON1

This is the control register ([Figure 7.45](#)) used to enable the SPI mode and to set the clock polarity and the clock frequency. In addition, the transmit collision detection (bit 7), and receive overflow detection (bit 6) are indicated by this register.

### *Operation in the SPI Mode*

[Figure 7.46](#) shows a simplified block diagram with a master and a slave device communicating over the SPI bus. In this diagram, SPI module 1 is used. The SDO1 output of the master device is connected to the SDI1 input of the slave device, and the SDI1 input of the master device is connected to the SDO1 output of the slave device. The clock SCK1 is derived by the master device. The data communication is as follows:

SMP	CKE	D/A	P	S	R/W	UA	BF
7	6	5	4	3	2	1	0

Bit 7 **SMP**: SPI Data Input Sample bit  
 1 = Input data sampled at the end of data output time  
 0 = Input data sample at middle of data output time

Bit 6 **CKE**: SPI Clock Edge Select bit  
 1 = Transmit occurs on transition from active to idle clock state  
 0 = Transmission occurs on transition from idle to active clock state

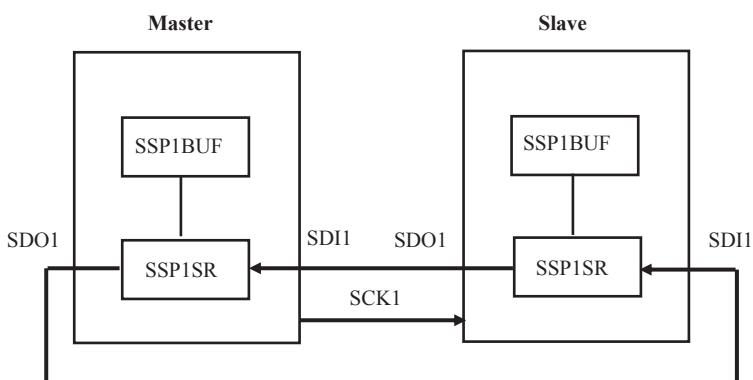
Bit 0 **BF**: Buffer Full Status bit  
 1 = Receive complete SSPxBUF is full  
 0 = Receive not complete SSPxBUF is empty

**Figure 7.44: SSPxSTAT Register Bit Configuration. Only the Bits when Operating in the SPI Master Mode are Described.**

WCOL	SSPxOV	SSPxEN	CKP	SSPxM			
7	6	5	4	3	2	1	0

- Bit 7 **WCOL:** Write Collision detect bit  
1 = Collision detected  
0 = No collision
- Bit 6 **SSPxOV:** Receive Overflow Indicator bit  
1 = Overflow occurred (can only occur in slave mode)  
0 = No overflow
- Bit 5 **SSPxEN:** Synchronous Serial Port Enable bit  
1 = Enables serial port and configures SCKx,SDOx,SDIx as the source of port pins  
0 = Disables serial port and configures these pins as I/O pins
- Bit 4 **CKP:** Clock Polarity Select bit  
1 = Idle state for clock is a high level  
0 = Idle state for clock is a low level
- Bit 3-0 **SSPxM:** Synchronous Serial Port Mode Select bits  
0000 = SPI Master mode, clock =  $F_{osc}/4$   
0001 = SPI Master mode, clock =  $F_{osc}/16$   
0010 = SPI Master mode, clock =  $F_{osc}/64$   
0011 = SPI Master mode, clock = TMR2 output/2  
0100 = SPI Master mode, clock = SCKx pin, SSx pin control enabled  
0101 = SPI Master mode, clock = SCKx pin, SSx pin control disabled  
1010 = SPI Master mode, clock =  $F_{osc}/4*(SSPxADD+1)$

**Figure 7.45: SSPxCON1 Register Bit Configuration. Only the Bits when Operating in the SPI Master Mode are Described.**



**Figure 7.46: A Master and a Slave Device on the SPI Bus.**

### ***Sending Data to the Slave***

To send data from the master to the slave, the master writes the data byte into its SSP1BUF register. This byte is also written automatically into the SSP1SR register of the master. As soon as a byte is written into the SSP1BUF register, eight clock pulses are sent out from the master SCK1 pin and at the same time the data bits are sent out from the master SSP1SR into the slave SSP1SR, that is, the contents of master and slave SSP1SR registers are swapped. At the end of this data transmission, the SSP1IF flag (PIR1 register) and the BF flag (SSP1STAT) will be set to show that the transmission is complete. Care should be taken not to write a new byte into SSP1BUF before the current byte is shifted out, otherwise an overflow error will occur (indicated by bit 7 of SSP1CON1).

### ***Receiving Data From the Slave***

To receive data from the slave device, the master has to write a “dummy” byte into its SSP1BUF register to start the clock pulses to be sent out from the master. The received data are then clocked into SS1PSR of the master, bit by bit. When the complete 8 bits are received, the byte is transferred to the SSP1BUF register and flags SSP1IF and BF are set. It is interesting to note that the received data are double buffered.

### ***Configuration of MSSP for the SPI Master Mode***

The following MSSP parameters for the master device must be set up before the SPI communication can take place successfully (SPI module 1 is assumed):

- Set data clock rate,
- Set clock edge mode,
- Clear bit 5 of TRISC (i.e. SDO1 = RC5 is output),
- Clear bit 3 of TRISC (i.e. SCK1 = RC3 is output),
- Enable the SPI mode.

Note that the SDI1 pin (pin RC4) direction is automatically controlled by the SPI module.

### ***Data Clock Rate***

The clock is derived by the master, and the clock rate is user programmable to one of the following values via the SSP1CON1 register bits 0–3:

- Fosc/4,
- Fosc/16,
- Fosc/64,
- Timer2 output/2.

## Clock Edge Mode

The clock edge is user programmable via register SSP1CON1 (bit 4) and the user can either set the clock edge as *idle high* or *idle low*. In the idle high mode, the clock is high when the device is not transmitting, and in the idle low mode the clock is low when the device is not transmitting. Data can be transmitted either at the rising or the falling edge of the clock. The CKE bit of SSP1STAT (bit 6) is used to select the clock edge.

## Enabling the SPI Mode

Bit 5 of SSP1CON1 must be set to enable the SPI mode. To reconfigure the SPI parameters, this bit must be cleared, SPI mode configured, and then the SSPEN bit set back to one.

An example is given below to demonstrate how to set the SPI parameters.

### Example 7.1

It is required to operate the MSSP module 1 of a PIC18F45K22 microcontroller in the SPI mode. The data should be shifted on the rising edge of the clock and the SCK1 signal must be idle low. The required data rate is at least 1 Mbps. Assume that the microcontroller clock rate is 16 MHz and that the input data are to be sampled at the middle of data output time. What should be the settings of the MSSP registers?

#### Solution 7.1

*Register SSP1CON1 should be set as follows*

- Clear bits 6 and 7 of SSP1CON1 (i.e. no collision detect and no overflow);
- Clear bit 4 to 0 to select idle low for the clock;
- Set bits 0 through 3 to 0000 or 0001 to select the clock rate to Fosc/4 (i.e. 16/4 = 4 Mbps data rate), or Fosc/16 (i.e. 16/16 = 1 Mbps);
- Set bit 5 to enable the SPI mode.

Thus, register SSP1CON1 should be set to the following bit pattern:

00 1 0 0000 i.e. 0x20

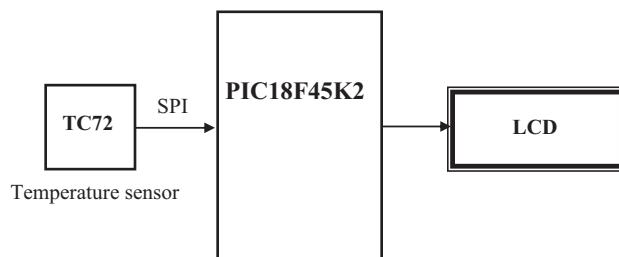
*Register SSP1STAT should be set as follows:*

- Clear bit 7 to sample the input data at the middle of data output time;
- Clear bit 6–0 to transmit data on the rising edge (low to high) of the SCK1 clock;
- Bits 5–0 are not used in the SPI mode.

Thus, register SSP1STAT should be set to the following bit pattern:

0 0 0 00000 i.e. 0x00

Figure 7.47 shows the block diagram of the project. The temperature sensor TC72 is used in this project. The sensor is connected to the SPI bus pins of a PIC18F45K22 microcontroller. In addition, the microcontroller is connected to an LCD to display the temperature.



**Figure 7.47: Block Diagram of the Project.**

The specifications and operation of the TC72 temperature sensor are described below in detail.

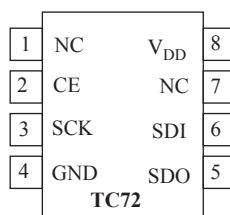
### **TC72 Temperature Sensor**

The TC72 is an SPI bus compatible digital temperature sensor IC that is capable of reading temperatures from  $-55$  to  $+125$  °C.

The device has the following features

- SPI bus compatible,
- A 10-bit resolution with  $0.25$  °C/bit,
- A  $\pm 2$  °C accuracy from  $-40$  to  $+85$  °C,
- A 2.65- to 5.5-V operating voltage,
- A  $250\text{-}\mu\text{A}$  typical operating current,
- A  $1\text{-}\mu\text{A}$  shutdown operating current,
- Continuous and one-shot operating modes.

The pin configuration of TC72 is shown in Figure 7.48. The device is connected to an SPI bus via standard SPI bus pins SDI, SDO, and SCK. Pin CE is the chip-enable pin and is used to select a particular device in multiple TC72 applications. CE must be logic 1 for the device to be enabled. The device is disabled (output in tristate mode) when CE is logic 0.



**Figure 7.48: TC72 Pin Configuration.**

The TC72 can operate either in the *one-shot* mode or in the *continuous* mode. In the one-shot mode, the temperature is read after a request is sent to read the temperature. In the continuous mode, the device measures the temperature approximately every 150 ms.

Temperature data are represented in 10-bit two's complement format with a resolution of 0.25 °C/bit. The converted data are available in two 8-bit registers. The most significant bit (MSB) register stores the decimal part of the temperature, whereas the least significant bit (LSB) register stores the fractional part. Only bits 6 and 7 of this register are used. The format of these registers is shown below:

MSB	S	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
LSB		$2^{-1}$	$2^{-2}$	0	0	0	0	0

Where S is the sign bit. An example is given below.

### Example 7.2

The MSB and LSB settings of a TC72 are as follows:

MSB: 00101011

LSB: 10000000

Find the temperature read.

### Solution 7.2

The temperature is found to be

$$\text{MSB} = 2^5 + 2^3 + 2^1 + 2^0 = 43,$$

$$\text{LSB} = 2^{-1} = 0.5$$

Thus, the temperature is 43.5 °C.

Table 7.2 shows sample temperature output data of the TC72 sensor.

**Table 7.2: TC72 Temperature Output Data.**

Temperature (°C)	Binary (MSB/LSB)	Hex
+125	0111 1101/0000 0000	7D00
+74.5	0100 1010/1000 0000	4A80
+25	0001 1001/0000 0000	1900
+1.5	0000 0001/1000 0000	0180
+0.5	0000 0000/1000 0000	0080
+0.25	0000 0000/0100 0000	0040
0	0000 0000/0000 0000	0000
-0.25	1111 1111/1100 0000	FFC0
-0.5	1111 1111/1000 0000	FF80
-13.25	1111 0010/1100 0000	F2C0
-25	1110 0111/0000 0000	E700
-55	1100 1001/0000 0000	C900

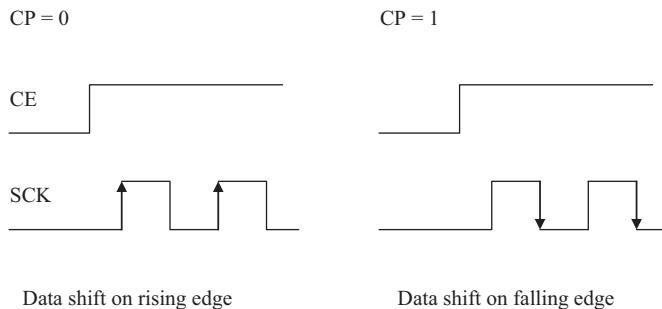


Figure 7.49: Serial Clock Polarity.

### TC72 Read/Write Operations

The SDI input writes data into TC72's control register, while SDO outputs the temperature data from the device. The TC72 can operate using either the rising or the falling edge of the clock (SCK). The clock idle state is detected when the CE signal goes high. As shown in Figure 7.49, the clock polarity (CP) determines whether data are transmitted on the rising or the falling clock edge.

The maximum clock frequency (SCK) of TC72 is specified as 7.5 MHz. Data transfer consists of an address byte, followed by one or more data bytes. The most significant bit (A7) of the address byte determines whether a read or a write operation will occur. If A7 = 0, one or more read cycles will occur; otherwise, if A7 = 1, one or more write cycles will occur. The multibyte read operation will start by writing to the highest desired register and then reading from high to low addresses. For example, the temperature high byte address can be sent with A7 = 0 and then the result high byte, low byte, and the control register can be read as long as the CE pin is held active (CE = 1).

The procedure to read temperature from the device is as follows (assuming SPI module 1 is used):

- Configure the microcontroller SPI bus for the required clock rate and clock edge.
- Enable TC72 by setting CE = 1.
- Send temperature result high byte read address (0x02) to the TC72 (Table 7.3).

Table 7.3: TC72 Internal Registers.

Register	Read Address	Write Address	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Control	0x00	0x80	0	0	0	OS	0	1	0	SHDN
LSB temperature	0x01	N/A	T1	T0	0	0	0	0	0	0
MSB temperature	0x02	N/A	T9	T8	T7	T6	T5	T4	T3	T2
Manufacturer ID	0x03	N/A	0	1	0	1	0	1	0	0

- Write a “dummy” byte into the SSP1BUF register to start eight pulses to be sent out from the SCK1 pin and then read the temperature result high byte.
- Write a “dummy” byte into the SSP1BUF register to start eight pulses to be sent out from the SCK1 pin and then read the temperature low byte.
- Set CE = 0 to disable the TC72 so that a new data transfer can begin.

### ***Internal Registers of the TC72***

As shown in [Table 7.3](#), the TC72 has four internal registers: Control register, LSB temperature register, MSB temperature register, and the Manufacturer ID register.

#### ***Control Register***

This is a read and write register used to select the mode of operation as shutdown, continuous, or one-shot. The address of this register is 0x00 when reading, and 0x80 when writing to the device. [Table 7.4](#) shows how different modes are selected. At power-up, the shutdown bit (SHDN) is set to 1 so that the device is in the shutdown mode at startup and the device is used in this mode to minimize the power consumption.

A temperature conversion is initiated by a write operation to the Control register to select either the continuous mode or the one-shot mode. The temperature data will be available in the MSB and LSB registers after about 150 ms of the write operation. The one-shot mode performs a single temperature measurement after which time the device returns to the shutdown mode. In the continuous mode, new temperature data are available at 159-ms intervals.

#### ***LSB and MSB Registers***

These are read-only registers that contain the 10-bit measured temperature data. The address of the MSB register is 0x02, and LSB register is 0x01.

#### ***Manufacturer ID***

This is a read-only register with address 0x03. This register identifies the device as a temperature sensor, returning 0x054.

**Table 7.4: Selecting the Mode of Operation.**

Operating Mode	One Shot (OS)	Shutdown (SHDN)
Continuous	X	0
Shutdown	0	1
One-shot	1	1

## Project Hardware

The circuit diagram of the project is shown in [Figure 7.50](#). The TC72 temperature sensor is connected to the SPI bus pins of a PIC18F45K22 microcontroller, which is operated from a 4-MHz crystal. The CE pin of the TC72 is controlled from pin RC0 of the microcontroller. An LCD is connected to PORTB of the microcontroller as follows:

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

The connection between the TC72 and the microcontroller are as follows:

Microcontroller	TC72
RC0	CE
RC3	SCK
RC4	SDO
RC5	SDI

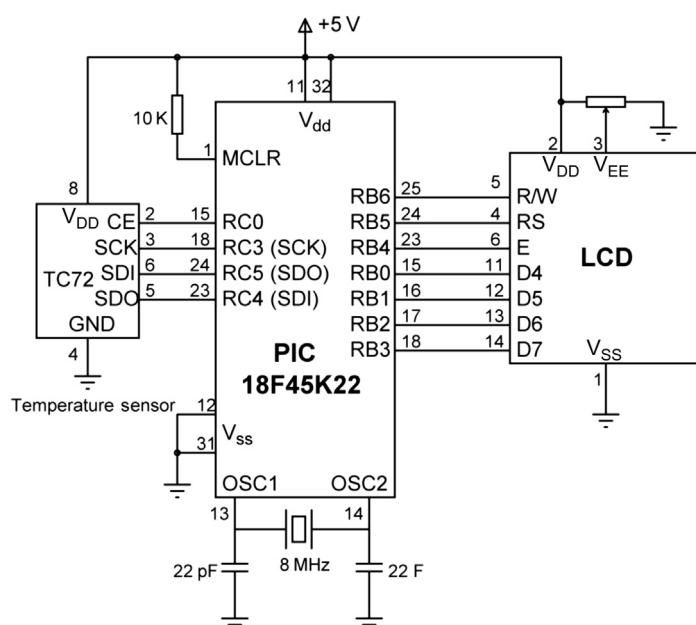


Figure 7.50: Circuit Diagram of the Project.

The microcontroller sends control commands to the TC72 sensor to initiate temperature conversions every second. The temperature data are then read and displayed on the LCD.

### ***The Program***

#### **MPLAB XC8**

The MPLAB XC8 program listing of the project is shown in [Figure 7.51](#) (XC8-SPI.C). The program reads the temperature from the TC72 sensor and displays on the LCD every second. In this version of the program, only the positive temperatures and only the integer part are displayed.

The program consists of a number of functions. Some functions used in the program are

**Init\_LCD:** This function initializes the LCD to 4-bit operation with  $5 \times 7$  characters. The function also calls LCD\_Clear to clear the LCD screen.

**Init\_SPI:** This function initializes the microcontroller SPI bus to:

- Clock rate: Fosc/4 (i.e. 1 MHz)
- Clock Idle Low, Shift ta on clock falling edge
- Input data sample at end of data out

**Send\_To\_TC72:** This function loads a byte to SPI register SSP1BUF and then waits until the data are shifted out.

**Read\_Temperature:** This function communicates with the TC72 sensor to read the temperature. The following operations are performed by this function:

1. Enable TC72 (CE = 1, for single byte write),
2. Send Address 0x80 (A7 = 1),
3. Clear BF flag,
4. Send One-Shot command (Control = 0001 0001),
5. Disable TC72 (CE = 0, end of single byte write),
6. Clear BF flag,
7. Wait at least 150 ms for the temperature to be available,
8. Enable TC72 (CE = 1, for multiple data transfer),
9. Send Read MSB command (Read address = 0x02),
10. Clear BF flag,
11. Send dummy output to start clock and read data (Send 0x00),
12. Read high temperature into variable MSB,
13. Send dummy output to start clock and read data (Send 0x00),
14. Read low temperature into variable LSB,
15. Disable TC72 data transfer (CE = 0),
16. Copy high result into variable “result”.

```
*****
          SPI BUS BASED DIGITAL THERMOMETER
*****
```

In this project a TC72 type SPI bus based temperature sensor IC is used. The IC is connected to the SPI bus pins of a PIC18F45K22 type microcontroller (i.e. to pins RC3=SCK, RC4=SDI, and RC5=SDO) and the microcontroller is operated from an 8 MHz crystal.

In addition PORT B pins of the microcontroller are connected to a standard LCD.

The microcontroller reads the temperature every second and displays on the LCD as a positive number (fractional part of the temperature, or negative temperatures are not displayed in this version of the program).

An example display is:

23

```
Author: Dogan Ibrahim
Date: October 2013
File: XC8-SPI.C
*****/
#include <xc.h>
#include <string.h>
#include <plib/usart.h>
#include <plib/xlcd.h>
#include <plib/spi.h>
#include <stdlib.h>

#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

#define CE PORTCbits.RC0
#define Ready SSPSTATbits.BF

unsigned char LSB, MSB;
int result;

// 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);
    }
}
```

**Figure 7.51: MPLAB XC8 Program.**

```
//  
// This function creates milliseconds delay. The argument specifies the delay time in ms  
//  
void Delay_Ms(unsigned int ms)  
{  
    unsigned int i;  
  
    for(i = 0; i < ms; i++) __delay_ms(1);  
}  
  
//  
// 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);  
}
```

**Figure 7.51**  
cont'd

```
}

//  
// Initialize the LCD, clear and home the cursor  
//  
void Init_LCD(void)  
{  
    OpenXLCD(FOUR_BIT & LINE_5X7);           // 8 bit, 5x7 character  
    LCD_Clear();                            // Clear LCD  
}

//  
// Initialize the SPI bus  
//  
void Init_SPI(void)  
{  
    OpenSPI(SPI_FOSC_4, MODE_01, SMPEND);      // SPI clk = 2MHz  
}

//  
// This function sends a control byte to the TC72 and waits until the  
// transfer is complete  
//  
void Send_To_TC72(unsigned char cmd)  
{  
    SSPBUF = cmd;                           // Send control to TC72  
    while(!Ready);                          // Wait until data is shifted out  
}

//  
// This function reads the temperature from the TC72 sensor  
//  
// Temperature data is read as follows:  
//  
// 1. Enable TC72 (CE=1, for single byte write)  
// 2. Send Address 0x80 (A7=1)  
// 3. Clear BF flag  
// 4. Send One-Shot command (Control = 0001 0001)  
// 5. Disable TC72 (CE=0, end of single byte write)  
// 6. Clear BF flag  
// 7. Wait at least 150ms for temperature to be available  
// 8. Enable TC72 (CE=1, for multiple data transfer)  
// 9. Send Read MSB command (Read address=0x02)  
// 10. Clear BF flag  
// 11. Send dummy output to start cloak and read data (Send 0x00)  
// 12. Read high temperature into variable MSB
```

**Figure 7.51**  
cont'd

```

// 13. Send dummy output to start clock and read data (Send 0x00)
// 14. Read low temperature into variable LSB
// 15. Disable TC72 data transfer (CE=0)
// 16. Copy high result into variable "result"
void Read_Temperature(void)
{
    char dummy;

    CE = 1;                                // Enable TC72
    Send_To_TC72(0x80);                     // Send control write with A7=1
    dummy = SSPBUF;                         // Clear BF flag
    Send_To_TC72(0x11);                     // Set for one-shot operation
    CE = 0;                                 // Disable TC72
    dummy = SSPBUF;                         // Clear BF flag
    Delay_Ms(200);                          // Wait 200 ms for conversion
    CE = 1;                                 // Enable TC72
    Send_To_TC72(0x02);                     // Read MSB temperature address
    dummy = SSPBUF;                         // Clear BF flag
    Send_To_TC72(0x00);                     // Read temperature high byte
    MSB = SSPBUF;                           // save temperature and clear BF
    Send_To_TC72(0x00);                     // Read temperature low byte
    LSB = SSPBUF;                           // Save temperature and clear BF
    CE = 0;                                 // Disable TC72
    result = MSB;
}

// This function formats the temperature for displaying on the LCD.
// We have to convert to a string to display on the LCD.
//
// Only the positive MSB is displayed in this version of the program
//
void Format_Temperature(char *tmp)
{
    itoa(tmp,result,10);                   // Convert integer to ASCII
}

//
// This function clears the LCD, homes the cursor and then displays the
// temperature on the LCD
//
void Display_Temperature(char *d)
{
    LCD_Clear();                           // Clear LCD and home cursor
    putsLCD(d);
}

```

**Figure 7.51**  
cont'd

```
void main(void)
{
    char msg[] = "Temperature...";
    char tmp[3];

    ANSELB = 0;                                // Configure PORTB as digital
    ANSELC = 0;                                // Configure PORTC as digital

    TRISCbits.RCO = 0;                          // Configure RCO (CE) as output
    TRISB = 0;                                  // Configure PORT B as outputs

    Delay_Seconds(1);
//
// Initialize the LCD
//
    Init_LCD();
    while(BusyLCD());                         // Wait if the LCD is busy
    WriteCmdLCD(DON);                        // Turn Display ON
    while(BusyLCD());                         // Wait if the LCD is busy
    WriteCmdLCD(0x06);                        // Move cursor right
    LCD_Clear();                               // Clear LCD
//
// Display a message on the LCD
//
    putsLCD(msg);
    Delay_Seconds(2);
    LCD_Clear();
    Init_SPI();                                // Initialize the SPI bus
//
// Endless loop. Inside this loop read the TC72 temperature, display on the LCD,
// wait for 1 s and repeat the process
//
    for(;;)                                    // Endless loop
    {
        Read_Temperature();                   // Read the TC72 temperature
        Format_Temperature(tmp);            // Format the data for display
        Display_Temperature(tmp);          // Display the temperature
        Delay_Seconds(1);                  // Wait 1 s
    }
}
```

**Figure 7.51**

cont'd

**Format\_Temperature:** This function converts the integer temperature into an ASCII string so that it can be displayed on the LCD.

**Display\_Temperature:** This function calls to LCD\_Clear to clear the LCD screen and home the cursor. The temperature is then displayed calling function putsLCD.

**Main Program:** At the beginning of the main program, the port directions are configured, the LCD is initialized, the message “Temperature...” is sent to the LCD, and the

microcontroller SPI bus is initialized. The program then enters an endless loop where the following functions are called inside this loop:

```
Read_Temperature();
Format_Temperature(tmp);
Display_Temperature(tmp);
One_Second_Delay();
```

### ***Displaying Negative Temperatures***

The program given in [Figure 7.51](#) displays only the positive temperatures. Negative temperatures are stored in TC72 in two's complement format. If bit 8 of the MSB byte is set, the temperature is negative and two's complement should be taken to find the correct temperature. For example, if the MSB and LSB bytes are “1110 0111/1000 0000”, the correct temperature is

1110 0111/1000 0000 → the complement is 0001 1000/0111 1111

adding “1” to find the two's complement gives: 0001 1000/1000 0000,  
that is, the temperature is “−24.5 °C”.

Similarly, if the MSB and LSB bytes are “1110 0111/0000 0000”, the correct temperature is

1110 0111/0000 0000 → the complement is 0001 1000/1111 1111

adding “1” to find the two's complement gives: 0001 1001/0000 0000,  
that is, the temperature is “−25 °C”.

In the modified program, both negative and positive temperatures are displayed where the sign “−“ is inserted in-front of negative temperatures. The temperature is displayed in integer format with no fractional part in this version of the program. The Format\_Temperature function is modified such that if the temperature is negative the two's complement is taken, the sign bit is inserted, and then the value is shifted right by 8 digits and converted into an ASCII string for the display.

The new Format\_Temperature function is shown below:

```
//  
// Positive and negative temperatures are displayed in this version of the program  
//  
void Format_Temperature(char *tmp)  
{  
    if(result & 0x8000)          // If negative
```

```
{  
    result = ~result;      // Take complement  
    result++;             // Take 2's complement  
    result >>= 8;         // Get integer part  
    *tmp++ = '-';        // Insert "-" sign  
}  
else  
{  
    result >>= 8;         // Get integer part  
}  
itoa(tmp,result,10);    // Convert integer to ASCII  
}
```

### ***Displaying the Fractional Part***

The program in [Figure 7.51](#) does not display the fractional part of the temperature. The program can be modified to display the fractional part as well. In the new function, the LSB byte of the converted data is taken into consideration and the fractional part is displayed as “.00”, “.25”, “.50”, or “.75”. The two most significant bits of the LSB byte are shifted right by 6 bits. The fractional part then takes one of the following values:

Two Shifted LSB Bits	Fractional Part
00	.00
01	.25
10	.50
11	.75

[Figure 7.52](#) shows the modified program (XC8-SPI2.C).

### ***Project 7.6—Real-Time Clock Using an RTC Chip***

In this project, we will design a clock using a real-time clock (RTC) chip. We will be using three push-button switches to set the clock initially: Mode, Up, and Down. Mode button will select the date and time field, Up and Down buttons will increment and decrement the selected field, respectively.

There are several RTC chips available. The one that we will be using in this project is the PCF8583 eight-pin DIL chip. The specifications of this chip are as follows:

- Clock, alarm, and timer functions;
- I<sup>2</sup>C bus interface;
- A +2.5- to +6-V operation;
- A 32.768-kHz time base (requires an external 32.768-kHz crystal);
- Programmable alarm, timer, and interrupt functions;
- A 240 × 8 random access memory (RAM).

```
*****
          SPI BUS BASED DIGITAL THERMOMETER
*****
```

In this project a TC72 type SPI bus based temperature sensor IC is used. The IC is connected to the SPI bus pins of a PIC18F45K22 type microcontroller (i.e. to pins RC3=SCK, RC4=SDI, and RC5=SDO) and the microcontroller is operated from an 8 MHz crystal.

In addition PORT B pins of the microcontroller are connected to a standard LCD.

The microcontroller reads the temperature every second and displays on the LCD

This version of the program displays the sign as well as the fractional part of the temperature. An example display is:

-23.75

Author: Dogan Ibrahim  
 Date: October 2013  
 File: XC8-SPI2.C

```
*****
#include <xc.h>
#include <string.h>
#include <plib/usart.h>
#include <plib/xlcd.h>
#include <plib/spi.h>
#include <stdlib.h>

#pragma config MCLRE = EXTMCLR, WDTEN = OFF, FOSC = HSHP
#define _XTAL_FREQ 8000000

#define CE PORTCbits.RC0
#define Ready SSPSTATbits.BF

unsigned char LSB, MSB;
int result,int_part,fract_part;

// 
// 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);
    }
}
```

**Figure 7.52: Modified MPLAB XC8 Program to Display Fractional Part as Well.**

```
//  
// This function creates milliseconds delay. The argument specifies the delay time in ms  
//  
void Delay_Ms(unsigned int ms)  
{  
    unsigned int i;  
  
    for(i = 0; i < ms; i++)__delay_ms(1);  
}  
  
//  
// 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);  
}
```

**Figure 7.52**  
cont'd

```
//  
// Initialize the LCD, clear and home the cursor  
//  
void Init_LCD(void)  
{  
    OpenLCD(FOUR_BIT & LINE_5X7);           // 8 bit, 5x7 character  
    LCD_Clear();                            // Clear LCD  
}  
  
//  
// Initialize the SPI bus  
//  
void Init_SPI(void)  
{  
    OpenSPI(SPI_FOSC_4, MODE_01, SMPEND);    // SPI clk = 2MHz  
}  
  
//  
// This function sends a control byte to the TC72 and waits until the  
// transfer is complete  
//  
void Send_To_TC72(unsigned char cmd)  
{  
    SSPBUF = cmd;                          // Send control to TC72  
    while(!Ready);                         // Wait until data is shifted out  
}  
  
//  
// This function reads the temperature from the TC72 sensor  
//  
// Temperature data is read as follows:  
//  
// 1. Enable TC72 (CE=1, for single byte write)  
// 2. Send Address 0x80 (A7=1)  
// 3. Clear BF flag  
// 4. Send One-Shot command (Control = 0001 0001)  
// 5. Disable TC72 (CE=0, end of single byte write)  
// 6. Clear BF flag  
// 7. Wait at least 150ms for temperature to be available  
// 8. Enable TC72 (CE=1, for multiple data transfer)  
// 9. Send Read MSB command (Read address=0x02)  
// 10. Clear BF flag  
// 11. Send dummy output to start clock and read data (Send 0x00)  
// 12. Read high temperature into variable MSB  
// 13. Send dummy output to start clock and read data (Send 0x00)
```

**Figure 7.52**  
cont'd

```

// 14. Read low temperature into variable LSB
// 15. Disable TC72 data transfer (CE=0)
// 16. Copy high result into variable "result"
void Read_Temperature(void)
{
    char dummy;

    CE = 1;                                // Enable TC72
    Send_To_TC72(0x80);                     // Send control write with A7=1
    dummy = SSPBUF;                         // Clear BF flag
    Send_To_TC72(0x11);                     // Set for one-shot operation
    CE = 0;                                 // Disable TC72
    dummy = SSPBUF;                         // Clear BF flag
    Delay_Ms(200);                          // Wait 200 ms for conversion
    CE = 1;                                 // Enable TC72
    Send_To_TC72(0x02);                     // Read MSB temperature address
    dummy = SSPBUF;                         // Clear BF flag
    Send_To_TC72(0x00);                     // Read temperature high byte
    MSB = SSPBUF;                           // save temperature and clear BF
    Send_To_TC72(0x00);                     // Read temperature low byte
    LSB = SSPBUF;                           // Save temperature and clear BF
    CE = 0;                                 // Disable TC72
    result = MSB*256+LSB;                   // The complete temperature
}

// Positive and negative temperatures are displayed as well as the fractional part
// void Format_Temperature(char *tmp)
{
    if(result & 0x8000)                      // If negative
    {
        result = ~result;                   // Take complement
        result++;                          // Take 2's complement
        int_part = result >> 8;           // Get integer part
        *tmp++ = '-';
    }
    else
    {
        int_part = result >> 8;           // Get integer part
    }

    itoa(tmp,result,10);                    // Convert integer to ASCII
}

// Now find the fractional part. First we must find the end of the string "tmp"
// and then append the fractional part to it
//
//     while(*tmp != '\0')tmp++;           // find end of string "tmp"
//

```

**Figure 7.52**  
cont'd

```

// Now add the fractional part as ".00", ".25", ".50", or ".75"
//
fract_part = result & 0x00C0;           // fractional part
fract_part = fract_part >> 6;          // fract is between 0-3
switch(fract_part)
{
    case 1:                         // Fractional part = 0.25
        *tmp++ = '.';
        *tmp++ = '2';
        *tmp++ = '5';
        break;
    case 2:                         // Fractional part = 0.50
        *tmp++ = '.';
        *tmp++ = '5';
        *tmp++ = '0';
        break;
    case 3:                         // Fractional part = 0.75
        *tmp++ = '.';
        *tmp++ = '7';
        *tmp++ = '5';
        break;
    case 0:                         // Fractional part = 0.00
        *tmp++ = '.';
        *tmp++ = '0';
        *tmp++ = '0';
        break;
}
*tmp++ = '\0';                      // Null terminator
}

//
// This function clears the LCD and then displays the temperature on the LCD
//
void Display_Temperature(char *d)
{
    LCD_Clear();                     // Clear LCD and home cursor
    putsLCD(d);
}

void main(void)
{
    char msg[] = "Temperature...";
    char tmp[8];

    ANSELB = 0;                      // Configure PORTB as digital
    ANSELC = 0;                      // Configure PORTC as digital
    TRISCbits.RC0 = 0;                // Configure RC0 (CE) as output
}

```

**Figure 7.52**  
cont'd

```

TRISB = 0;                                // Configure PORT B as outputs

Delay_Seconds(1);

// Initialize the LCD
//
Init_LCD();
while(BusyLCD());
WriteCmdLCD(DON);                         // Wait if the LCD is busy
while(BusyLCD());                          // Turn Display ON
WriteCmdLCD(0x06);                         // Wait if the LCD is busy
LCD_Clear();                               // Move cursor right
                                         // Clear LCD

// Display a message on the LCD
//
putsLCD(msg);
Delay_Seconds(2);
LCD_Clear();
Init_SPI();                                // Initialize the SPI bus

// Endless loop. Inside this loop read the TC72 temperature, display on the LCD,
// wait for 1 s and repeat the process
//
for(;);                                    // Endless loop
{
    Read_Temperature();                    // Read the TC72 temperature
    Format_Temperature(tmp);              // Format the data for display
    Display_Temperature(tmp);            // Display the temperature
    Delay_Seconds(1);                   // Wait 1 s
}
}

```

**Figure 7.52**  
cont'd

The PCF8583 operates as a slave I<sup>2</sup>C device with devices addresses 0xA1 or 0xA3 for reading, and 0xA0 or 0xA2 for writing.

[Figure 7.53](#) shows the block diagram of the project.

Before going into the details of the design, it is worthwhile to review the basic principles of the I<sup>2</sup>C bus communications protocol. I<sup>2</sup>C is a bidirectional two-line communication between a master and one or more slave devices. The two lines are named SDA (serial data) and SCL (serial clock). Both lines must be pulled up to the supply voltage using suitable resistors. [Figure 7.54](#) shows a typical system configuration with one master and three slaves communicating over the I<sup>2</sup>C bus.

Most high-level language compilers provide libraries for I<sup>2</sup>C communication. We can also easily develop our own I<sup>2</sup>C library. Although the available libraries can easily be used, it is worthwhile to look at the basic operating principles of the bus.

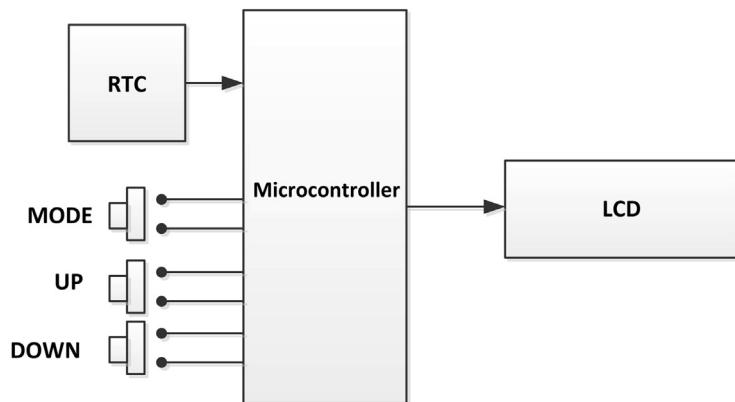


Figure 7.53: Block Diagram of the Project.

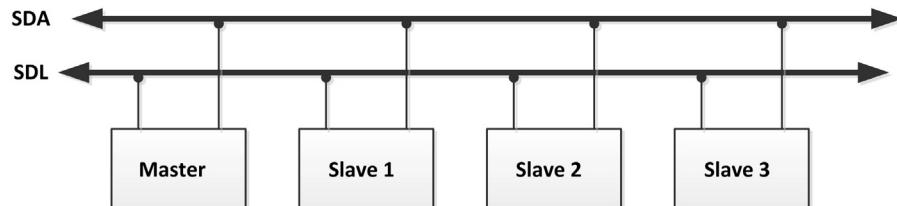
Figure 7.54: I<sup>2</sup>C System Configuration.

Figure 7.55: Start and STOP Bit Conditions.

The I<sup>2</sup>C bus must not be busy before data can be sent over the bus. Data are sent serially, and synchronized with the clock. Both SDA and SCL lines are HIGH when the bus is not busy. The START bit is identified by the HIGH-to-LOW transition of the SDA line while the SCL is HIGH. Similarly, a LOW-to-HIGH transition of the SDA line while the SCL is HIGH is identified as the STOP bit. [Figure 7.55](#) shows both the START and STOP bit conditions.

One bit of data is transferred during each clock pulse. Data on the bus must be stable when SCL is HIGH; otherwise, the data will be interpreted as a control signal. Data can change when the SCL line is LOW. [Figure 7.56](#) shows how bit transfer takes place on the bus.

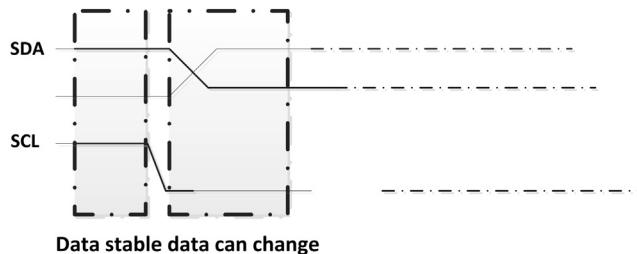


Figure 7.56: Bit Transfer on the Bus.

Each byte of 8 bits on the bus is followed by an acknowledgment cycle. The acknowledgement cycle has the following requirements:

- An addressed slave device must generate an acknowledgement after the reception of each byte from the master.
- A master receiver must acknowledge after the reception of each data byte from the slave (except the last byte).
- The acknowledge signal is identified by a device by lowering the SDA line during the acknowledge clock HIGH pulse.
- A master receiver must signal the end of data to the transmitter by not lowering the SDA line during the acknowledge clock HIGH pulse. In this case, the transmitter leaves the SCL line HIGH so that the master can generate the STOP bit.

The communication over the I<sup>2</sup>C bus is based on addressing where each device has a unique 8-bit address, usually setup by hardware configuration. Before sending any data, the address of the device that is expected to respond is sent after the START bit.

During the PCF8583 write cycle, the following events occur:

- Master sends START bit.
- Master sends the slave address. Bit 0 of the address is 0 for a write operation.
- Slave sends an acknowledgement bit.
- Master sends the register address to specify the slave register to be accessed.
- Slave sends an acknowledgement bit.
- Slave sends the required data.
- Slave sends an acknowledgement bit.
- Master sends a STOP bit.

During the PCF8583 read cycle, the following events occur (8-bit address mode):

- Master sends START bit.
- Master sends the slave address. Bit 0 of the address is 1 for a read operation.
- Slave sends an acknowledgement bit.

- Slave sends data.
- Master sends acknowledgement bit.
- Slave sends the last byte.
- Master does not send the acknowledgement bit.
- Master sends the STOP bit.

In some read applications, as in the RTC clock project, it is sometimes necessary to send a 16-bit address to the slave device in the form of the actual device address, followed by the address of the register to be accessed. In such applications, the PCF8583 read cycle is as follows:

- Master sends START bit.
- Master sends the slave address in the write mode. Bit 0 of the address is 0 for a write operation.
- Slave sends an acknowledgement bit.
- Master sends the register address.
- Slave sends the acknowledgement bit.
- Master sends repeated START bit.
- Master sends the slave address in the read mode. Bit 0 of the address is 1 for a read operation.
- Slave sends the acknowledgement bit.
- Slave sends data.
- Master sends the acknowledgement bit.
- Slave sends the last byte.
- Master does not send the acknowledgement bit.
- Master sends the STOP bit.

When in the clock mode, the operation of the PCF8583 RTC chip is configured with seven registers. [Figure 7.57](#) shows all the registers of the chip. The remaining registers are used to configure the timer and alarm functions as we shall see in the next project.

The first register is the control/status register. This register by default is loaded by 0x00 after reset. The important bit here as far as the clock operation is concerned is bit 7. This bit stops and restarts the internal clock counter. The normal state of this bit is 0, but it must be set to 1 to stop the counter during loading the current date and time information to the chip.

The date and time information is stored in the BCD format, the upper nibble holding the 10 s and the lower nibble holding the 1 s. For example, number 25 is stored in binary pattern as “0010 0101”. The data should be converted into the correct format before being displayed on the LCD or before new data are loaded into the registers.

It is important to know the format of the registers during programming. Figures 7.58–7.60 give the format of the hour register, year–date register, and the weekday–month register, respectively. Note that the format of the year–day register is different from the others. The year is stored in 2 bits, having values 0–3. Thus, for example, to display year 2013, we have to provide the first three digits (201x) and read the last digit from the clock chip.

	Bit 7	Bit 4 Bit 3	Bit 0
	<b>0</b>	Control/status	
Hundredths of a second	<b>1</b>	1/10 s	1/100 s
Seconds	<b>2</b>	10 s	1 s
Minutes	<b>3</b>	10 min	1 min
Hours	<b>4</b>	10 h	1 h
Year/day	<b>5</b>	10 day	1 day
Weekday/month	<b>6</b>	10 month	1 month
Timer	<b>7</b>	10 day	1 day
Alarm control	<b>8</b>	Alarm control	
Hundredths of a second	<b>9</b>	1/10 s	1/100 s
Alarm seconds	<b>10</b>	10 s	1 s
Alarm minutes	<b>11</b>	10 min	1 min
Alarm hours	<b>12</b>	10 h	1 h
Alarm date	<b>13</b>	10 day	1 day
Alarm month	<b>14</b>	10 month	1 month
Alarm timer	<b>15</b>		
Free RAM	<b>16</b>	RAM (240 Bytes)	

Figure 7.57: PCF8583 Registers.

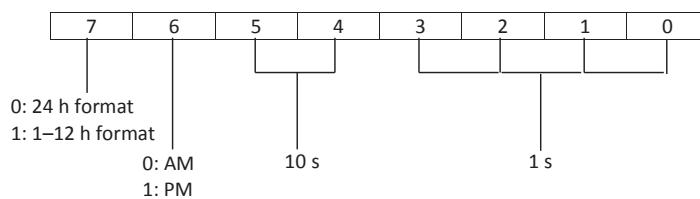


Figure 7.58: Hours Register.

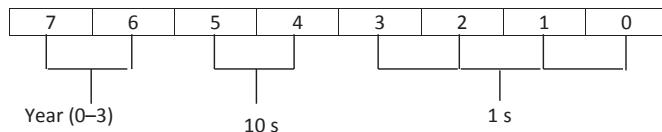


Figure 7.59: Year–Date Register.

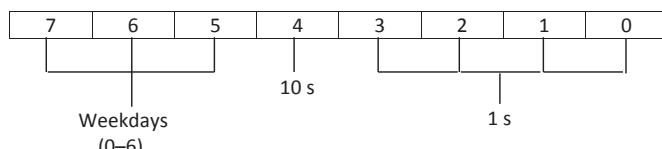


Figure 7.60: Weekday–Month Register.

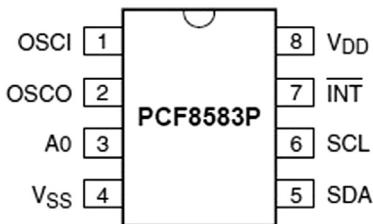


Figure 7.61: PCF8583 Pin Layout.

### Project Hardware

Figure 7.61 shows the pin layout of the PCF8583 chip. The pin descriptions are as follows:

OSC1	oscillator or event pulse input
OSC0	oscillator input/output
A0	address input
V <sub>SS</sub> , V <sub>DD</sub>	power lines
SDA	data pin
SCL	clock pin
INT	interrupt output pin (active LOW, open drain)

If pin A0 is connected LOW the device responds to addresses 0xA0 and 0xA1 for writing and reading, respectively; otherwise, it responds to 0xA2 and 0xA3 for writing and reading, respectively.

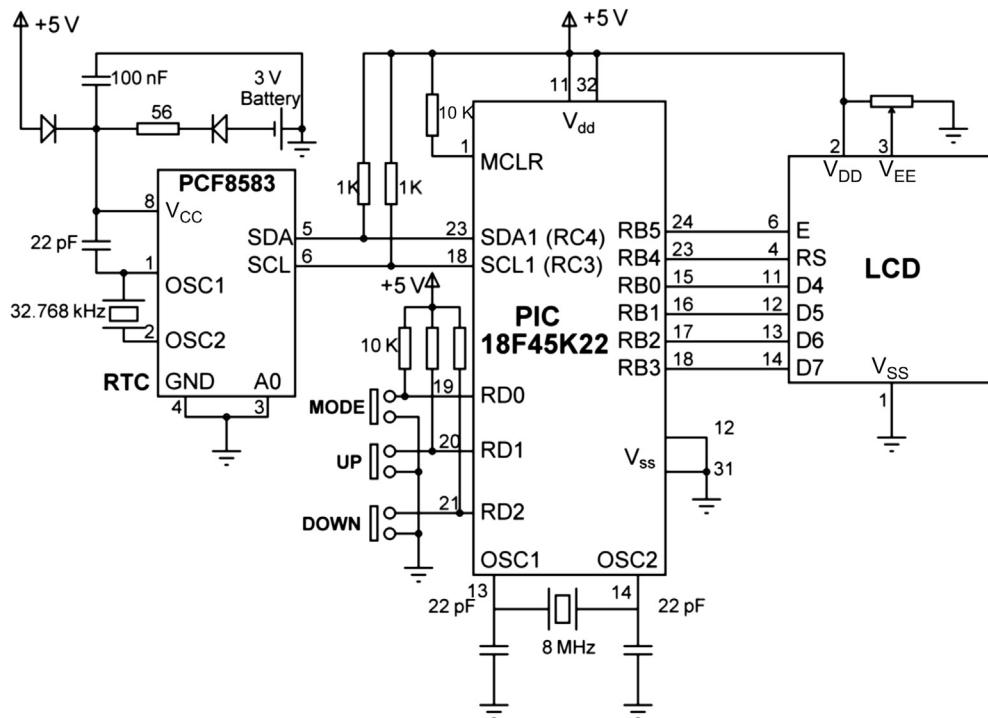
The circuit diagram of the project is shown in Figure 7.62. The SDA and SCL pins of the PCF8583 are connected to microcontroller pins SDA1 (RC4) and SCL1 (RC3), respectively. Pin A0 is connected to the ground to select slave addresses 0xA0 and 0xA1. An LCD is connected to PORTB to display the clock. Three push-button switches connected to PORTD pins are used to set the initial date and time:

MODE (RD0)	this button is used to select the date or time field that will be set
UP (RD1)	this button increments the selected field
DOWN (RD2)	this button decrements the selected field

A 32.768-kHz crystal is used to provide timing pulses to the chip. A button shaped 3-V lithium battery (CR2032) is used to power the RTC chip so that it keeps the time even after the microcontroller power is turned off.

If you are using the mikroElektronika RTC board with the EasyPIC V7 development board, set the following jumpers on the RTC board and plug-in the board to PORTC:

- Set switch 3 ON (Connect SDA line)
- Set switch 4 ON (Connect SCL line)



**Figure 7.62: Circuit Diagram of the Project.**

In addition, set the following jumper on the EasyPIC V7 development board so that when a button is pressed the button state goes from logic HIGH to logic LOW:

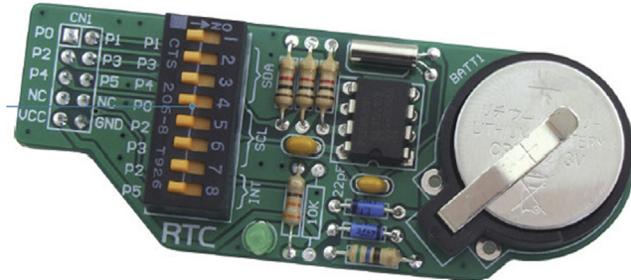
Set J17 to GND

PORTD RD0 switch to pull-up

PORTD RD1 switch to pull-up

## PORTD RD2 switch to pull-up

Figure 7.63 shows the mikroElektronika RTC board.



**Figure 7.63:** mikroElektronika RTC Board.

## **Project PDL**

The project PDL is shown in [Figure 7.64](#).

## **Project Program**

### *mikroC Pro for PIC*

The mikroC Pro for the PIC compiler I<sup>2</sup>C library supports the following functions ('x' refers to the I<sup>2</sup>C module used in microcontrollers with more than one module):

I2Cx_Init	Initialize the I <sup>2</sup> C library. The I <sup>2</sup> C clock rate must be entered as an argument
I2Cx_Start	Sends START bit on the bus
I2C_Repeated_Start	Sends repeated START bits
I2C_Is_Idle	Returns 1 if the bus is free, otherwise returns 0
I2Cx_Rd	Reads one byte from the slave. If the argument is 1 an acknowledgement is sent, otherwise acknowledgement is not sent.
I2Cx_Wr	Sends a byte to the slave device. Returns 0 if there are no errors
I2Cx_Stop	Sends STOP bit on the bus

The mikroC Pro for PIC program is given in [Figure 7.65](#) (MIKROC-I2C.C). Before looking at the software in detail let us assume that we wish to set the date and time to 10-09-2013 08:10:15. The steps are given below:

- Reset the microcontroller while pressing the MODE button.
- The LCD should display:

DAY:  
31

- Keep pressing the UP button until 10 is displayed.
- Press MODE button to change the field to month:

MONTH:  
12

- Keep pressing the DOWN button until 9 is displayed.
- Press MODE button to change the field to year:

YEAR (201x):  
6

- Keep pressing the UP button until 3 is displayed (only the last digit of the year is entered).
- Press MODE button to change the field to hour:

HOUR:  
23

**Main Program:**

```
BEGIN
    Define LCD – microcontroller interface
    Assign symbols MODE, UP, DOWN to port pins
    Configure PORTB, PORTC, PORTD as digital
    Configure RD0, RD1, RD2 as inputs
    Initialize LCD
    Initialize I2C bus
    IF SETUP mode
        CALL Set_Date_Time to read new date and time values
        CALL SET_RTC to load the new values into clock chip
    ENDIF
    DO FOREVER
        CALL Read_Date_Time to read date and time from the clock chip
        CALL Convert_Date_Time to convert into displayable format
        CALL Display_Date_Time to display the date and time on the LCD
    ENDDO
END
```

**BEGIN/Set\_Date\_Time**

```
Display maximum field value
WHILE MODE button not pressed
    IF UP button pressed
        Increment value
        IF value > maximum
            Set value = minimum
        ENDIF
    ENDIF
    IF DOWN button pressed
        Decrement value
        IF value < minimum
            Set value = maximum
        ENDIF
    ENDIF
WEND
Return new value to the calling program
END/Set_Date_Time
```

**BEGIN/SET\_RTC**

```
Convert date and time into BCD
Load date and time to the clock chip
END/SET_RTC
```

**BEGIN/Read\_Date\_Time**

```
Get date and time from the clock chip
END/Read_Date_Time
```

**BEGIN/Convert\_Date\_Time**

**Figure 7.64: Project PDL.**

Convert date and time into ASCII for display  
**END/Convert\_Date\_Time**

**BEGIN/Display\_Date\_Time**  
 Display converted date on row 1  
 Display converted time on row 2  
**END/Display\_Date\_Time**

**Figure 7.64**  
 cont'd

- Keep pressing the UP button until 8 is displayed.
- Press the MODE button to change the field to minutes:

MINUTES:  
 59

- Keep pressing the UP button until 10 is displayed.
- Press the MODE button to change the field to seconds:

SECONDS:  
 59

- Keep pressing the UP button until 15 is displayed.
- Press the MODE button to terminate the setup. The clock should start working from the set date and time.

At the beginning of the program, the connections between the LCD and the microcontroller are defined. Symbols MODE, UP, DOWN and SETUP are assigned to port bits, PORTB, PORTC, PORTD are configured as digital with RD0, RD1, and RD2 pins configured as inputs. Then, the LCD and the I<sup>2</sup>C modules are initialized. If the MODE button is pressed (MODE = 0) during the reset, the program enters the SETUP phase. Here, the new data and time values are read via the UP/DOWN/MODE buttons as described earlier, using function Set\_Date\_Time. This function displays the field to be modified at the first row of the LCD. The user changes the displayed value by pressing the UP or DOWN buttons. When the required value is displayed, the MODE button is pressed to move to the next field. This process continues until the last field value is selected and then the program calls function SET\_RTC to load the new date and time values into the registers of PCF8583.

The rest of the program is executed in an endless loop. Here, the date and time are read from the PCF8583 using function Read\_Date\_Time. Function Convert\_Date\_Time is called to convert these values into a form that can be displayed on the LCD. Finally, function Display\_Date\_Time is called to display the date and time.

```
*****  
REAL TIME CLOCK  
=====
```

This project is about designing an accurate real time clock (RTC) using the RTC chip PCF8583

The PCF8583 chip is connected to the I2C pins (module 1) of a PIC18F45K22 microcontroller. The connections between the PCF8583 and the microcontroller are as follows (the SDA and SCL lines are pulled high with resistors):

PCF8583	Microcontroller
=====	=====
SDA	SDA1 (RC4)
SCL	SCL1 (RC3)

Clock timing to the PCF8583 is provided with a 32.768 kHz crystal. 3 Push-button switches are used to set the clock initially:

- MODE (RD0): Selects the date or time field to be set
- UP (RD1): Increments the value
- DOWN (RD2): Decements the value

An LCD is connected to PORTB of the microcontroller to help in setting the clock and also for displaying the clock data in real time.

The microcontroller is operated with an 8 MHz crystal with the PLL disabled (i.e. the actual running clock frequency is 8 MHz).

The software has 2 phases: SETUP and RUNNING. The SETUP phase is entered if the MODE button is pressed during the startup. In this phase the clock is set to current date and time. The RUNNING phase is entered if the MODE button is not pressed during the startup and this is the normal running state of the clock where the date and time are displayed on the LCD in the following format:

dd-mm-yyyy  
hh:mm:ss

Author: Dogan Ibrahim  
Date: October 2013  
File: MIKROC-I2C.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;
```

**Figure 7.65: mikroC Pro for PIC Program.**

```

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 MODE PORTD.RD0                                // MODE button
#define UP PORTD.RD1                                 // UP button
#define DOWN PORTD.RD2                               // DOWN button
#define SETUP 0

unsigned char seconds, minutes, hours, day, month, year;
unsigned char newday, newmonth, newyear, newhour, newminutes, newseconds;
//
// This function reads the Date and Time from the RTC chip
//
void Read_Date_Time()
{
    I2C1_Start();                                     // Send START bit to RTC chip
    I2C1_Wr(0xA0);                                  // Address the RTC chip
    I2C1_Wr(0x02);                                  // Start from address 2 (seconds)
    I2C1_Repeated_Start();                           // Issue repeated START bit
    I2C1_Wr(0xA1);                                  // Address the RTC chip for reading
    seconds = I2C1_Rd(1);                           // Read seconds, send ack
    minutes = I2C1_Rd(1);                          // Read minutes, send ack
    hours = I2C1_Rd(1);                            // Read hours, send ack
    day = I2C1_Rd(1);                             // Read year/day, send ack
    month = I2C1_Rd(0);                           // Read month, no ack (last byte)
    I2C1_Stop();                                    // Send STOP bit to RTC chip
}

//
// This function converts the date-time into correct format for displaying on the LCD. The
// numbers in RTC memory are in BCD form and are converted as follows:
// "extract upper byte, shift right 4 bits, multiply by 10, add lower byte".
// For example, number 25 in RTC memory is stored as 37. i.e bit pattern: "0010 0101".
// After the conversion we obtain the required number 25.
//
void Convert_Date_Time()
{
    seconds = ((seconds & 0xF0) >> 4)*10 + (seconds & 0x0F);
    minutes = ((minutes & 0xF0) >> 4)*10 + (minutes & 0x0F);
    hours = ((hours & 0xF0) >> 4)*10 + (hours & 0x0F);
    month = ((month & 0x10) >> 4)*10 + (month & 0x0F);
    year = (day & 0xC0) >> 6;
    day = ((day & 0x30) >> 4)*10 + (day & 0x0F);
}

```

**Figure 7.65**  
cont'd

```

// Display the date and time on the LCD
//
void Display_Date_Time()
{
//
// Write day, month, year as: dd=mm=xxxY
//
    Lcd_Chr(1, 1, (day / 10) + '0');
    Lcd_Chr(1, 2, (day % 10) + '0');
    Lcd_Chr(1, 4, (month / 10) + '0');
    Lcd_Chr(1, 5, (month % 10) + '0');
    Lcd_Chr(1, 10, year + '0');
//
// Write hour, minutes, seconds as: hh:mm:ss
//
    Lcd_Chr(2, 1, (hours / 10) + '0');
    Lcd_Chr(2, 2, (hours % 10) + '0');
    Lcd_Chr(2, 4, (minutes / 10) + '0');
    Lcd_Chr(2, 5, (minutes % 10) + '0');
    Lcd_Chr(2, 7, (seconds / 10) + '0');
    Lcd_Chr(2, 8, (seconds % 10) + '0');
}

//
// This function gets the date and Time from the user via the 3 buttons. New values
// of Date and Time are returned to the calling program. Initially the maximum values
// are shown and these can be changed using the UP and DOWN buttons.
//
unsigned char Set_Date_Time(unsigned char *str, unsigned char min, unsigned char max)
{
    unsigned char c, Txt[4];

    ByteToStr(max, Txt);                                // Convert max value to string in Txt
    Lcd_Cmd(_LCD_CLEAR);                               // Display field name (e.g. "DAY:")
    Lcd_Out(1,1,str);                                 // Display max value to start with
    c = max;

    while(MODE == 1)                                    // While MODE button not pressed
    {
        if(UP == 0)                                     // If UP button pressed (increment)
        {
            Delay_Ms(10);
            while(UP == 0);                            // Wait until UP button is released
            c++;
            if(c > max)c = min;
        }
        if(DOWN == 0)                                   // If DOWN button is pressed
        {
            Delay_Ms(10);
            while(DOWN == 0);                         // Wait until DOWN button is released
        }
    }
}

```

**Figure 7.65**  
cont'd

```

        c--;
        if(c < min || c == 255)c = max;           // Decrement value
                                                // If less than min, rollover to max
    }
    ByteToStr(c, Txt);                      // Convert selected value to string
    Lcd_Out(2,1,Txt);                      // Display selected value on LCD
}
Delay_Ms(10);
while(MODE == 0);                         // Wait until MODE button is released
return c;                                  // return number to calling program
}

// This function sets the RTC with the new Date and Time values. The number to be sent to
// the RTC chip is divided by 10, shifted left 4 digits, and the remainder is added to it. Thus,
// for example if the number is decimal 25, it is converted into bit pattern "0010 0101" and
// stored at the appropriate RTC memory
//
void Set_RTC()
{
//
// Convert Date and Time into a format compatible with the RTC chip
//
seconds = ((newseconds / 10) << 4) + (newseconds % 10);
minutes = ((newminutes / 10) << 4) + (newminutes % 10);
hours = ((newhour / 10) << 4) + (newhour % 10);
month = ((newmonth / 10) << 4) + (newmonth % 10);
day = (newyear << 6) + ((newday / 10) << 4) + (newday % 10);

I2C1_Start();                            // Send START bit to RTC chip
I2C1_Wr(0xA0);                         // Address the RTC chip
I2C1_Wr(0x00);                         // Start from address 0
I2C1_Wr(0x80);                         // Pause RTC counter
I2C1_Wr(0x00);                         // Write to hundredths memory location
I2C1_Wr(seconds);                      // Write to seconds memory location
I2C1_Wr(minutes);                      // Write to minutes memory location
I2C1_Wr(hours);                        // Write to hours memory location;
I2C1_Wr(day);                          // Write to year/day memory location
I2C1_Wr(month);                        // Write to month memory location
I2C1_Stop();                           // Send STOP bit

I2C1_Start();                            // Send START bit to RTC chip
I2C1_Wr(0xA0);                         // Address the RTC chip
I2C1_Wr(0);                            // Start from address 0 (Configuration reg)
I2C1_Wr(0);                            // Write 0 to Conf reg to start counter
I2C1_Stop();                           // Send STOP bit
}

void main()

```

**Figure 7.65**

cont'd

```

{
    ANSELB = 0;                                // Configure PORTB as digital
    ANSELC = 0;                                // Configure PORTC as digital
    ANSELD = 0;                                // Configure PORTD as digital
    TRISD.RD0 = 1;                             // Configure RD0 (MODE)as input
    TRISD.RD1 = 1;                             // Configure RD1 (UP) as input
    TRISD.RD2 = 1;                             // Configure RD2 (DOWN) as input

    Lcd_Init();                                // Initialize LCD
    Lcd_Cmd(_LCD_CURSOR_OFF);                  // Disable cursor
    Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD

    I2C1_Init(100000);                        // Initialize I2C module
//
// If the MODE button is pressed on startup we must get into SETUP phase
//
if(MODE == SETUP)                         // If SETUP mode
{
    while(MODE == SETUP);                  // Wait until MODE button is released
    newday = Set_Date_Time("DAY:",1,31);    // Get current day
    newmonth = Set_Date_Time("MONTH:",1,12); // Get current month
    newyear = Set_Date_Time("YEAR (201x)",3,6); // Get current year (201x)
    newhour = Set_Date_Time("HOUR:",0,23);   // Get current hour
    newminutes = Set_Date_Time("MINUTES:",0,59); // Get current minutes
    newseconds = Set_Date_Time("SECONDS:",0,59); // get current seconds
//
// We have got all the new Date and Time values. Now set the RTC with these values
//
    Set_RTC();
}

//
// Read the Date and Time from the RTC chip and display on the LCD in the following format:
// Row 1: dd-mm-yyyy
// Row 2: hh:mm:ss
//
Lcd_Out(1, 1, "dd-mm-2013");
Lcd_Out(2, 1, "hh:mm:ss");
while(1)
{
    Read_Date_Time();                     // Read Date and Time from RTC chip
    Convert_Date_Time();                 // Convert into a form to display
    Display_Date_Time();                // Display Date and Time on the LCD
}
}

```

**Figure 7.65**

cont'd

It is important to notice that the PCF8583 RTC chip stores the date and time values in the BCD format. Thus, for example, if the current minutes is 28, it is stored internally as “00101000” (which is decimal 40). Function Convert\_Date\_Time converts the BCD minutes into decimal using the following code:

```
minutes = ((minutes & 0xF0) >> 4)*10 + (minutes & 0x0F);
```

The upper nibble (“0010”) is shifted right by four digits, which becomes decimal 2. It is then multiplied by 10 to give 20. The lower nibble (8) is then added to give the required value 28.

Function `Display_Date_Time` divides the given number by 10 to extract the 10s digit. Then, character ‘0’ is added to convert it into ASCII, and it is then displayed using the `LCD_Chr` function. The 1s digit is also determined by finding the remainder of the division by 10. Again the value is converted into ASCII and displayed on the LCD. The following code is used:

```
Lcd_Chr(2, 4, (minutes/10) + '0');
Lcd_Chr(2, 5, (minutes % 10) + '0');
```

Before loading the clock registers, the opposite process is done, that is, the number is converted into BCD. As an example, the following code is used to convert the minutes:

```
minutes = ((newminutes/10) << 4) + (newminutes % 10);
```

The decimal number is divided by 10 to find the 10s digit. This number is shifted left by four digits so that it occupies the upper nibble position. Then the 1s digit is determined by finding the remainder, and added to the number as the second BCD digit.

#### *MPLAB XC8*

The MPLAB XC8 compiler supports the following I<sup>2</sup>C functions (header file `<plib/i2C.h>` must be included at the beginning of the program):

<code>AckI2C</code>	Generate acknowledgement
<code>CloseI2C</code>	disable MSSP module
<code>DataRdyI2C</code>	Check if data is available in the I <sup>2</sup> C buffer
<code>getclI2C</code>	Read a byte from the I <sup>2</sup> C bus
<code>getslI2C</code>	Read a string from the I <sup>2</sup> C bus (in master mode)
<code>IdleI2C</code>	Loop until I <sup>2</sup> C bus is idle
<code>NotAckI2C</code>	Generate not acknowledgement condition
<code>OpenI2C</code>	Configure the I <sup>2</sup> C module
<code>putclI2C</code>	Write a byte to the I <sup>2</sup> C bus
<code>RestartI2C</code>	Generate a restart condition
<code>StartI2C</code>	Generate a start condition
<code>StopI2C</code>	Generate a stop condition

The program is basically similar to the mikroC Pro for PIC version, but here the initialization routines and the LCD functions are different. Also, the E, RS, and RW pins of the LCD are connected to port pins RB4, RB5, and RB6, respectively.

The MPLAB XC8 version of the program is left as an exercise for the reader.

## Project 7.7—Real-Time Alarm Clock

This project is an extension to Project 7.5. In this project, we set up a daily alarm using the PCF8583 RTC chip. The time of the alarm can be set as in the previous project. An LED is used to indicate the alarm condition (we could have also used a buzzer) and the LED turns ON when alarm occurs. The alarm condition stays until a button is pressed to stop the alarm. The alarm occurs daily at the same time every day.

[Figure 7.66](#) shows the block diagram of the project. The functions of the buttons are as follows:

MODE: Used to enter the clock setup mode. Keep this button pressed while resetting the microcontroller. This button is also used to move between the fields while setting the clock or the alarm time.

UP: Used during clock or alarm setup. Pressing the button increments the value in the selected field.

DOWN: Used during clock or alarm setup. Pressing the button decrements the value in the selected field.

ALARM SETUP: Used to enter the alarm setup mode. Keep this button pressed while resetting the microcontroller.

STOP ALARM: Pressing this button stops the present alarm condition (turns OFF the LED). The alarm will occur at the same time every day.

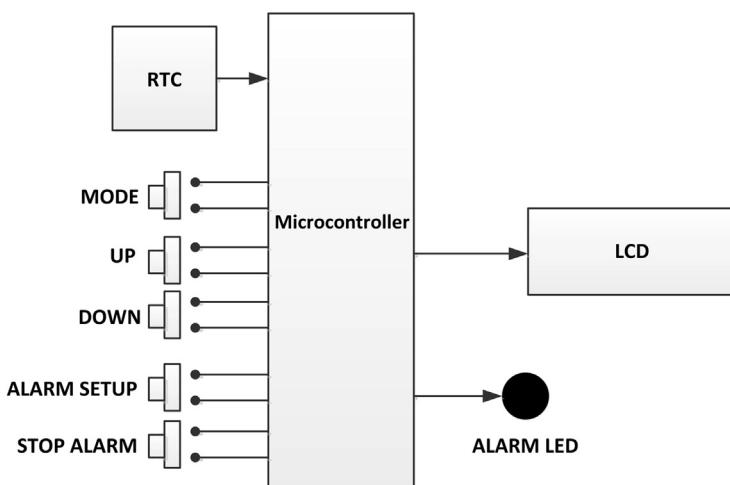


Figure 7.66: Block Diagram of the Project.

Bit 2 of the control and status register (at address 0x00) must be set for the alarm functions to be enabled. When an alarm occurs the INT pin of the PCF8583 goes from logic 1 to logic 0 to indicate the alarm condition. The INT bit can be cleared by clearing bit 0 of the control and status register.

Alarm functions are configured via register 0x08. Figure 7.67 shows the bit definitions of this register. To set daily alarms, the following bits must be set:

Bit 7: Lower INT pin when alarm occurs.

Bit 4: Set daily alarms.

When daily alarms are set, the day, month, and year fields are ignored. An alarm is generated when the contents of the alarm registers match the involved counter registers.

New date and time are loaded into the chip using the MODE, UP, and DOWN buttons as described in the previous project. New daily alarm time is loaded into the chip using the ALARM SETUP, MODE, UP, and DOWN buttons. The steps are given below as an example to set the daily alarm to occur every day at 10:00:00:

- Reset the microcontroller while pressing the ALARM SETUP button.
- The LCD should display:

ALRM HOUR:

23

- Keep pressing the UP button until 10 is displayed.
- Press MODE button to change the field to minutes:

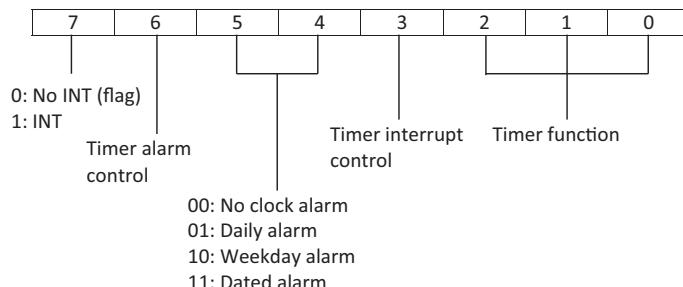
ALRM MINS:

59

- Keep pressing the UP button until 0 is displayed.
- Press MODE button to change the field to seconds:

ALRM SECS:

59



**Figure 7.67: Alarm Control Register (Address 0x08).**

- Keep pressing the UP button until 0 is displayed.
- Press MODE button to return to the clock mode. The daily alarm time will be set to 10:00:00.

### Project Hardware

The circuit diagram of the project is shown in [Figure 7.68](#). This circuit is similar to the one given in [Figure 7.62](#), but here additional buttons and an LED are used for the alarm part of the project. Also, the alarm output pin (INT) of the PCF8583 is connected to the RC2 pin of the microcontroller. Note that this pin is active LOW, that is, it is normally HIGH and goes LOW when an alarm occurs.

### Project PDL

The project PDL is shown in [Figure 7.68](#).

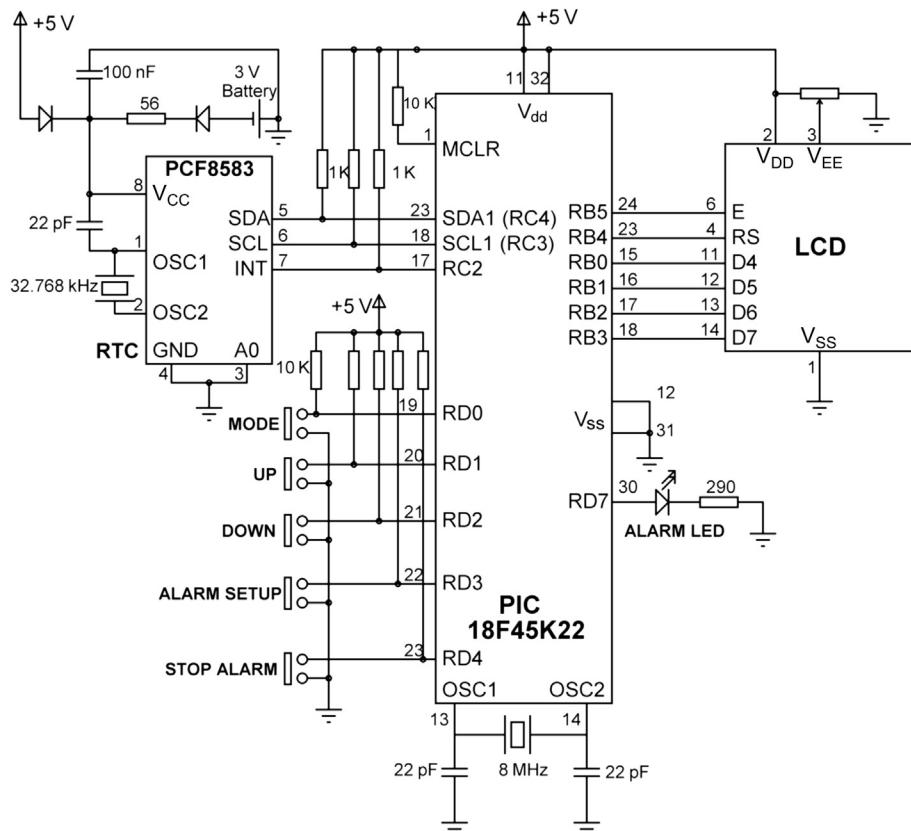


Figure 7.68: Circuit Diagram of the Project.

## **Project Program**

### *mikroC Pro for PIC*

The mikroC Pro for the PIC program listing is given in [Figure 7.69](#) (MIKEOC-I2C2.C). The major part of the program is the same as the clock setting and display program given in the previous project. At the beginning of the program, the connections between the LCD and the microcontroller are defined. Symbols MODE, UP, DOWN, ALARM SETUP, STOP ALARM, and SETUP are assigned to port bits, PORTB, PORTC, PORTD are configured as digital with RD0:RD4 pins configured as inputs. Then, the LCD and the I<sup>2</sup>C modules are initialized. If the MODE button is pressed (MODE = 0) during the reset, the program enters the clock setup phase.

If the ALARM SETUP button is pressed during the reset, the program enters the alarm setup phase. Here, the alarm hours, minutes, and seconds are read via the UP/DOWN/MODE buttons as described earlier, using function Set\_Date\_Time function. The, Set\_RTC\_Alarm function is called to load the PCF8583 registers for the daily alarm so that the alarm occurs every day exactly at the selected time. Here, the alarm register is loaded with 0x90, which enables daily alarms and selects the INT pin as the alarm output. While displaying the current date and time, the program continuously checks the state of the INT pin (this could also be configured as an external interrupt, but the external interrupt pins are used for the LCD) and an alarm condition is said to occur if this pin goes LOW. The program turns ON the ALARM LED to indicate the alarm condition. The present alarm condition can be cleared by pressing the STOP ALARM button. Pressing this button calls function Reset\_Alarm\_Flag, which clears the timer flag (located at bit 0 of the control and status register) to set the INT pin back to HIGH to stop the alarm condition.

## **Project 7.8—SD Card Projects—Write Text To a File**

In this and the next few projects, we will be using SD cards as storage devices. But before going into the details of these projects, we should take a look at the basic principles and operation of SD card memory devices.

SD cards are commonly used in many electronic devices where a large amount of nonvolatile data storage is required. Some application areas are as follows:

- Digital cameras,
- Camcorders,
- Printers,
- Laptop computers,
- GPS receivers,

```
*****  
REAL TIME ALARM CLOCK  
=====
```

This project is about designing an accurate real time alarm clock using the PCF8583 RTC chip

The PCF8583 chip is connected to the I2C pins (modul 1) of a PIC18F45K22 microcontroller. The connections between the PCF8583 and the microcontroller are as follows The SDA, SCL and INT lines are pulled high with resistors):

PCF8583 Microcontroller

```
===== ======  
SDA SDA1 (RC4)  
SCL SCL1 (RC3)  
INT RC2
```

Clock timing to the PCF8583 is provided with a 32.768 kHz crystal. In this project both the clock and daily alarms can be set. When an alarm occurs, an LED connected to pin RD7 is turned ON.

MODE (RD0):	Enter clock setup mode. Also change fields during setup
UP (RD1) :	Increments the value
DOWN (RD2):	Decrements the value
ALARM SETUP (RD3):	Enter Alarm setup mode
STOP ALARM (RD4):	Stop present alarm condition

AN LCD is connected to PORTB of the microcontroller to help in setting the clock and alarm time and also for displaying the clock data in real time.

The microcontroller operated with an 8 MHz crystal with the PLL disabled (i.e. the actual running clock frequency is 8 MHz).

Author: Dogan Ibrahim  
Date: October 2013  
File: MIKROC-I2C2.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
```

**Figure 7.69: mikroC Pro for PIC Program.**

```

#define MODE PORTD.RD0           // MODE button
#define UP PORTD.RD1            // UP button
#define DOWN PORTD.RD2          // DOWN button
#define ALARM_SETUP PORTD.RD3   // ALARM SETUP button
#define STOP_ALARM PORTD.RD4    // Alarm stop button
#define ALARM_INT PORTC.RC2     // RTC clock alarm INT pin
#define ALARM_LED PORTD.RD7     // LED connected to RD7
#define SETUP 0

unsigned char seconds, minutes, hours, day, month, year;
unsigned char newday, newmonth, newyear, newhour, newminutes, newseconds;
// 
// This function reads the Date and Time from the RTC chip
//
void Read_Date_Time()
{
    I2C1_Start();                // Send START bit to RTC chip
    I2C1_Wr(0xA0);              // Address the RTC chip
    I2C1_Wr(0x02);              // Start from address 2 (seconds)
    I2C1_Repeated_Start();       // Issue repeated START bit
    I2C1_Wr(0xA1);              // Address the RTC chip for reading
    seconds = I2C1_Rd(1);        // Read seconds, send ack
    minutes = I2C1_Rd(1);        // Read minutes, send ack
    hours = I2C1_Rd(1);          // Read hours, send ack
    day = I2C1_Rd(1);            // Read year/day, send ack
    month = I2C1_Rd(0);          // Read month, no ack (last byte to read)
    I2C1_Stop();                 // Send STOP bit to RTC chip

}

//
// This function converts the date-time into correct format for displaying on the LCD. The numbers
// in RTC memory are in BCD form and are converted as follows:
// "extract upper byte, shift right 4 bits, multiply by 10, add lower byte".
// For example, number 25 in RTC memory is stored as 37. i.e bit pattern: "0010 0101". After the
// conversion we obtain the required number 25.
//
Convert_Date_Time()
{
    seconds = ((seconds & 0xF0) >> 4)*10 + (seconds & 0x0F);
    minutes = ((minutes & 0xF0) >> 4)*10 + (minutes & 0x0F);
    hours = (((hours & 0xF0) >> 4)*10 + (hours & 0x0F));
    month = (((month & 0x10) >> 4)*10 + (month & 0x0F));
    year = (day & 0xC0) >> 6;
    day = (((day & 0x30) >> 4)*10 + (day & 0x0F));
}

//

```

**Figure 7.69**  
cont'd

```

// Display the date and time on the LCD
//
void Display_Date_Time()
{
//
// Write day, month, year as: dd=mm=xxx
//
    Lcd_Chr(1, 1, (day / 10) + '0');
    Lcd_Chr(1, 2, (day % 10) + '0');
    Lcd_Chr(1, 4, (month / 10) + '0');
    Lcd_Chr(1, 5, (month % 10) + '0');
    Lcd_Chr(1, 10, year + '0');
//
// Write hour, minutes, seconds as: hh:mm:ss
//
    Lcd_Chr(2, 1, (hours / 10) + '0');
    Lcd_Chr(2, 2, (hours % 10) + '0');
    Lcd_Chr(2, 4, (minutes / 10) + '0');
    Lcd_Chr(2, 5, (minutes % 10) + '0');
    Lcd_Chr(2, 7, (seconds / 10) + '0');
    Lcd_Chr(2, 8, (seconds % 10) + '0');
}

//
// This function gets the date and Time from the user via the 3 buttons. New values of Date
// and Time are returned to the calling program. Initially the maximum values are shown
// and these can be changed using the UP and DOWN buttons.
//
unsigned char Set_Date_Time(unsigned char *str, unsigned char min, unsigned char max)
{
    unsigned char c, Txt[4];

    ByteToStr(max, Txt);                                // Convert maximum value to string in Txt
    Lcd_Cmd(LCD_CLEAR);                                // Display field name (e.g. "DAY:")
    Lcd_Out(1,1,str);                                 // Display maximum value to start with
    c = max;

    while(MODE == 1)                                    // While MODE button is not pressed
    {
        if(UP == 0)                                     // If UP button is pressed (increment)
        {
            Delay_Ms(10);
            while(UP == 0);                            // Wait until UP button is released
            c++;
            if(c > max)c = min;
        }
        if(DOWN == 0)                                  // If DOWN button is pressed
        {
            Delay_Ms(10);
            while(DOWN == 0);                         // Wait until DOWN button is released
        }
    }
}

```

**Figure 7.69**

cont'd

```

        c--;
        if(c < min || c == 255)c = max;
    }
    ByteToStr(c, Txt);
    Lcd_Out(2,1,Txt);
}
Delay_Ms(10);
while(MODE == 0);
return c;
}

// This function sets the RTC with the new Date and Time values. The number to be sent to
// the RTC chip is divided by 10, shifted left 4 digits, and the remainder is added to it. Thus
// for example if the number is decimal 25, it is converted into bit pattern "0010 0101" and
// stored at the appropriate RTC memory
//
void Set_RTC()
{
//
// Convert Date and Time into a format compatible with the RTC chip
//
    seconds = ((newseconds / 10) << 4) + (newseconds % 10);
    minutes = ((newminutes / 10) << 4) + (newminutes % 10);
    hours = ((newhour / 10) << 4) + (newhour % 10);
    month = ((newmonth / 10) << 4) + (newmonth % 10);
    day = (newyear << 6) + ((newday / 10) << 4) + (newday % 10);

    I2C1_Start();                                // Send START bit to RTC chip
    I2C1_Wr(0xA0);                             // Address the RTC chip
    I2C1_Wr(0x00);                             // Start from address 0
    I2C1_Wr(0x80);                             // Pause RTC counter
    I2C1_Wr(0x00);                             // Write to hundredths memory location
    I2C1_Wr(seconds);                          // Write to seconds memory location
    I2C1_Wr(minutes);                          // Write to minutes memory location
    I2C1_Wr(hours);                           // Write to hours memory location;
    I2C1_Wr(day);                            // Write to year/day memory location
    I2C1_Wr(month);                           // Write to month memory location
    I2C1_Stop();                               // Send STOP bit

    I2C1_Start();                                // Send START bit to RTC chip
    I2C1_Wr(0xA0);                             // Address the RTC chip
    I2C1_Wr(0);                                // Start from address 0 (Configuration reg)
    I2C1_Wr(0);                                // Write 0 to Configuration reg to start
    counter
    I2C1_Stop();                               // Send STOP bit
}

// This function sets the RTC alarm. The alarm is configured to occur every day

```

**Figure 7.69**

cont'd

```

// at the set time. The BUZZER sounds when the alarm occurs
//
void Set_RTC_Alarm()
{
//
// Convert Alarm Time into a format compatible with the RTC chip. For daily Alarm
// setup the Date fields (day, month, year) are ignored
//
seconds = ((newseconds / 10) << 4) + (newseconds % 10);
minutes = ((newminutes / 10) << 4) + (newminutes % 10);
hours = ((newhour / 10) << 4) + (newhour % 10);

I2C1_Start();                                     // Send START bit to RTC chip
I2C1_Wr(0xA0);                                  // Address the RTC chip
I2C1_Wr(0x00);                                  // Start from address 0
I2C1_Wr(0x04);                                  // Enable Alarm Control register
I2C1_Stop();

I2C1_Start();                                     // Address the RTC chip
I2C1_Wr(0xA0);                                  // Start from address 8
I2C1_Wr(0x90);                                  // Enable Daily alarms, enable INT output
I2C1_Wr(0x00);                                  // Write to hundredths memory location
I2C1_Wr(seconds);                                // Write to seconds memory location
I2C1_Wr(minutes);                                // Write to minutes memory location
I2C1_Wr(hours);                                 // Write to hours memory location;
I2C1_Stop();                                     // Send STOP bit
}

//
// This function stops the alarm by clearing the timer register of the
// control and status register
//
void Reset_Alarm_Flag()
{
I2C1_Start();                                     // Send START bit to RTC chip
I2C1_Wr(0xA0);                                  // Address the RTC chip
I2C1_Wr(0x00);                                  // Start from address 0
I2C1_Wr(0x04);                                  // Reset Alarm flag
I2C1_Stop();                                     // Send STOP bit
}

void main()
{
    ANSELB = 0;                                    // Configure PORTB as digital
    ANSELC = 0;                                    // Configure PORTC as digital
    ANSELD = 0;                                    // Configure PORTD as digital
    TRISC.RC2 = 1;                                 // Alarm INT input
    TRISD.RD0 = 1;                                 // Configure RD0 (MODE)as input
    TRISD.RD1 = 1;                                 // Configure RD1 (UP) as input
}

```

**Figure 7.69**

cont'd

```

TRISD.RD2 = 1;                                // Configure RD2 (DOWN) as input
TRISD.RD3 = 1;                                // Configure RD3 (Alarm setup)
TRISD.RD4 = 1;                                // Configure RD4 as input (Alarm STOP)
TRISD.RD7 = 0;                                // Alarm LED is output

ALARM_LED = 0;                                 // Alarm LEDOFF to start with
Lcd_Init();                                    // Initialize LCD
Lcd_Cmd(_LCD_CURSOR_OFF);                     // Disable cursor
Lcd_Cmd(_LCD_CLEAR);                          // Clear LCD

I2C1_Init(100000);                           // Initialize I2C module
// If the MODE button is pressed on startup we must get into SETUP phase
//
if(MODE == SETUP)                            // If CLOCK SETUP mode
{
    while(MODE == SETUP);                    // Wait until MODE button is released
    newday = Set_Date_Time("DAY:",1,31);      // Get current day
    newmonth = Set_Date_Time("MONTH:", 1,12);  // Get current month
    newyear = Set_Date_Time("YEAR (201x):", 3,6); // Get current year (201x)
    newhour = Set_Date_Time("HOUR:", 0, 23);   // Get current hour
    newminutes = Set_Date_Time("MINUTES:", 0, 59); // Get current minutes
    newseconds = Set_Date_Time("SECONDS:", 0, 59); // Get current seconds
    //
    // We have got all the new Date and Time values. Now set the RTC with these values
    //
    Set_RTC();
}
if(ALARM_SETUP == 0)                         // If ALARM SETUP mode
{
    while(ALARM_SETUP == 0);                // Wait until ALARM button is released
    newhour = Set_Date_Time("ALRM HOUR:", 0, 23);
    newminutes = Set_Date_Time("ALRM MINS:", 0, 59);
    newseconds = Set_Date_Time("ALRM SECS:", 0, 59);
    //
    // We have got the Daily Alarm time. Now set the RTC clock to generate alarm
    // at this time every day. Sound the BUZZER when alarm is generated.
    //
    Set_RTC_Alarm();
}

//
// Read the Date and Time from the RTC chip and display on the LCD in the following format:
// Row 1: dd-mm-yyyy
// Row 2: hh:mm:ss
//
Lcd_Out(1, 1, "dd-mm-2013");
Lcd_Out(2, 1, "hh:mm:ss");
while(1)

```

**Figure 7.69**  
cont'd

```
{  
    Read_Date_Time(); // Read Date and Time from RTC chip  
    Convert_Date_Time(); // Convert into a form to display  
    Display_Date_Time(); // Display Date and Time on the LCD  
  
    if(ALARM_INT == 0) // If alarm occurred (active LOW)  
    {  
        ALARM_LED = 1; // Turn ON ALARM LED  
        if(STOP_ALARM == 0) // if STOP ALARM button pressed  
        {  
            ALARM_LED = 0; // Turn OFF ALARM LED  
            Reset_Alarm_Flag(); // Reset alarm flag back to 1  
        }  
    }  
}
```

**Figure 7.69**  
cont'd

- Electronic games,
- Personal digital assistants (PDAs),
- Mobile phones,
- Embedded electronic systems.

Figure 7.70 shows the picture of a typical SD card.



**Figure 7.70:** A Typical SD Card.

**Table 7.5: Different Size SD Card Specifications.**

	<b>Standard SD</b>	<b>miniSD</b>	<b>mikroSD</b>
Dimensions	32 × 24 × 2.1 mm	21.5 × 20 × 1.4 mm	15 × 11 × 1 mm
Card weight	2.0 g	0.8 g	0.25 g
Operating voltage	2.7–3.6 V	2.7–3.6 V	2.7–3.6 V
Write protect	Yes	No	No
Pins	9	11	8
Interface	SD or SPI	SD or SPI	SD or SPI
Current consumption	100 mA (Write)	100 mA (Write)	100 mA (Write)

The SD card is a flash memory storage device designed to provide a high capacity, nonvolatile, and rewritable storage in small size. The memory capacity of the SD cards is increasing all the time. Currently, they are available in capacities from several gigabytes to >128 GB. SD cards are available in three sizes: *standard SD card*, *miniSD card*, and the *microSD card*. **Table 7.5** lists the main specifications of different size cards.

SD card specifications are maintained by the *SD Card Association*, which has >600 members. MiniSD and microSD cards are electrically compatible with the standard SD cards, and they can be inserted in special adapters and used as standard SD cards in standard card slots.

SD card speeds are measured in three different ways: in kilobytes per second (kB/s), in megabytes per second (MB/s), or in an “x” rating similar to that of CD-ROMS where “x” is the speed corresponding to 150 kB/s. Thus, the various “x” based speeds are as follows:

- 4x: 600 kB/s,
- 16x: 2.4 MB/s,
- 40x: 6.0 MB/s,
- 66x: 10 MB/s.

As far as the memory capacity is concerned, we can divide SD cards into three families: Standard-Capacity (SDSC), High-Capacity (SDHC), and eXtended-Capacity (SDXC). SDSC are the older cards with capacities 1–2 GB. SDHC have capacities 4–32 GB, and SDXC cards have capacities >32–128 GB. The SD and SDHC families are available in all three sizes, but the SDXC family is not available in the mini size.

In the projects in this book, we shall be using the standard SD cards only. The use of the smaller size SD cards is virtually the same and is not described here any further.

SD cards can be interfaced to microcontrollers using two different protocols: SD card protocol and the SPI protocol. The SPI protocol is the most commonly used protocol and

is the one used in the projects in this book. SPI bus is currently used by microcontroller interface circuits to talk to a variety of devices such as

- Memory devices (SD cards),
- Sensors,
- RTCs,
- Communications devices,
- Displays.

The advantages of the SPI bus are as follows:

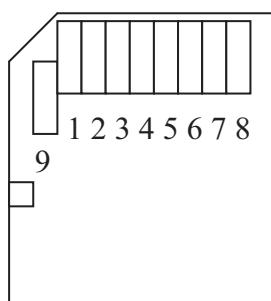
- Simple communication protocol,
- Full duplex communication,
- Very simple hardware interface.

In addition, the disadvantages of the SPI bus are

- Requires four pins,
- No hardware flow control,
- No slave acknowledgement.

It is important to realize that there are no SPI standards governed by any international committee. As a result of this, there are several versions of the SPI bus implementation. In some applications, the MOSI and MISO lines are combined into a single data line, thus reducing the line requirements into three. Some implementations have two clocks, one to capture (or display) data and another to clock it into the device. Also, in some implementations, the chip select line may be active-high rather than active low.

The standard SD card has nine pins with the pin layout shown in [Figure 7.71](#). Depending on the interface protocol used, pins have different functions. [Table 7.6](#) gives the function of each pin in both the SD mode and the SPI mode of operation.



**Figure 7.71: Standard SD Card Pin Layout.**

**Table 7.6: Standard SD Card Pin Definitions.**

<b>Pin</b>	<b>Name</b>	<b>SD Description</b>	<b>SPI Description</b>
1	CD/DAT3/CS	Data line 3	Chip select
2	CMD/Datain	Command/response	Host to card command and data
3	V <sub>SS</sub>	Supply ground	Supply ground
4	V <sub>DD</sub>	Supply voltage	Supply voltage
5	CLK	Clock	Clock
6	V <sub>SS2</sub>	Supply voltage ground	Supply voltage ground
7	DAT0	Data line 0	Card to host data and status
8	DAT1	Data line 1	Reserved
9	DAT2	Data line 2	Reserved

### ***Operation of the SD Card in the SPI Mode***

When the SD card is operated in the SPI mode only seven pins are used:

- Two power supply ground (pins 3 and 6),
- Power supply (pin 4),
- Chip select (pin 1),
- Data out (pin 7),
- Data in (pin 2),
- CLK (pin 5).

Three pins are used for the power supply, leaving four pins for the SPI mode of operation:

- Chip select (pin 1),
- Data out (pin 7),
- Data in (pin 2),
- CLK (pin 5).

At power-up, the SD card defaults to the SD bus protocol. The card is switched to the SPI mode if the CS signal is asserted during the reception of the reset command. When the card is in the SPI mode, it only responds to SPI commands. The host may reset a card by switching the power supply off and on again.

Most high-level language compilers normally provide a library of commands for initializing, reading, and writing to SD cards. In general, it is not necessary to know the internal structure of an SD card before it can be used since the available library functions can easily be used. It is however important to have some knowledge about the internal structure of an SD card so that it can be used efficiently. In this section, we shall be looking briefly at the internal architecture and the operation of SD cards.

An SD card has a set of registers that provide information about the status of the card. When the card is operated in the SPI mode, these registers are as follows:

- Card Identification Register (CID),
- Card Specific Data Register (CSD),
- SD Configuration Register (SCR),
- Operation Control Register (OCR).

The CID consists of 16 bytes, and it contains the manufacturer ID, product name, product revision, card serial number, manufacturer date code, and a checksum byte.

The CSD consists of 16 bytes, and it contains card-specific data such as the card data transfer rate, read/write block lengths, read/write currents, erase sector size, file format, write protection flags, and checksum.

The SCR is 8 bytes long, and it contains information about the SD card's special features capabilities such as the security support, and data bus widths supported.

The OCR is only 4 bytes long, and it stores the V<sub>DD</sub> voltage profile of the card. The OCR shows the voltage range in which the card data can be accessed.

All SD card SPI commands are 6 bytes long with the MSB transmitted first. The first byte is known as the “command” byte, and the remaining 5 bytes are “command arguments”. Bit 6 of the command byte is set to “1” and the MSB bit is always “0”. With the remaining 6 bits, we have 64 possible commands, named CMD0 to CMD63. Some of the important commands are

- CMD0      GO\_IDLE\_STATE (Resets the SD card),
- CMD1      SEND\_OP\_COND (Initializes the card),
- CMD9      SEND\_CSD (Get CSD data),
- CMD10     SEND\_CID (Get CID data),
- CMD16     SET\_BLOCKLEN (Selects a block length in bytes),
- CMD17     READ\_SINGLE\_BLOCK (Reads a block of data),
- CMD24     WRITE\_BLOCK (Writes a block of data),
- CMD32    ERASE\_WR\_BLK\_START\_ADDR (Sets the address of the first write block to be erased),
- CMD33    ERASE\_WR\_BLK\_END\_ADDR (Sets the address of the last write block to be erased),
- CMD38    ERASE (Erases all previously selected blocks).

In response to a command, the card sends a status byte known as R1. The MSB bit of this byte is always “0” and the other bits indicate various error conditions.

### **Reading Data**

The SD card in the SPI mode supports single block and multiple block read operations. The host should set the block length, and after a valid read command, the card responds with a response token, followed by a data block and a CRC check. The block length can be between 1 and 512 bytes. The starting address can be any valid address range of the card.

In multiple block read operations, the card sends data blocks with each block having its own CRC check attached to the end of the block.

### **Writing Data**

The SD card in the SPI mode supports single or multiple block write operations. After receiving a valid write command from the host, the card will respond with a response token and will wait to receive a data block. A 1 byte “start block” token is added to the beginning of every data block. After receiving the data block, the card responds with a “data response” token and the card will be programmed as long as the data block has been received with no errors.

In multiple write operations, the host sends the data blocks one after the other, each preceded with a “start block” token. The card sends a response byte after receiving each data block.

A card can be inserted and removed from the bus without any damage. This is because all data transfer operations are protected by CRC codes, and any bit changes as a result of inserting or removing a card can easily be detected. SD cards operate with a typical supply voltage of 2.7 V. The maximum allowed power supply voltage is 3.6 V. If the card is to be operated from a standard 5.0-V supply, a voltage regulator should be used to drop the voltage to 2.7 V.

The use of an SD card requires the card to be inserted into a special card holder with external contacts ([Figure 7.72](#)). Connections can then be made easily to the required card pins.



**Figure 7.72: SD Card Holder.**

### Project Description

This project is about using the card filing system. In this project, a file called “MYFILE55.TXT” is created on an SD card. Text “This is MYFILE.TXT” is written to the file initially. The text “This is the added data...” is appended to the file. The file can be opened on a PC and its contents can be verified.

### Project Hardware

Figure 7.73 shows the circuit diagram of the project. The SD card is inserted into a card holder, and the holder is connected to PORTC of the microcontroller. The interface between the SD card and the microcontroller ports is as follows:

SD Card Pin	Microcontroller Pin
CS	RC2
CLK	RC3
DO	RC3
DI	RC5

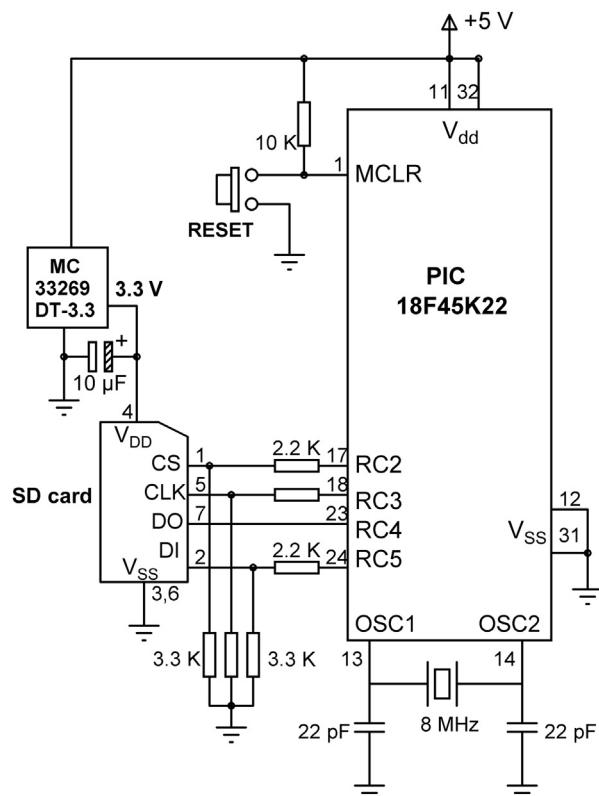


Figure 7.73: Circuit Diagram of the Project.

```

BEGIN
  Define CS port pin and direction
  Define file open argument definitions
  Configure PORTC as digital
  Initialize SPI bus
  Initialize Mmc_FAT library
  Create new file (if it does not exist)
  Position the cursor at the beginning for writing
  Write text "This is MYFILE.TXT." to the file
  Write "This is the added data..." to the file
  Close the file
END

```

**Figure 7.74: Project PDL.**

According to the SD card specifications, when the card is operating with a supply voltage of  $V_{DD} = 3.3$  V, the input–output pin voltage levels are as follows:

- Minimum produced output HIGH voltage,  $VOH = 2.475$  V,
- Maximum produced output LOW voltage,  $VOL = 0.4125$  V,
- Minimum required input HIGH voltage,  $VIH = 2.0625$ ,
- Maximum input HIGH voltage,  $VIH = 3.6$  V,
- Maximum required input LOW voltage,  $VIL = 0.825$  V.

Although the output produced by the card (2.475 V) is sufficient to drive the input port of a PIC microcontroller, the logic HIGH output of the microcontroller (about 4.3 V) is too high for the SD card inputs (maximum 3.6 V). As a result of this, a potential divider is setup at the three inputs of the SD card using 2.2 and 3.3 K resistors. Thus, the maximum voltage at the inputs of the SD card is limited to about 2.5 V:

$$\text{SD card input voltage} = 4.3 \text{ V} \times 3.3 \text{ K} / (2.2 \text{ K} + 3.3 \text{ K}) = 2.48 \text{ V}$$

The microcontroller is powered from a 5-V supply, which is obtained using a 7805 type 5-V regulator with a 9-V input. The 2.7- to -3.6-V supply required by the SD card is obtained using a MC33269DT-3.3 type regulator with a 3.3-V output, and is driven from the 5-V input voltage.

### **Project PDL**

The project PDL is shown in [Figure 7.74](#).

### **Project Program**

#### *mikroC Pro for PIC*

The program listing of the project is given in [Figure 7.75](#) (MIKROC-SD1.C). mikroC Pro for PIC language provides an extensive set of library functions to read and write data to

```
*****
SD CARD PROJECT - WRITE TEXT TO A FILE
=====
```

In this project an SD card is connected to PORT C as follows:

CS RC2  
CLK RC3  
DO RC4  
DI RC5

The program creates a new file called MYFILE55.TXT on the SD card and writes the text "This is MYFILE.TXT." to the file. Then the string "This is the added data..." is appended to the file.

```
Author: Dogan Ibrahim
Date: October 2013
File: MIKROC-SD1.C
*****
// MMC module connections
sbit Mmc_Chip_Select at LATC2_bit;
sbit Mmc_Chip_Select_Direction at TRISC2_bit;
// MMC module connections

#define FILE_READ 0x01           // read only
#define FILE_WRITE 0x02          // write only
#define FILE_APPEND 0x04         // append to file

char filename[] = "MYFILE55.TXT";
unsigned char txt[] = "This is the added data...";
unsigned short character;
unsigned long file_size,i;

void main()
{
    unsigned char fhandle;

    ANSEL0 = 0;                  // Configure PORTC as digital
//
// Initialise the SPI bus
//
    SPI1_Init_Advanced(_SPI_MASTER_OSC_DIV64, _SPI_DATA_SAMPLE_MIDDLE,
                       _SPI_CLK_IDLE_LOW, _SPI_LOW_2_HIGH);
//
// Initialize the Mmc library. Wait until card detected
//
    while(Mmc_Fat_Init());
//
// Create new file (if it does not exist)
//
    fhandle = MMc_Fat_Open(&filename,FILE_WRITE,0x80);
//
// Clear the file, start at the beginning for writing
//
    Mmc_Fat_Rewrite();
//
// Write to the file, specifying the length of the text
```

Figure 7.75: mikroC Pro for PIC Program.

```

// Mmc_Fat_Write("This is MYFILE.TXT.",19);
//
// Add more data to the end...
//
Mmc_Fat_Append();
Mmc_Fat_Write(txt,sizeof(txt));
//
// Now close the file (releases the handle)
//
Mmc_Fat_Close();

while(1);                                // Wait here forever
}

```

**Figure 7.75**  
cont'd

SD cards (and also MultiMedia Cards, MMC). Data can be written or read from a given sector of the card, or the file system on the card can be used for more sophisticated applications.

mikroC Pro for PIC compiler supports an SD card library (called the Mmc library) with many functions. Some commonly used functions for file handling are listed below:

- Mmc\_Fat\_Init (Initialize the card)
- Mmc\_Fat\_QuickFormat (Format the card)
- Mmc\_Fat\_Assign (Assign the file we will be working with)
- Mmc\_Fat\_Reset (Reset the file pointer. Opens the currently assigned file for reading)
- Mmc\_Fat\_Rewrite (Reset the file pointer and clear assigned file. Opens the assigned file for writing)
- Mmc\_Fat\_Append (Move file pointer to the end of assigned file so that new data can be appended to the file)
- Mmc\_Fat\_Read (Read the byte at which file pointer points to)
- Mmc\_Fat\_Write (Write a block of data to the assigned file)
- Mmc\_Fat\_Delete (Delete a file)
- Mmc\_Set\_File\_Date (Write system timestamp to a file)
- Mmc\_Fat\_Get\_File\_Dat (Read file timestamp)
- Mmc\_Fat\_Get\_File\_Size (Get file size in bytes)
- Mmc\_Fat\_Tell (Get cursor position in a file)
- Mmc\_Fat\_Seek (Set cursor position in a file)
- Mmc\_Fat\_Rename (Rename a file)
- Mmc\_Fat\_MakeDir (Create a new directory)
- Mmc\_Fat\_Exists (Returns information about a file's existence)
- Mmc\_Fat\_Activate (Select a file when multiple files are used)

- Mmc\_Fat\_ReadN (Read multiple bytes)
- Mmc\_Fat\_Open (Open/create a file)
- Mmc\_Fat\_Close (Close currently open file)
- Mmc\_Fat\_EOF (Check if end of file is reached)

The SPI module has to be initialized through `SPIx_Init_Advanced` routine with the following parameters. Once the MMC/SD card is initialized, the SPI module can be operated at higher speeds:

- SPI Master,
- Primary prescaler 64,
- Data sampled in the middle of data output time,
- Clock idle low,
- Serial output data changes on transition from low to high edge.

In this project, a new file is created, text is written inside the file, and then the file is closed. At the beginning of the program, the program creates file MYFILE55.TXT by calling library function `Mmc_Fat_Open` with the arguments as the *filename* and the creation flag 0x80, which tells the function to create a new file if the file does not exist. The filename should be in “filename.extension” format, although it is also possible to specify an eight-digit filename and a three-digit extension with no “.” in between as the “.” will be inserted by the function. Other allowed values of the creation flag are given in [Table 7.7](#). The `Mmc_Fat_Open` function returns a file handle. In multiple file operations, we can select the file we wish to operate on by specifying this handle in the `Mmc_Fat_Activate` function call. Note that the SD card must have been formatted in FAT16 before we can read or write to the card. Most new cards are already formatted, but we can also use the `Mmc_Fat_QuickFormat` function to format a card.

Function `Mmc_Fat_Rewrite` is called to clear the file and position the cursor to the beginning, ready for writing. Initial text is written to the file using function

**Table 7.7: Mmc\_Fat\_Open File Creation Flags.**

Flag	Description
0x01	Read only
0x02	Hidden
0x04	System
0x08	Volume label
0x10	Subdirectory
0x20	Archive
0x40	Device (internal use only, never found on disk)
0x80	File creation flag. If the file does not exist and this flag is set, a new file with specified name will be created.

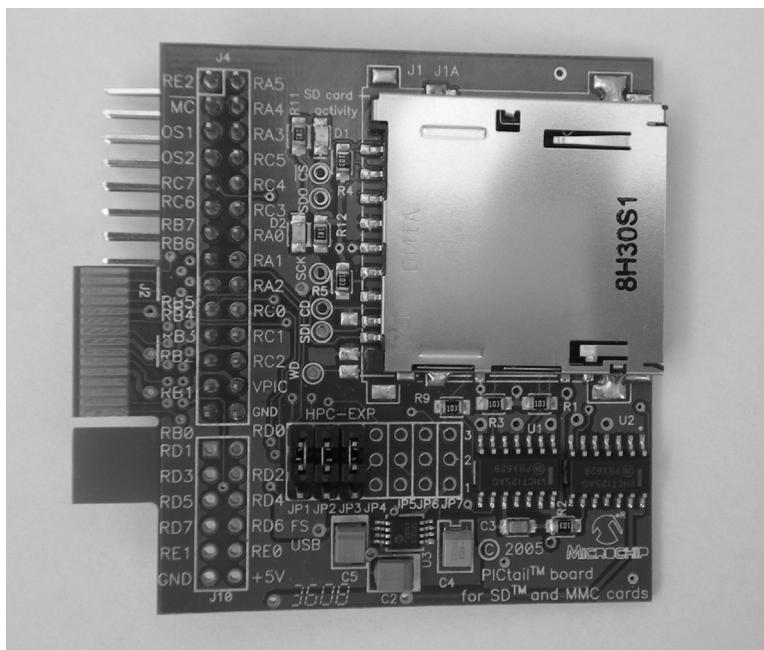
Mmc\_Fat\_Write. Then, function Mmc\_Fat\_Append is called to append the second text to the file. Finally, function Mmc\_Fat\_Close is called to close the file and release the handle.

Note that one of the arguments to the Mmc\_Fat\_Open function is the file mode (FILE\_READ, FILE\_WRITE, or FILE\_APPEND). The following definitions must be made at the beginning of the program before one of these arguments can be used:

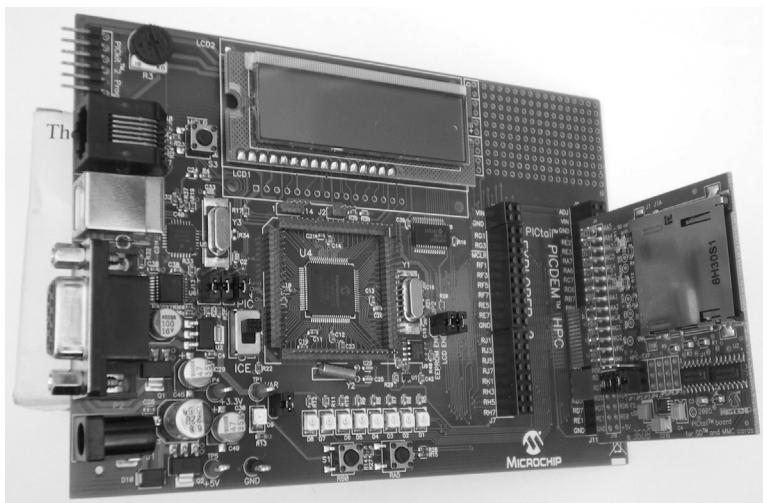
```
#define FILE_READ 0x01    // read only
#define FILE_WRITE 0x02   // write only
#define FILE_APPEND 0x04  // append to file
```

### MPLAB XC8

For the MPLAB XC8 version of the program, we shall be using the PICDEM PIC18 Explorer board (Chapter 5). A Microchip daughter SD card board (known as the PICtail Daughter Board for SD & MMC Cards, see [Figure 7.76](#)) is used as the SD card interface. This board directly plugs onto the PICDEM PIC18 Explorer board ([Figure 7.77](#)) and provides the SD card interface to the demonstration board (*note that there are minor design faults with the voltage level conversion circuitry on some of the PICtail Daughter Boards for SD & MMC Cards. You can get around these problems by providing a 3.3-V supply for the daughter board directly from the PICDEM Explorer board. Cut short the*



**Figure 7.76: PICtail Daughter Board for SD & MMC Cards.**



**Figure 7.77: The Daughter Board Plugs onto the PICDEM Board.**

*power supply pin of the daughter board connector, and connect this pin to the +3.3-V test point on the PICDEM Explorer board).*

The SD card daughter board has an on-board positive-regulated charge pump direct current (DC)/DC converter chip (MCP1253) used to convert the +5-V supply to +3.3 V required for the SD card. In addition, the board has buffers to provide correct voltages for the SD card inputs. Seven jumpers are provided on the board to select the SD card signal interface. The following jumpers should be selected:

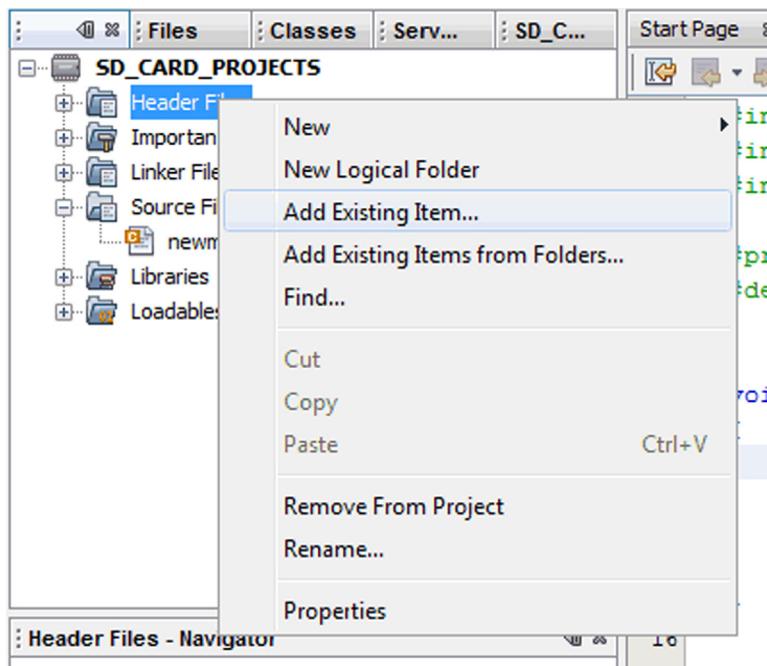
Jumper	Description
JP1	SCK connected to RC3
JP2	SDI connected to RC4
JP3	SDO connected to RC5
JP4	Card detect to RB4 (not used)
JP5	Write protect to RA4 (not used)
JP6	CS connected to RB3
JP7	Shutdown (not used)

The default jumper positions are connected by circuit tracks on the board, and these tracks should be cut to change the jumper positions if different connections are desired. Signals “card detect”, “write protect”, and “shutdown” are not used in this book, and the jumper settings can be left as they are. We shall be programming the PIC18 Explorer board using the ICD 3 programmer/debugger as described in earlier chapters.

The MPLAB XC8 version of the SD card program is slightly more complex. This is because a number of included files will have to be extracted and loaded in their correct

places. The steps in loading the required files are given below (see Microchip Inc “Application Note 1045 Implementing File I/O Functions Using Microchip’s Memory Disk Drive File System Library”, located in folder “microchip\_solutions\_v2013-06-15\Microchip\MDD File System\Documentation” for further information).

- Download and install the “microchip solutions” library from the Microchip Inc website. At the time of writing this book, this library had the name “microchip\_solutions\_v2013-06-15”. The installation process should create a folder called “microchip\_solutions\_v2013-06-15” under the main C:\ folder.
- Create a new MPLAB XC8 project as before. Choose the processor type as PIC18F8722 since the PICDEM PIC18 Explorer board is shipped with this microcontroller. Create a new main C source file.
- Right click Header Files on the left top window of MPLAB X IDE and select Add Existing Item ([Figure 7.78](#)). Find folder “microchip\_solutions\_v2013-06-15” and select the following files from the subfolders (one at a time):
  - C:\Microchip Solutions\Microchip\MDD File System\FSIO.c
  - C:\Microchip Solutions\Microchip\MDD File System\SD-SPI.c
  - C:\Microchip Solutions\Microchip\PIC18 salloc\salloc.c
  - C:\Microchip Solutions\Microchip\Include\Compiler.h
  - C:\Microchip Solutions\Microchip\Include\GenericTypeDefs.h



**Figure 7.78: Add the Required Header Files.**

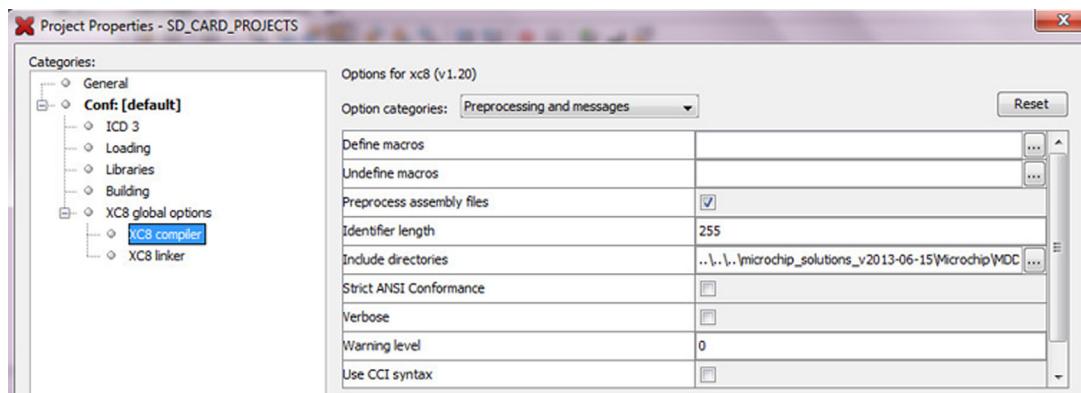


Figure 7.79: Select File Properties and XC8 Compiler.

C:\Microchip Solutions\MDD File System-SD Card\Pic18f\FSconfig.h  
 C:\Microchip Solutions\MDD File System-SD Card\Pic18f\HardwareProfile.h  
 C:\Microchip Solutions\Microchip\Include\MDD File System\FSDefs.h  
 C:\Microchip Solutions\Microchip\Include\MDD File System\SD-SPI.h  
 C:\Microchip Solutions\Microchip\Include\MDD File System\FSIO.h  
 C:\Microchip Solutions\Microchip\Include\INCLUDE\salloc\salloc.h

- Set the file paths in the compiler by: Select File → Project Properties in MPLAB X IDE and then select XC8 compiler as shown in [Figure 7.79](#).
- Click Include Directories, and browse and include the following folders ([Figure 7.80](#)):  
 microchip\_solutions\_v2013-06-15\Microchip\MDD File System  
 microchip\_solutions\_v2013-06-15\Microchip\INCLUDE\salloc

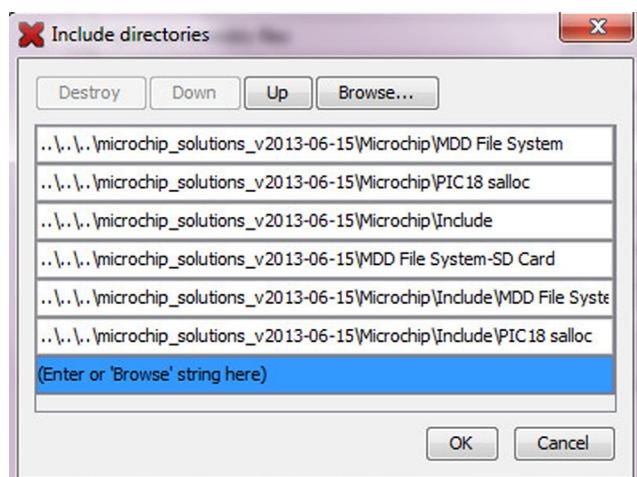


Figure 7.80: Set Folders to be Included.

```

// File: 18f8722.lkr
// Sample linker script for the PIC18F8722 processor

LIBPATH .

FILES c018i.o
FILES clib.lib
FILES p18f8722.lib

CODEPAGE NAME=page START=0x0 END=0xFFFF
CODEPAGE NAME=idlocs START=0x200000 END=0x200007 PROTECTED
CODEPAGE NAME=config START=0x300000 END=0x30000D PROTECTED
CODEPAGE NAME=devid START=0x3FFFFE END=0x3FFFFF PROTECTED
CODEPAGE NAME=eedata START=0xF00000 END=0xF003FF PROTECTED

ACCESSBANK NAME=accessram START=0x0 END=0x5F
DATABANK NAME=gpr0 START=0x60 END=0xFF
DATABANK NAME=gpr1 START=0x100 END=0x1FF
DATABANK NAME=gpr2 START=0x200 END=0x2FF
DATABANK NAME=gpr3 START=0x300 END=0x3FF
DATABANK NAME=gpr4 START=0x400 END=0x4FF
DATABANK NAME=gpr5 START=0x500 END=0x5FF
DATABANK NAME=gpr6 START=0x600 END=0x6FF
DATABANK NAME=buffer1 START=0x700 END=0x8FF PROTECTED
DATABANK NAME=buffer2 START=0x900 END=0xAFF PROTECTED
DATABANK NAME=gpr7 START=0xB00 END=0xBF
DATABANK NAME=gpr8 START=0xC00 END=0xCF
DATABANK NAME=gpr9 START=0xD00 END=0xEFF
//DATABANK NAME=gpr9 START=0xE00 END=0xEFF
//DATABANK NAME=gpr10 START=0xF00 END=0xFFFF
DATABANK NAME=gpr11 START=0xF00 END=0xF5F
ACCESSBANK NAME=accesssfr START=0xF60 END=0xFFF PROTECTED

SECTION NAME=CONFIG ROM=config
SECTION NAME=_SRAM_ALLOC_HEAP RAM=gpr7
SECTION NAME=dataBuffer RAM=buffer1
SECTION NAME=FATBuffer RAM=buffer2

STACK SIZE=0x200 RAM=gpr9

```

**Figure 7.81: Modified Linker File.**

```

microchip_solutions_v2013-06-15\Microchip\Include
microchip_solutions_v2013-06-15\MDD File System-SD Card
microchip_solutions_v2013-06-15\Microchip\Include\MDD File System
microchip_solutions_v2013-06-15\Microchip\Include\PIC18 salloc

```

Finally, before writing our program, we have to modify the compiler linker file to include a 512-byte section for the data read and write and also a 512-byte section for the FAT allocation. This is done by editing the linker file 18f8722.lkr in the folder and adding lines for a `dataBuffer`, and an `FATBuffer`. In addition, it is required to add a section named `_SRAM_ALLOC_HEAP` to the linker file. The modified linker file is shown in [Figure 7.81](#) (check the last part of the file for the modifications).

## Setting the Configuration Files

It is now necessary to customize some of the header files for our requirements. You should make the following modifications when using the PICDEM PIC18 Explorer Demonstration board with the PICtail SD Card Daughter board (you are recommended to make copies of the original files before modifying them in case you may want to return to them):

- Modify File C:\Microchip Solutions\MDD File System-SD Card\Pic18f\FSconfig.h and enable the following defines:

```
1. #define FS_MAX_FILES_OPEN 2
2. #define MEDIA_SECTOR_SIZE 512
3. #define ALLOW_FILESEARCH
    #define ALLOW_WRITES
    #define ALLOW_DIRS
    #define ALLOW_PGMFUNCTIONS
4. #define USERDEFINEDCLOCK
5. Make sure that the file object allocation is dynamic. i.e.
    #if 1
```

- Modify File C:\Microchip Solutions\MDD File System-SD Card\Pic18f\HardwareProfile.h and set the following options (notice that the system clock is 10 MHz, but the configuration option OSC = HSPLL will be used in our projects to multiply the clock by a factor of four, and it should be set to 40 MHz):

1. Set clock rate to 10 MHz:

```
#define GetSystemClock() 40000000
```

2. Enable SD-SPI interface.

```
#define USE_SD_INTERFACE_WITH_SPI
```

3. Define SD card interface pins and SPI bus pins to be used:

#define SD_CS	PORTBbits.RB3
#define SD_CS_TRIS	TRISBbits.TRISB3
#define SD_CD	PORTBbits.RB4
#define SD_CD_TRIS	TRISBbits.TRISB4
#define SD_WE	PORTAbits.RA4
#define SD_WE_TRIS	TRISAbits.TRISA4
#define SPICON1	SSP1CON1
#define SPISTAT	SSP1STAT
#define SPIBUF	SSP1BUF
#define SPISTAT_RBF	SSP1STATbits.BF
#define SPICON1bits	SSP1CON1bits
#define SPISTATbits	SSP1STATbits
#define SPICLOCK	TRISCbits.TRISC3
#define SPIIN	TRISCbits.TRISC4
#define SPIOUT	TRISCbits.TRISC5
#define SPICLOCKLAT	LATCbits.LATC3
#define SPIINLAT	LATCbits.LATC4
#define SPIOUTLAT	LATCbits.LATC5
#define SPICLOCKPORT	PORTCbits.RC3
#define SPIIMPORT	PORTCbits.RC4
#define SPIEXPORT	PORTCbits.RC5

## MPLAB XC8 MDD Library Functions

Before writing our program, let us look at the MPLAB XC8 MDD library functions.

The MDD library provides a large number of “File and Disk Manipulation” functions that can be called and used from our programs. The functions can be collected into following groups:

- Initialize a card.
- Open/create/close/delete/locate/rename a file on the card.
- Read/write to an opened file.
- Create/delete/change/rename a directory on the card.
- Format a card.
- Set file creation and modification date and time.

Table 7.8–Table 7.13 give a summary of each function briefly.

**Table 7.8: Initialize a Card Function.**

Function	Description
FSInit	Initialize the card

**Table 7.9: Open/Create/Delete/Locate/Rename Functions.**

Function	Description
FSopen/FSfopenpgm	This function opens an existing file for reading, or appending at the end of the file, or creates a new file for writing.
FSfclose	Updates and closes a file. The file time-stamping information is also updated
FSRemove/FSremovepgm	Delete a file
FSrename	Change the name of a file
FindFirst/FindFirstpgm	Locate a file in the current directory that matches the specified name and attributes
FindNext	Locate the next file in the current directory that matches the name and attributes specified earlier

pgm versions are to be used with PIC18 microcontrollers where the arguments are specified in the ROM.

**Table 7.10: Read/Write Functions.**

Function	Description
FSfread	Reads data from an open file to a buffer
FSfwrite	Writes data from a buffer to an open file
FSfseek	Return the current position in a file
FSfprintf	Write a formatted string to a file

**Table 7.11: Create/Delete/Change/Rename Directory.**

Function	Description
FSmkdir	Create a new subdirectory in the current working directory
FSrmdir	Delete the specified directory
FSchdir	Change the current working directory
FSrename	Change the name of a directory
FSgetcwd	Return the name of the current working directory

**Table 7.12: Format a Card.**

Function	Description
FSformat	Format a card

**Table 7.13: File Time-Stamping Function.**

Function	Description
SetClockVars	Set the date and time that will be applied to files when they are created or modified

**Table 7.14: MDD Library Options (in File FSconfig.h).**

Library Option	Description
ALLOW_WRITES	Enables write functions to write to the card
ALLOWS_DIRS	Enables directory functions (Writes must be enabled)
ALLOW_FORMATS	Enable card formatting function (Writes must be enabled)
ALLOW_FILESEARCH	Enables file and directory search
ALLOW_PGMFUNCTIONS	Enabled pgm functions for getting parameters from the ROM
ALLOW_FSFPRINTF	Enables Fsfprintf function (Writes must be enabled)
SUPPORT_FAT32	Enables FAT32 functionality

### ***Library Options***

A number of options are available in the MDD library. These options are enabled or disabled by uncommenting or commenting them, respectively, in include file FSconfig.h. The available options are given in [Table 7.14](#).

### ***Microcontroller Memory Usage***

The MPLAB XC8 program memory and data memory usage with the MDD library functions is shown in [Table 7.15](#). Note that 512 bytes of data are used for the data buffer

**Table 7.15: MPLAB C18 Memory Usage with MDD Library.**

<b>Functions Included</b>	<b>Program Memory (bytes)</b>	<b>Data memory (bytes)</b>
Read-only mode (basic)	11099	2121
File search enabled	+2098	+0
Write enabled	+7488	+0
Format enabled	+2314	+0
Directories enabled	+8380	+90
pgm functions enabled	+288	+0
FSprintf enabled	+2758	+0
FAT32 support enabled	+407	+4

and an additional 512 bytes are used for the file allocation table buffer. The amount of required memory also depends on the number of files opened at a time. In [Table 7.15](#), it is assumed that two files are opened. The first row shows the minimum memory requirements, and additional memory will be required when any of the subsequent row functionality is enabled.

### ***Sequence of Function Calls***

The sequence of the function calls to read or write data to a file, or to delete an existing file are given in this section.

#### ***Reading From an Existing File***

The steps to open an existing file and read from it are as follows:

- Call FSInit to initialize the card and SPI bus
- Call FSfopen or FSfopenpgm to open the existing file in read mode
- Call FSfread to read data from the file
- Call FSfclose to close the file

The FSread function can be called as many times as required.

#### ***Writing to an Existing File***

The steps to open an existing file and append data to it are as follows:

- Call FSInit to initialize the card and SPI bus
- Call FSfopen or FSfopenpgm to open the existing file in append mode
- Call FSfwrite to write data to the file
- Call FSfclose to close the file

The FSwrite function can be called as many times as required.

### ***Deleting an Existing File***

The steps to delete an existing file are as follows:

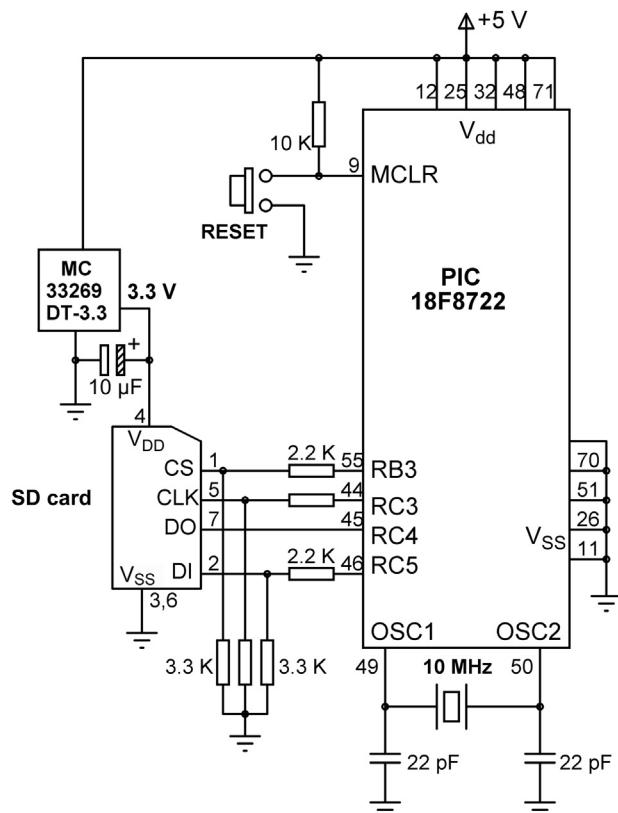
- Call FSInit to initialize the card and SPI bus
- Call FSfopen or FSfopenpgm to open the existing file in **write** mode
- Call FSremove or FSremovepgm to delete the file
- Call FSfclose to close the file

The circuit diagram of the XC8 version of the project is shown in [Figure 7.82](#).

The program listing for the MPLAB XC8 version of the project is shown in [Figure 7.83](#) (XC8-SD.C).

### ***Project 7.9—SD Card-Based Temperature Data Logger***

In this project, the design of a temperature data logger system is described. The ambient temperature is read every 10 s, and 100 records are stored on an SD card, or the contents



**Figure 7.82: Circuit Diagram of the XC8 Version of the Project.**

```

=====
PROJECT TO WRITE SHORT TEXT TO AN SD CARD
=====

In these projects a PIC18F8722 type microcontroller is used. The microcontroller
is operated with an 10 MHz crystal.

An SD card is connected to the microcontroller as follows:

SD card      microcontroller
CS           RB3
CLK          RC3
DO           RC4
DI           RC5

The program uses the Microchip MDD library functions to read and write to the
SD card.

In this version of the program an LED is connected to port RD0 and the LED is turned
ON when the program is terminated successfully.

Author: Dogan Ibrahim
Date: October 2013
File: XC8-SD.C
=====

#include <p18f8722.h>
#include <FSIO.h>

#pragma config WDT = OFF, OSC = HSPLL,LVP = OFF
#pragma config MCLRE = ON,CCP2MX = PORTC, MODE = MC

#define LED PORTDbits.RD0
#define ON 1
#define OFF 0

/* ===== START OF MAIN PROGRAM ===== */
// 
// Start of MAIN Program
//
void main(void)
{
    FSFILE *MyFile;
    unsigned char txt[]="This is a TEXT message";

    TRISD = 0;
    PORTD = 0;
//
// Initialize the SD card routines
//
//        while(!FSInit());
//
// Create a new file called MESSAGE.TXT
//
    MyFile = FSopenpgm("MESSAGE.TXT", "w+");
    if(MyFile == NULL)while(1);
//
// Write message to the file
//
//        if(FSfwrite((void *)txt, 1, 22, MyFile) != 22)while(1);
//
// Close the file
//
//        if(FSfclose(MyFile) != 0)while(1);
//
// Success. Turn ON the LED
//
//        LED = ON;
while(1);
}

```

**Figure 7.83: MPLAB XC8 Program.**

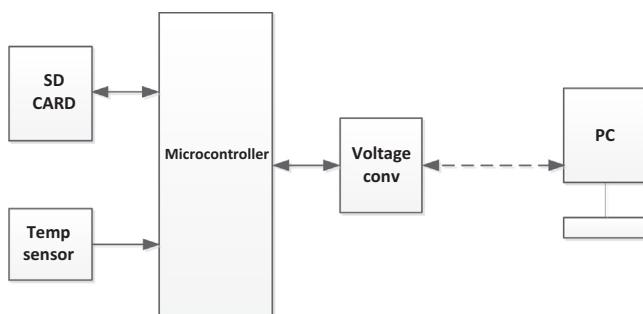


Figure 7.84: Block Diagram of the Project.

of an already saved file are sent to the PC. The program is menu based, and the user is given the options of (Figure 7.84)

- Send saved temperature readings on the SD card to a PC.
- Save temperature readings in a new file on SD card.
- Append the temperature readings to an existing file on SD card.

The block diagram of the project is shown in Figure 7.83.

### **Hardware Description**

The circuit diagram of the project is shown in Figure 7.85. An SD card is connected to the microcontroller. Additionally, the UART pins (RX6 and RX7) are connected to an RS232

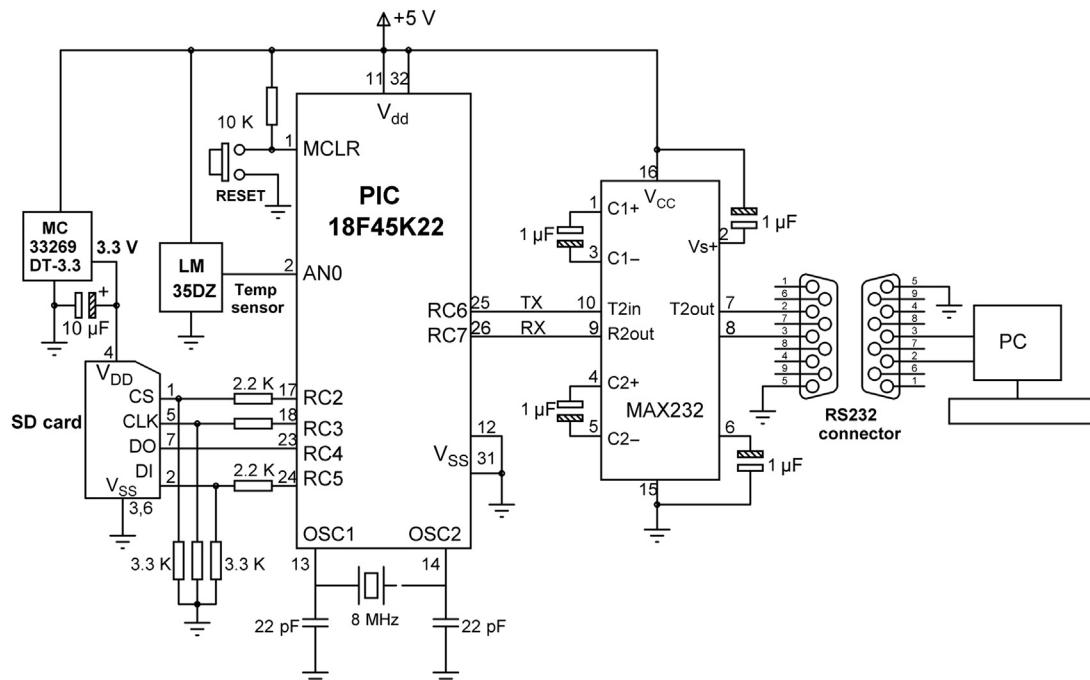


Figure 7.85: Circuit Diagram of the Project.

connector via a MAX232 voltage level translator chip. The temperature is sensed via the LM35DZ-type analog temperature sensor, connected to AN0 pin.

LM35DZ is a three-pin analog temperature sensor that can measure temperature with a 1 °C accuracy in the interval 0 to +100 °C. One pin of the device is connected to the supply (+5 V), the other pin to the ground and the third pin is the analog output. The output voltage of the sensor is directly proportional to the temperature, that is,  $V_o = 10 \text{ mV/}^{\circ}\text{C}$ . If, for example, the temperature is 10 °C, the output voltage will be 100 mV. Similarly, if the temperature is 35 °C, the output voltage of the sensor will be 350 mV.

### **Project Program**

#### *mikeoC Pro for PIC*

When the program is started, the following menu will be displayed on the PC screen:

TEMPERATURE DATA LOGGER

1. Send temperature data to the PC
  2. Save temperature data in a new file
  3. Append temperature data to an existing file
- Choice ?

The user is then expected to choose the required option. At the end of an option, the program does not return to the menu and the system should be restarted to display the menu again.

The mikroC Pro for PIC program listing of the project is shown in [Figure 7.86](#) (MIKROC-SD2.C). In this project, a file called “TEMPERTR.TXT” is created on the SD card to store the temperature readings.

The following functions are created at the beginning of the program, before the *main* program:

Newline: This function sends a carriage return and a line feed to the UART so that the cursor moves to the next line.

Get\_Temperature: This function starts the A/D conversion and receives the converted data into a variable called *Vin*. The voltage corresponding to this value is then calculated in millivolts and divided by 10 to find the actual measured temperature in degrees Celsius. The decimal part of the temperature found is then converted into string form using function LongToStr. The leading spaces are removed from this string, and the resulting string is stored in character array *temperature*. Then the fractional parts of the measured temperature, a carriage return, and a line feed are added to this character array, which is later written onto the SD card.

```
*****
TEMPERATURE DATA LOGGER PROJECT
=====
```

In this project an SD card is connected to PORTC as follows:

CS	RC2
CLK	RC3
DO	RC4
DI	RC5

In addition, a MAX232 type RS232 voltage level converter chip is connected to serial ports RC6 and RC7. Also, a LM35DZ type analog temperature sensor is connected to analog input AN0 of the microcontroller.

The program is menu based. The user is given options of either to send the saved temperature data to a PC, or to read and save new data on the SD card, or to read temperature data and append to the existing file. Temperature is read at every 10 s.

The temperature is stored in a file called "TEMPERTR.TXT"

Author: Dogan Ibrahim  
 Date: September 2013  
 File: MIKROC-SD2.C

```
*****
// MMC module connections
sbit Mmc_Chip_Select      at LATC2_bit;
sbit Mmc_Chip_Select_Direction at TRISC2_bit;
// End of MMC module connections

char filename[] = "TEMPERTR.TXT";
unsigned short character;
unsigned long file_size,i,rec_size;
unsigned char ch1,ch2,flag,ret_status,choice;
unsigned char temperature[10],txt[12];

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

//
// This function sends a space character to USART
//
void Space()
{
```

**Figure 7.86: mikroC Pro for PIC Program.**

```

        Uart1_Write(0x20);
    }

// This function reads the temperature from analog input AN0
//
void Get_Temperature()
{
    unsigned long Vin, Vdec,Vfrac;
    unsigned char op[12];
    unsigned char i,j;

    Vin = Adc_Read(0);                                // Read from channel 0 (AN0)
    Vin = 488*Vin;                                    // Scale up the result
    Vin = Vin /10;                                     // Convert to temperature in C
    Vdec = Vin / 100;                                  // Decimal part
    Vfrac = Vin % 100;                                 // Fractional part
    LongToStr(Vdec,op);                               // Convert Vdec to string in "op"

    //
    // Remove leading blanks
    //
    j=0;
    for(i=0;i<=11;i++)
    {
        if(op[i] != ' ')                                // If a blank
        {
            temperature[j]=op[i];
            j++;
        }
    }

    temperature[j] = '.';                            // Add "."
    ch1 = Vfrac / 10;                                // fractional part
    ch2 = Vfrac % 10;
    j++;
    temperature[j] = 48+ch1;                          // Add fractional part
    j++;
    temperature[j] = 48+ch2;
    j++;
    temperature[j] = 0x0D;                            // Add carriage-return
    j++;
    temperature[j] = 0x0A;                            // Add line-feed
    j++;
    temperature[j]='\0';
}

//

```

**Figure 7.86**  
cont'd

```
// Start of MAIN program
//
void main()
{
    unsigned char i;

    ANSELC = 0;                                // Configure PORTC as digital
    ANSELA = 1;                                // Configure RAO as analog
    TRISA = 0x1;                               // RAO (AN0) is input
//
// Configure the serial port
//
    Uart1_Init(2400);
//
// Initialise the SPI bus
//
    SPI1_Init_Advanced(_SPI_MASTER_OSC_DIV64, _SPI_DATA_SAMPLE_MIDDLE,
                       _SPI_CLK_IDLE_LOW, _SPI_LOW_2_HIGH);
//
// Initialise the SD card FAT file system
//
    while(Mmc_Fat_Init());
//
// Display the MENU and get user choice
//
while(1)
{
    Newline();
    Newline();
    Uart1_Write_Text("TEMPERATURE DATA LOGGER");      // Display heading on the PC
    Newline();
    Newline();
    Uart1_Write_Text("1. Send temperature data to the PC"); // Display opt 1 on the PC
    Newline();
    Uart1_Write_Text("2. Save temperature data in a new file"); // Display opt 2 on the PC
    Newline();
    Uart1_Write_Text("3. Append temperature data to an existing file"); // Display opt 3 on the PC
    Newline();
    Newline();
    Uart1_Write_Text("Choice ? ");                  // Get choice

//
// Read a character from the PC keyboard
//
    flag = 0;
    do {
        if (Uart1_Data_Ready() == 1)                // If data received
        {
            choice = Uart1_Read();                 // Read the received data
            Uart1_Write(choice);                  // Echo received data
            flag = 1;
        }
    }
}
```

**Figure 7.86**  
cont'd

```

    }
} while (!flag);
Newline();
Newline();
rec_size = 0;

//
// Now process user choice
//
switch(choice)
{
    case '1':
        ret_status = Mmc_Fat_Assign(&filename,1);
        if(!ret_status)
        {
            Uart1_Write_Text("File does not exist...");
            Newline();
            Uart1_Write_Text("Try again...");
        }
        else
        {
            //
            // Read the data and send to UART
            //
            Uart1_Write_Text("Sending saved data to the PC... ");
            Newline();
            Mmc_Fat_Reset(&file_size);
            for(i=0; i<file_size; i++)
            {
                Mmc_Fat_Read(&character);
                Uart1_Write(character);
            }
            Newline();
            Uart1_Write_Text("End of data... ");
        }
        break;
    case '2':
        //
        // Start the A/D converter, get temperature readings every
        // 10 s, and then save in a NEW file
        //
        Uart1_Write_Text("Saving data in a NEW file... ");
        Newline();
        Mmc_Fat_Assign(&filename,0x80);                                // Assign the file
        Mmc_Fat_Rewrite();
        Mmc_Fat_Write("TEMPERATURE DATA - SAVED EVERY 10 SECONDS\r\n",43);
        //
        // Read the temperature from A/D converter, format and save
        //
        for(i = 0; i < 100; i++)
        {
}

```

**Figure 7.86**  
cont'd

```
    Mmc_Fat_Append();
    Get_Temperature();
    Mmc_Fat_Write(temperature,9);
    rec_size++;
    LongToStr(rec_size,txt);
    Newline();
    Uart1_Write_Text("Saving record:");
    Uart1_Write_Text(txt);
    Delay_ms(10000);
}
break;
case '3':
//
// Start the A/D converter, get temperature readings every
// 10 s, and then APPEND to the existing file
//
Uart1_Write_Text("Appending data to the existing file...");
Newline();
ret_status = Mmc_Fat_Assign(&filename,1);           // Assign the file
if(!ret_status)
{
    Uart1_Write_Text("File does not exist - can not append...");
    Newline();
    Uart1_Write_Text("Try again...");
    Newline();
}
else
{
//
// Read the temperature from A/D converter, format and save
//
    for(i = 0; i < 100; i++)
    {
        Mmc_Fat_Append();
        Get_Temperature();
        Mmc_Fat_Write(temperature,9);
        rec_size++;
        LongToStr(rec_size,txt);
        Newline();
        Uart1_Write_Text("Appending new record:");
        Uart1_Write_Text(txt);
        Delay_ms(10000);
    }
}
break;
default:
    Uart1_Write_Text("Wrong choice.Try again...");
}
}
}
```

**Figure 7.86**  
cont'd

The following operations are performed inside the main program:

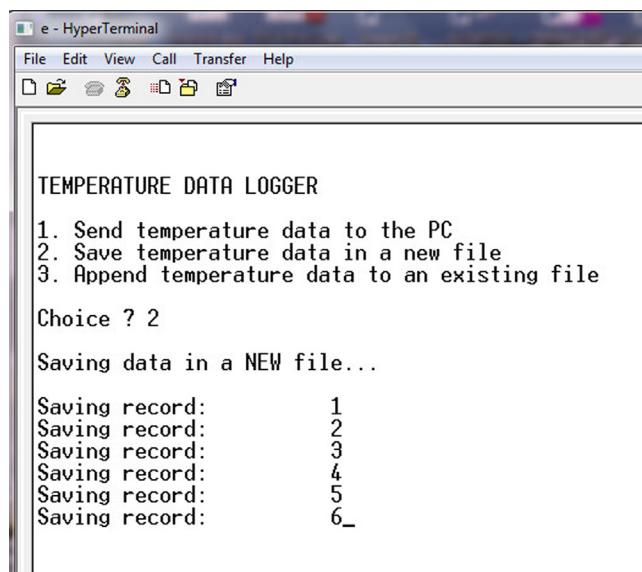
- Initialize the UART to 2400 Baud.
- Initialize the SPI bus.
- Initialize the FAT file system.
- Display menu on the PC screen.
- Get a choice from the user (between 1 and 3).
- If the choice = 1, then open the saved temperature file, read the temperature records, and send them to the PC.
- If the choice = 2, then create a new temperature file, get new temperature readings every 10 s, and store 100 records in the file.
- If the choice = 3, then assign to the temperature file, get new temperature readings every 10 s, and append them to the existing temperature file. Hundred records are appended to the file.
- If the choice is not 1–3, display an error message on the screen.

The menu options are described below in more detail:

Option 1: The program attempts to open an existing temperature file with name TEMPERRT.TXT (notice here that Mmc\_Fat\_Assign function is used. We could have used the Mmc\_Fat\_Open function instead). If the file does not exist, the error messages: “**File does not exist...**” and “**Try again...**” are displayed on the screen. If on the other hand the temperature file already exists, then the message: “**Sending saved data to the PC...**” is displayed on the PC screen. **Mmc\_Fat\_Reset** function is called to set the file pointer to the beginning of the file and also to return the size of the file in bytes. Then a **for** loop is formed, temperature records are read from the card 1 byte at a time using function **Mmc\_Fat\_Read**, and these records are sent to the PC screen. At the end of the data, the message “**End of data...**” is sent to the PC screen.

Option 2: In this option, the message: “**Saving data in a NEW file...**” is sent to the PC screen, a new file is created, with the create flag set to 0x80. The message “**TEMPERATURE DATA – SAVED EVERY 10 SECONDS**” is written on the first line of the file using function **Mmc\_Fat\_Write**. Then, a **for** loop is formed, the SD card is set into file append mode by calling function **Mmc\_Fat\_Append**, and a new temperature reading is obtained by calling function **Get\_Temperature**. The temperature is then written to the SD card. Also, the current record number is shown on the PC screen to indicate that the program is actually working. This process is repeated after a 10-s delay, until 100 records are written to the file. After this time, the main menu is displayed again.

Option 3: This option is very similar to Option 2. The only difference is that here a new file is not created, but the existing temperature file is opened in the append mode, and 100



The screenshot shows a window titled "e - HyperTerminal". The menu bar includes "File", "Edit", "View", "Call", "Transfer", and "Help". Below the menu is a toolbar with icons for copy, paste, cut, find, and others. The main window displays the following text:

```

TEMPERATURE DATA LOGGER
1. Send temperature data to the PC
2. Save temperature data in a new file
3. Append temperature data to an existing file

Choice ? 2

Saving data in a NEW file...

Saving record:      1
Saving record:      2
Saving record:      3
Saving record:      4
Saving record:      5
Saving record:      6_

```

**Figure 7.87: Saving Temperature Records on the SD Card with Option 2.**

records are written to the file. If the file does not exist, then an error message is displayed on the PC screen.

Default: If the user entry is a number outside 1–3, then this option runs and displays the error message “**Wrong choice... Try again...**” on the PC screen.

The project can be tested by connecting the output of the microcontroller to the serial port of a PC (e.g. COM1) and then running a terminal emulation software (e.g. Hyperterm or mikroC Pro for PIC built-in terminal emulator—USART Terminal). Set communication parameters to 2400 baud, 8 data bits, 1 stop bit, no parity bit, and no flow control.

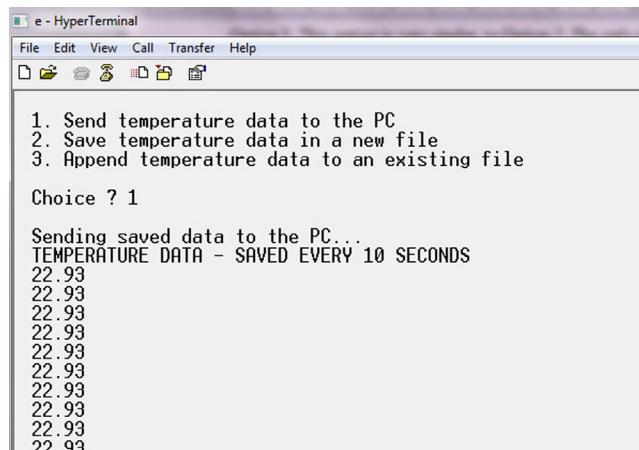
[Figure 7.87](#) shows a snapshot of the PC screen when Option 2 is selected to save temperature records in a new file. Note that the current record numbers are displayed on the screen as they are written to the SD card.

[Figure 7.88](#) shows a screen snapshot where Option 1 is selected to read the temperature records from the SD card and display them on the PC screen.

Finally, [Figure 7.89](#) shows a screen snapshot when option 3 is selected to append the temperature readings to the existing file.

#### MPLAB XC8

The MPLAB XC8 version of the program is left as an exercise to the reader.



**Figure 7.88:** Displaying the Records on the PC Screen with Option 1.

## **Project 7.10—Using Graphics LCD—Displaying Various Shapes**

Graphics LCD (GLCDs) are used in many consumer applications, such as mobile phones, MP3 players, GPS systems, games, and educational toys. Another important applications area of GLCDs is in industrial automation and control where various plant characteristics can easily be monitored or changed.

There are several GLCD screens and GLCD controllers in use currently. For small applications, the  $128 \times 64$  pixel monochrome GLCD with the KS0107/8 controller is one of the most commonly used display. For larger display requirements and more complex projects, one can select the  $240 \times 128$  pixel monochrome GLCD screen with the T6963 (or RA6963) controller. For color GLCD-based applications, thin film transistor (TFT)-type displays seem to be the best choice currently.

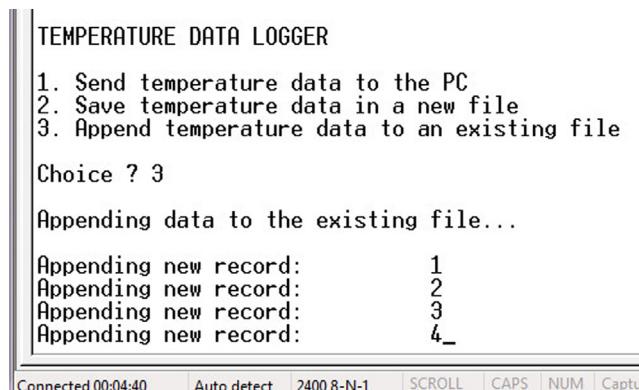


Figure 7.89: Saving Temperature Records on an SD Card with Option 3.

In this project, we shall be looking at how the standard  $128 \times 64$  GLCD can be interfaced and used in microcontroller-based projects. In this simple project, we shall see how to display various shapes on the GLCD.

### **The $128 \times 64$ Pixel GLCD**

These GLCDs have dimensions of  $7.8 \times 7.0$  cm and a thickness of 1.0 cm. The viewing area is  $6.2 \times 4.4$  cm. The display consists of  $128 \times 64$  pixels, organized as 128 pixels in the horizontal direction and 64 pixels in the vertical direction. The display operates with a +5-V supply, consumes typically 8-mA current, and comes with a built-in KS0108-type display controller. A backlight LED is provided for visibility in low ambient light conditions. This LED consumes about 360 mA when operated. Basically two controllers are used internally: one for segments 1–64, and the other one for segments 65–128.

The display is connected to the external world through a 20-pin SIL (Single-In-Line) type connector. [Table 7.16](#) gives the pin numbers and corresponding pin names.

The description of each pin is as follows:

**/CSA, /CSB:** Chip select pins for the two controllers. The display is logically divided into two sections, and these signals control which half should be enabled at any time.

**Table 7.16:  $128 \times 64$  Pixel GLCD Pin Configuration.**

Pin No	Pin Name	Function
1	\CSA or CS1	Chip select for controller 1
2	\CSB or CS2	Chip select for controller 2
3	V <sub>SS</sub>	Ground
4	V <sub>DD</sub>	+5 V
5	V <sub>O</sub>	Contrast adjustment
6	D/I	Register select
7	R/W	Read-write
8	E	Enable
9	DB0	Data bus bit 0
10	DB1	Data bus bit 1
11	DB2	Data bus bit 2
12	DB3	Data bus bit 3
13	DB4	Data bus bit 4
14	DB5	Data bus bit 5
15	DB6	Data bus bit 6
16	DB7	Data bus bit 7
17	RST	Reset
18	V <sub>EE</sub>	Negative voltage
19	A	LED +4.2 V
20	K	LED ground

**V<sub>CC</sub>, GND:** Power supply and ground pins.

**V<sub>0</sub>:** Contrast adjustment. A 10 K potentiometer should be used to adjust the contrast. The wiper arm should be connected to this pin, and the other two arms should be connected to V<sub>EE</sub> and the ground.

**D/I:** Register select pin. Logic HIGH is data mode, logic LOW is instruction mode.

**R/W:** Read–write pin. Logic HIGH is read, logic LOW is write.

**E:** Enable pin. Logic HIGH to LOW to enable.

**DB0–DB7:** Data bus pins.

**RST:** Reset pin. The display is reset if this pin is held LOW for at least 100 ns. During reset, the display is off, and no commands can be executed by the display controller.

**V<sub>EE</sub>:** Negative voltage output pin for contrast adjustment.

**A, K:** Power supply and ground pins for the backlight. Pin K should be connected to the ground and pin A should be connected to a +5-V supply through a 10- $\Omega$  resistor.

Figure 7.90 shows the connection of the GLCD to a microcontroller with the contrast adjustment potentiometer and backlight LED connections shown as well.

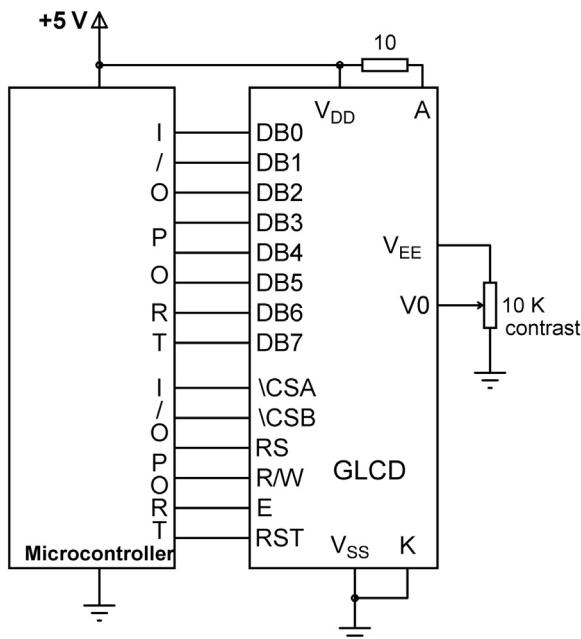


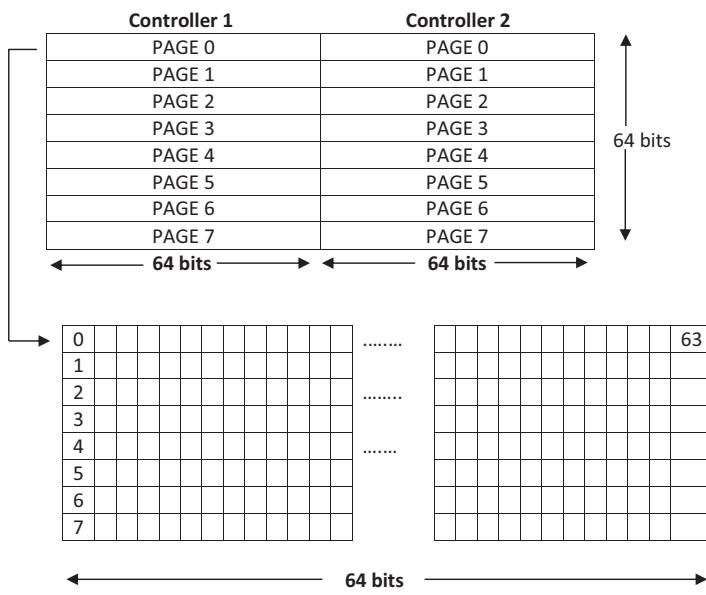
Figure 7.90: Connecting the 128 × 64 GLCD to a Microcontroller.

### **Operation of the GLCD**

The internal operation of the GLCD and the KS0108 controller is very complex and beyond the scope of this book. Most high-level microcontroller compiler developers provide libraries for using these displays in their programming languages. In this project, only the basic information required before using the GLCD library is given.

[Figure 7.91](#) shows the structure of the GLCD as far as programming the display is concerned. The  $128 \times 64$  pixel display is logically split into two halves. There are two controllers: controller A controlling the left half of the display and controller B controlling the right half, where the two controllers are addressed independently. Each half of the display consists of 8 pages where each page is 8 bits high and 8 bytes (64 bits) wide. Thus, each half consists of  $64 \times 64$  bits. Text is written to the pages of the display. Thus, a total of 16 characters across can be written for a given page on both halves of the display. Considering that there are 8 pages, a total of 128 characters can be written on the display.

The origin of the display is the top left hand corner ([Figure 7.92](#)). The X-direction extends toward the right, and Y-direction extends toward the bottom of the display. In the X-direction, the pixels range from 0 to 127, while in the Y-direction the pixels range from 0 to 63. Coordinate (127, 63) is at the bottom right-hand corner of the display.



**Figure 7.91: Structure of the GLCD.**

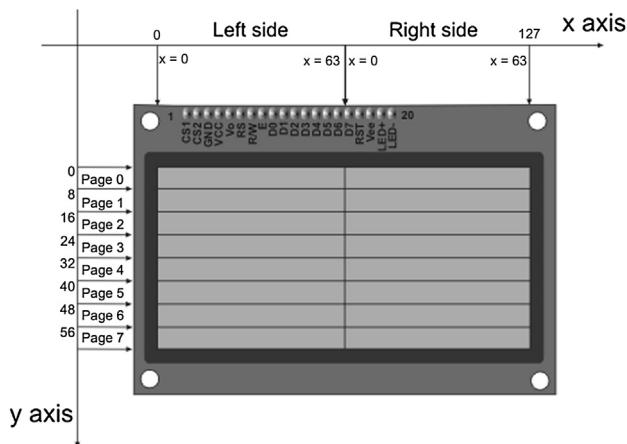


Figure 7.92: GLCD Coordinates.

### *mikroC Pro for PIC GLCD Library Functions*

mikroC Pro for PIC language supports the  $128 \times 64$  pixel GLCDs and provides a large library of functions for the development of GLCD-based projects. In actual fact there are libraries for several different types of GLCDs. In this section, we shall be looking at the commonly used library functions provided for the  $128 \times 64$  GLCDs, working with the KS0108 controller.

#### *Glcd\_Init*

This function initializes the GLCD module. The GLCD control and data lines can be configured by the user, but the eight data lines must be on a single port. Before this function is called, the interface between the GLCD and the microcontroller must be defined using *sbit* type statements of the following format. In the following example, it is assumed that the GLCD data lines are connected to PORTD, and in addition, the CS1, CS2, RS, RW, EN, and RST lines are connected to PORTB:

```
// GLCD pinout settings
char GLCD_DataPort at PORTD;

sbit GLCD_CS1 at RB0_bit;
sbit GLCD_CS2 at RB1_bit;
sbit GLCD_RS at RB2_bit;
sbit GLCD_RW at RB3_bit;
sbit GLCD_EN at RB4_bit;
sbit GLCD_RST at RB5_bit;

sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
```

```
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;

Example Call: Glcd_Init();
```

### ***Glcd\_Set\_Side***

This function selects the GLCD side based on the argument, which is the x coordinate. Values from 0 to 63 specify the left side of the display, while values from 64 to 127 specify the right side.

```
Example Call: Glcd_Set_Side(0); // Select left hand side of display
```

### ***Glcd\_Set\_X***

This function sets the x-axis position from the left border of the GLCD within the selected display side.

```
Example Call: Glcd_Set_X(10); // Set position to pixel 10
```

### ***Glcd\_Set\_Page***

This function selects a page of the GLCD. The argument to the function is the page number between 0 and 7.

```
Example Call: Glcd_Set_Page(2); // Select Page 2
```

### ***Glcd\_Write\_Data***

This function writes 1 byte to the current location on the GLCD memory and moves to the next location. The GLCD side and page number should be set before calling this function.

```
Example Call: Glcd_Write_Data(MyData);
```

### ***Glcd\_Fill***

This function fills the GLCD memory with the specified byte pattern, where the pattern is passed as an argument to the function.

```
Example Call: Glcd_Fill(0); // Clears the screen
```

### ***Glcd\_Dot***

This function draws a dot on the GLCD at coordinates x\_pos, y\_pos. The x and y coordinates and the color of the dot are passed as arguments. Valid x coordinates are

0–127, valid y coordinates are 0–63, and valid colors are 0–2, where 0 clears the dot, 1 places a dot, and 2 inverts the dot.

Example Call: `Glcd_Dot(0, 10, 1); // Place a dot at x = 0, y = 10`

### ***Glcd\_Line***

This function draws a line on the GLCD. The arguments passed to the function are

<code>x_start:</code>	x coordinate of the line starting position (0 to 127)
<code>y_start:</code>	y coordinate of the line starting position (0 to 63)
<code>x_end:</code>	x coordinate of the line ending position (0 to 127)
<code>y_end:</code>	y coordinate of the line ending position (0 to 63)
<code>colour:</code>	The colour value between 0 and 2. 0 is white, 1 is black, and 2 inverts each dot.

Example Call: `Glcd_Line(0, 0, 5, 10, 1); // Draw a line from (0,0) to (5,10)`

### ***Glcd\_V\_Line***

This function draws a vertical line on the GLCD. The arguments passed to the function are

<code>y_start:</code>	y coordinate of the line starting position (0 to 63)
<code>y_end:</code>	y coordinate of the line ending position (0 to 63)
<code>x_pos:</code>	x coordinate of the vertical line (0 to 127)
<code>colour:</code>	The colour value between 0 and 2. 0 is white, 1 is black, and 2 inverts each dot.

Example Call: `Glcd_V_Line(4, 10, 5, 1); // Draw a line from (5,4) to (5,10)`

### ***Glcd\_H\_Line***

This function draws a horizontal line on the GLCD. The arguments passed to the function are

<code>x_start:</code>	x coordinate of the line starting position (0 to 127)
<code>x_end:</code>	x coordinate of the line ending position (0 to 127)
<code>y_pos:</code>	y coordinate of the vertical line (0 to 63)
<code>colour:</code>	The colour value between 0 and 2. 0 is white, 1 is black, and 2 inverts each dot.

Example Call: `Glcd_H_Line(15, 55, 25, 1); // Draw a line from (15,25) to (55,25)`

### ***Glcd\_Rectangle***

This function draws a rectangle on the GLCD. The arguments passed to the function are

<code>x_upper_left:</code>	x coordinate of the upper left corner of rectangle (0 to 127)
<code>y_upper_left:</code>	y coordinate of the upper left corner of rectangle (0 to 63)

```
x_bottom_right: x coordinate of the lower right corner of rectangle (0 to 127)
y_bottom_right: y coordinate of the lower right corner of rectangle (0 to 63)
colour: The colour value between 0 and 2. 0 is white, 1 is black, and 2
        inverts each dot.
Example Call: Glcd_Rectangle(5, 5, 10, 10); // Draw rectangle between (5,5)
              and (10,10)
```

### ***Glcd\_Rectangle\_Round\_Edges***

This function draws a rounded-edge rectangle on the GLCD. The arguments passed to the function are

```
x_upper_left: x coordinate of the upper left corner of rectangle (0 to 127)
y_upper_left: y coordinate of the upper left corner of rectangle (0 to 63)
x_bottom_right: x coordinate of the lower right corner of rectangle (0 to 127)
y_bottom_right: y coordinate of the lower right corner of rectangle (0 to 63)
round radius: radius of the rounded edge
colour: The colour value between 0 and 2. 0 is white, 1 is black, and 2
        inverts each dot.
Example Call: Glcd_Rectangle_Round_Edge(5, 5, 10, 10, 15, 1);
              // Draw rectangle between (5,5) and (10,10) with edge radius 15
```

### ***Glcd\_Rectangle\_Round\_Edges\_Fill***

This function draws a filled rounded edge rectangle on the GLCD with color. The arguments passed to the function are

```
x_upper_left: x coordinate of the upper left corner of rectangle (0 to 127)
y_upper_left: y coordinate of the upper left corner of rectangle (0 to 63)
x_bottom_right: x coordinate of the lower right corner of rectangle (0 to 127)
y_bottom_right: y coordinate of the lower right corner of rectangle (0 to 63)
round radius: radius of the rounded edge
colour: colour of the rectangle border. The colour value is between 0 and
        2. 0 is white, 1 is black, and 2 inverts each dot.
Example Call: Glcd_Rectangle_Round_Edges_Fill(5, 5, 10, 10, 15, 1);
              // Draw rectangle between (5,5) and (10,10) with edge radius 15
```

### ***Glcd\_Box***

This function draws a box on the GLCD. The arguments passed to the function are:

```
x_upper_left: x coordinate of the upper left corner of box (0 to 127)
y_upper_left: y coordinate of the upper left corner of box (0 to 63)
x_bottom_right: x coordinate of the lower right corner of box (0 to 127)
y_bottom_right: y coordinate of the lower right corner of box (0 to 63)
colour: colour of the box fill. The colour value is between 0 and 2. 0 is
        white, 1 is black, 2 inverts each dot.
Example Call: Glcd_Box(5, 15, 20, 30, 1); // Draw box between (5,15) and (20,30)
```

### ***Glcd\_Circle***

This function draws a circle on the GLCD. The arguments passed to the function are as follows:

```
x_center:      x coordinate of the circle center (0 to 127)
y_center:      y coordinate of the circle center (0 to 63)
radius:        radius of the circle
colour:        colour of the circle line. The colour value is between 0 and 2.
               0 is white, 1 is black, 2 inverts each dot.
Example Call:  Glcd_Circle(30, 30, 5, 1);
               // Draw circle with center at (30,30), and radius 5
```

### ***Glcd\_Circle\_Fill***

This function draws a filled circle on the GLCD. The arguments passed to the function are as follows:

```
x_center:      x coordinate of the circle center (0 to 127)
y_center:      y coordinate of the circle center (0 to 63)
radius:        radius of the circle
colour:        The colour value is between 0 and 2. 0 is white, 1 is black, 2
               inverts each dot.
Example Call:  Glcd_Circle_Fill(30, 30, 5, 1);
               // Draw a filled circle with center at (30,30), and radius 5
```

### ***Glcd\_Set\_Font***

This function sets the font that will be used with functions: Glcd\_Write\_Char and Glcd\_Write\_Text. The arguments passed to the function are

```
activeFont:    font to be set. Needs to be formatted as an array of char
aFontWidth:   width of the font characters in dots.
aFontHeight:  height of the font characters in dots.
aFontOffs:    number that represents difference between the mikroC Pro for PIC
               character set and regular ASCII set (e.g. if A is 65 in ASCII
               character, and A is 45 in the mikroC Pro for the PIC character set,
               aFontOffs is 20)
```

List of supported fonts are as follows:

- Font\_Glcd\_System3x5,
- Font\_Glcd\_System5x7,
- Font\_Glcd\_5x7,
- Font\_Glcd\_Character8x7.

```
Example Call:  Glcd_Set_Font(&MyFont, 5, 7, 32);
               //Use custom 5x7 font MyFont which starts with space character (32)
```

### ***Glcd\_Set\_Font\_Adv***

This function sets the font that will be used with functions: Glcd\_Write\_Char\_Adv and Glcd\_Write\_Text\_Adv. The arguments passed to the function are

```
activeFont:          font to be set. Needs to be formatted as an array of char.  
font_colour:        sets font colour.  
font_orientation:   sets font orientation.  
Example Call:       Glcd_Set_Font_Adv(&MyFont, 0, 0);
```

### ***Glcd\_Write\_Char***

This function displays a character on the GLCD. if no font is specified, then the default Font\_Glcd\_System5x7 font supplied with the library will be used. The arguments passed to the function are

```
chr:                character to be displayed  
x_pos:              character starting position on x-axis (0 to 127- FontWidth)  
page_num:           the number of the page on which the character will be displayed (0 to 7)  
colour:             colour of the character between 0 and 2. 0 is white, 1 is black, 2  
                   inverts each dot  
Example Call:       Glcd_Write_Char('Z', 10, 2, 1);  
                   //Display character Z at x position 10, inside page 2
```

### ***Glcd\_Write\_Char\_Adv***

This function displays a character on the GLCD at coordinates (x, y).

```
ch:                character to be displayed.  
x:                 character position on x-axis.  
y:                 character position on y-axis.  
Example Call:      Glcd_Write_Char_Adv('A', 20, 10,); // Display A at (20,10)
```

### ***Glcd\_Write\_Text***

This function displays text on the GLCD. if no font is specified, then the default Font\_Glcd\_System5x7 font supplied with the library will be used. The arguments passed to the function are

```
text:               text to be displayed  
x_pos:              text starting position on x-axis.  
page_num:            the number of the page on which text will be displayed (0 to 7)  
colour:              The colour parameter between 0 and 2. 0 is white, 1 is black and 2  
                   inverts each dot.  
Example Call:       Glcd_Write_Text("My Computer", 10, 3, 1);  
                   //Display "My Computer" at x position 10 in page 3
```

### ***Glcd\_Write\_Text\_Adv***

This function displays text on the GLCD at coordinates (x, y). The arguments passed to the function are as follows:

```
text:          text to be displayed
x:            text position on x-axis.
y:            text position on y-axis.
Example Call: Glcd_Write_Text_Adv("My Computer", 10, 10);
               //Display text "My Computer" at coordinates (10,10)
```

### ***Glcd\_Write\_Const\_Text\_Adv***

This function displays text on the GLCD, where the text is assumed to be located in the program memory of the microcontroller. The text is displayed at coordinates (x, y). The arguments passed to the function are

```
text:          text to be displayed
x:            text position on x-axis.
y:            text position on y-axis.
const char Txt[ ] = "My Computer";
Example Call: Glcd_Write_Text_Adv(Txt, 10, 10);
               //Display text "My Computer" at coordinates (10,10)
```

### ***Glcd\_Image***

This function displays the bitmap image on the GLCD. The image to be displayed is passed as an argument to the function. The bitmap image array must be located in the program memory of the microcontroller. The GLCD Bitmap Editor of mikroC Pro for PIC compiler can be used to convert an image to a constant so that it can be displayed by this function.

```
Example Call: Glcd_Image(MyImage);
```

### ***Project Hardware***

The circuit diagram of the project is shown in [Figure 7.93](#). The GLCD is connected to PORTB and PORTD of the microcontroller. The microcontroller is operated from an 8-MHz crystal.

### ***Project Program***

#### ***mikroC Pro for PIC***

The mikroC Pro for PIC program listing is given in [Figure 7.94](#) (MIKROC-GLCD1.C). At the beginning of the program, the GLCD connections are

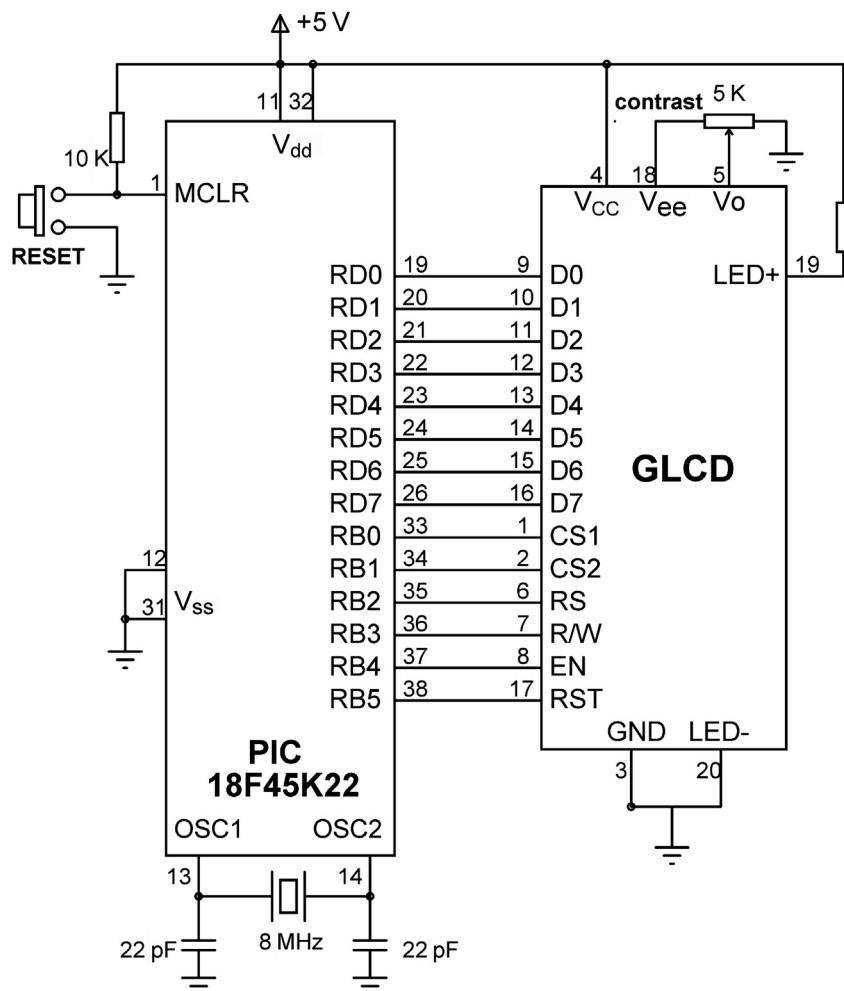


Figure 7.93: Circuit Diagram of the Project.

defined, and PORTB and PORTD are configured as digital. Then, the following shapes are drawn on the GLCD:

- A rectangle with rounded edges at coordinates (5, 5), (123, 59) and edge radius 10;
- A rectangle at coordinates (15, 15), (113, 49);
- A line from (50, 30) to (70, 30);
- A circle with center at (30, 30) and radius 10;
- A filled circle with the center at (50, 42) and radius 5;
- Text “Txt” at the x coordinate 80 and page 3;
- Text “LCD” at the x coordinate 80 and page 4;
- Text “micro” at coordinates (80, 38).

```
*****
GLCD LIBRARY EXAMPLE
-----
This program uses some of the mikroC GLCD library functions to show how the functions should
be used in programs.

The program was loaded to a PIC18F45K22 microcontroller and operated with a 8 MHz crystal.
The EasyPIC 7 development board is used for this project

Author: Dogan Ibrahim
Date: October, 2013
File: MIKROC-GLCD1.C
*****/
```

```
// Glcd module connections
char GLCD_DataPort at PORTD;

sbit GLCD_CS1 at RB0_bit;
sbit GLCD_CS2 at RB1_bit;
sbit GLCD_RS at RB2_bit;
sbit GLCD_RW at RB3_bit;
sbit GLCD_EN at RB4_bit;
sbit GLCD_RST at RB5_bit;

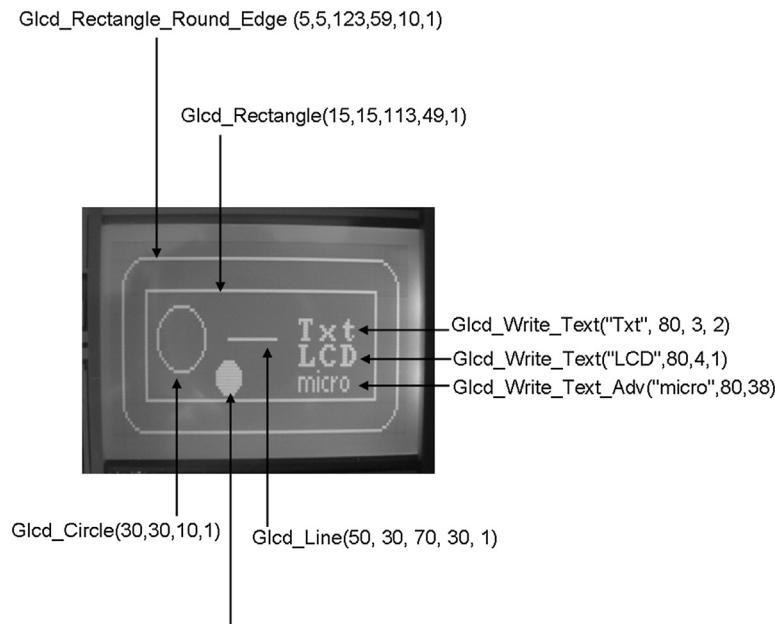
sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;
// End Glcd module connections

void main()
{
    ANSELB = 0;                                // Configure PORTB as digital
    ANSELD = 0;                                // Configure PORTD as digital

    Glcd_Init();                               // Initialize GLCD
    Glcd_Fill(0x00);                           // Clear GLCD
    Glcd_rectangle_round_edges(5,5,123,59,10,1); // Draw rectangle
    Glcd_Rectangle(15,15,113,49,1);            // Draw rectangle
    Glcd_Line(50, 30, 70, 30, 1);              // Draw line
    Glcd_Circle(30,30,10,1);                  // Draw circle
    Glcd_Circle_Fill(50,42,5,1);               // Draw filled circle
    Glcd_Set_Font(Font_Glcd_Character8x7, 8, 7, 32); // Change Font

    Glcd_Write_Text("Txt", 80, 3, 2);          // Write string "Txt"
    Glcd_Write_Text("LCD",80,4,1);             // Write string "LCD"
    Glcd_Write_Text_Adv("micro",80,38);        // Write string "micro"
}
```

**Figure 7.94: mikroC Pro for PIC Program.**



**Figure 7.95: Shapes Drawn on the GLCD.**

Figure 7.95 shows the shapes drawn on the GLCD.

### **Project 7.11—Barometer, Thermometer and Altimeter Display on a GLCD**

This project is about using a sensor to read and display the pressure, temperature, and the altitude on a GLCD.

The project is based on using a MEMS sensor called LPS331AP. This is basically a pressure sensor, but it can also measure the ambient temperature. The altitude is calculated mathematically from the pressure measurements.

It is necessary to know the features and basic operation of this sensor before it can be used in projects. The features of this sensor are as follows:

- A 260–1260 mbar absolute pressure measurement;
- A 0 to +80 °C temperature measurement;
- Very low-power consumption (30 µA in the high-resolution mode);
- A 24-bit digital pressure output in millibars;
- A 16-bit digital temperature output in degrees Celsius;
- SPI and I<sup>2</sup>C interfaces;
- A +1.71- to +3.6-V supply voltage.

The LPS331AP is a 16-pin device, having dimensions of  $3 \times 3 \times 1$  mm. In this project, it is operated in the I<sup>2</sup>C communications mode. When operated in this mode the following pins are used:

Pin	Description
1	VDD_IO (power supply)
4	SCL (I <sup>2</sup> C clock)
5	GND
6	SDA (I <sup>2</sup> C data)
7	SA0 (I <sup>2</sup> C device address LSB)
8	CS (Set to 1 for I <sup>2</sup> C mode)
9	INT2 (interrupt or data ready)
11	INT1 (interrupt or data ready)
12	GND
13	GND
14	VDD (power supply)
15	VCCA (analog power supply)
16	GND

When pin 7 is connected to the supply voltage, the device write and read addresses are 0xBA and 0xBB, respectively. Alternatively, when connected to the ground, the device write and read addresses are 0xB8 and 0xB9, respectively.

Figure 7.96 shows the output block diagram of the LPS331AP sensor. The device is controlled with 19 registers (Table 7.17). A 24-bit reference pressure output (registers REF\_P\_XL, REF\_P\_L, and REF\_P\_H) is subtracted from the measure sensor pressure to obtain the 24-bit output pressure via registers PRESS\_POUT\_XL\_REG, PRESS\_OUT\_L, and PRESS\_OUT\_H. The measured pressure is compared with two threshold pressures preloaded into registers THS\_P\_LOW\_REG and THS\_P\_HIGH\_REG. If the measured pressure is higher than THS\_P\_HIGH\_REG, then a High Pressure Interrupt (PH) is generated. Similarly, if the measured pressure is lower than THS\_P\_LOW\_REG, then a Low Pressure Interrupt (PL) is generated.

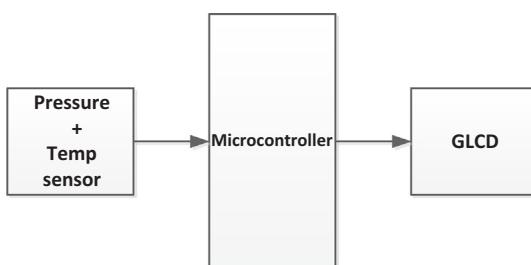


Figure 7.96: Block Diagram of the Project.

**Table 7.17: LPS331AP Registers.**

Name	Type	Address (Hex)
REF_P_XL	R/W	08
REF_P_L	R/W	09
REF_P_H	R/W	0A
WHO_AM_I	R	0F
RES_CONF	R/W	10
CTRL_REG1	R/W	20
CTRL_REG2	R/W	21
CTRL_REG3	R/W	22
INT_CFG_REG	R/W	23
INT_SOURCE_REG	R	24
THS_P_LOW_REG	R/W	25
THS_P_HIGH_REG	R/W	26
STATUS_REG	R	27
PRESS_POUT_XL_REG	R	28
PRESS_OUT_L	R	29
PRESS_OUT_H	R	2A
TEMP_OUT_L	R	2B
TEMP_OUT_H	R	2C
AMP_CTRL	R/W	30

The functions of some important registers are described below:

**WHO\_AM\_I:** This register is used to identify the device and returns 0xBB.

**RES\_CONF:** This register is used to select the internal pressure and temperature averaging to be used in a measurement. Loading the recommended value 0x78 configures for 256 averages for the pressure and 128 averages for the temperature measurements.

**CTRL\_REG1:** This register controls the active/power-down mode (bit 7), output data rates for pressure and temperature (bits 4–6), and output update control bit (bit 2). Loading 0x04 configures the device to enter power-down mode, one-shot pressure and temperature measurement, that is, a request must be done for a measurement.

**CTRL\_REG2:** Bit 0 of this register controls the one-shot action. When the bit is set to 1, a new measurement starts. The bit is cleared at the end of the measurement.

**STATUS\_REG:** This register can be used to check if new pressure or temperature data is available. Bit 0 (T\_DA) is set to 1 if new temperature data are available. Similarly, bit 1 (P\_DA) is set to 1 when new pressure data are available.

**PRESS\_OUTxxx:** 24-bit pressure output registers.

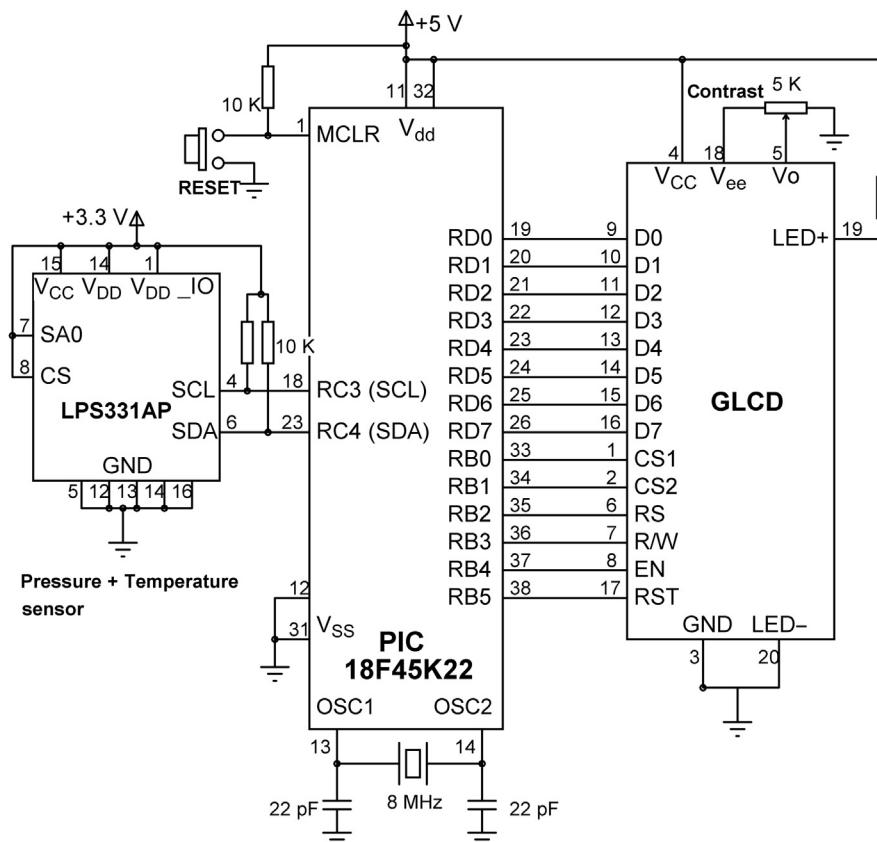
**TEMP\_OUTxxx:** 16-bit temperature output registers.

The block diagram of the project is shown in [Figure 7.96](#). The pressure is displayed in integer format while the other two are displayed in fractional format:

```
P (mb) : nnnn
T(C)   : nn.n
A(ft)   : nnn.n
```

### Project Hardware

The circuit diagram of the project is shown in [Figure 7.97](#). The sensor is connected to the I<sup>2</sup>C pins of the microcontroller via pull-up resistors. The SA0 pin is connected to supply voltage so that the device write and read addresses are 0xBA and 0xBB, respectively. The CS pin is connected to the supply voltage to select I<sup>2</sup>C communication protocol. The GLCD is connected to PORTB and PORTD pins in the default configuration.



**Figure 7.97: Circuit Diagram of the Project.**

An 8-MHz crystal is used for timing, but the clock PLL is enabled so that the actual running clock frequency is 32 MHz.

### **Project PDL**

The PDL of the project is shown in [Figure 7.98](#).

### **Project Program**

#### *mikroC Pro for PIC*

The mikroC Pro for PIC program is shown in [Figure 7.99](#) (MIKRO-GLCD2.C). At the beginning of the program, the GLCD connections to the microcontroller are defined. Also, the register names and addresses of the LPS331AP chip are defined. Inside the main program, PORTB, PORTC, and PORTD are configured as digital. The GLCD module is initialized and the screen is cleared, the I<sup>2</sup>C module is initialized (with 100 kHz clock rate).

The program then initializes the LPS331AP chip with full resolution and One-shot operation mode. The rest of the program is executed in an endless loop. Here, the pressure and temperature are read from the sensor, the altitude is calculated, and all three parameters displayed on the GLCD screen. This process is repeated every 5 s.

According to the LPS331AP data sheet, the pressure must be divided by 4096 (or shifted right by 12 bits) to convert it to millibars. The temperature must be divided by 480 and then 42.5 added to obtain the readings in degrees centigrade. The altitude calculation is done based on a formula given in the LPS331AP application note. This formula depends only on the pressure reading and is very approximate, given by

$$\text{Altitude (feet)} = \left[ 1 - \left( \frac{\text{Pressure}}{1013.25} \right)^{0.190284} \right] \times 145,366.45.$$

The following functions are used in the project:

Pressure\_Write: Write a byte to the sensor chip.

Pressure\_Read: Read a byte from the sensor chip.

Init\_Pressure: Initialize the sensor chip.

Read\_Pressure\_Value: Send a One-shot signal to the sensor chip to start measurement. Read the pressure when it is ready, convert to millibars, and return the value to the calling program.

Read\_Temperature\_Value: Read the temperature when it is ready, convert into degrees Centigrade, and return to the calling program.

**Main Program**

```

BEGIN
    Define connections between LCD and microcontroller
    Define LPS331AP register addresses
    Configure PORTB, PORTC, PORTD as digital
    Initialize GLCD
    Initialize I2C module
    CALL Init_Pressure_Chip
DO FOREVER
    CALL Read_Pressure_Value
    CALL Read_Temperature_Value
    CALL Read_Altimeter_Value
    Display pressure, temperature, and altitude
    Wait 5 seconds
    Clear GLCD
ENDDO
END

BEGIN/Pressure_Write
    Write byte to sensor chip
END/Pressure_Write

BEGIN/Pressure_Read
    Read byte from sensor chip
    Return the byte to calling program
END/Pressure_Read

BEGIN/Init_Pressure_Chip
    Configure sensor resolution
    Configure sensor to one-shot mode
    Check chip identity
END/Init_Pressure_Chip

BEGIN/Read_Pressure_Value
    Start conversion
    Wait until reading is available
    Get a pressure reading
    Convert into millibars
    Return the pressure to calling program
END/Read_Pressure_Value

BEGIN/Read_Temperature_Value
    Wait until temperature is available
    Read temperature
    Convert into degrees Centigrade
    Return temperature to calling program
END/Read_Temperature_Value

BEGIN/Read_Altimeter_Value
    Convert pressure into altitude
    Return altitude to calling program
END/Read_Altimeter_Value

BEGIN/Display_PTA
    Display pressure
    Display temperature
    Display altitude
END/Display_PTA

```

**Figure 7.98: Project PDL.**

```
*****
BAROMETER, THERMOMETER AND ALTIMETER DISPLAY
-----
This program uses the LPS331AP MEMS pressure sensor chip. In addition to pressure, the chip
also measures the temperature, and the altimeter reading can be obtained
from the pressure reading

The program displays the pressure, altitude, and temperature. The program works in One-shot
mode. i.e. a new sample is requested and the sample is received and displayed. Then a new
sample is requested and so on.

The clock PLL is enabled so that the actual clock frequency is 32 MHz (Enable the 4xPLL and set
oscillator frequency to 32 MHz in Project-> Edit Project)

Author: Dogan Ibrahim
Date: October, 2013
File: MIKROC-GLCD2.C
*****/
```

```
// Glcd module connections
char GLCD_DataPort at PORTD;

sbit GLCD_CS1 at LATB0_bit;
sbit GLCD_CS2 at LATB1_bit;
sbit GLCD_RS at LATB2_bit;
sbit GLCD_RW at LATB3_bit;
sbit GLCD_EN at LATB4_bit;
sbit GLCD_RST at LATB5_bit;

sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;
// End Glcd module connections

// Define LPS331AP registers
#define REF_P_XL 0x08          // Reference pressure (LSB)
#define REF_P_L 0x09            // Reference pressure (middle)
#define REF_P_H 0x0A            // Reference pressure (MSB)
#define WHO_AM_I 0x0F           // Device identification
#define RES_CONF 0x10           // Pressure resolution
#define CTRL_REG1 0x20          // Control register 1
#define CTRL_REG2 0x21          // Control register 2
#define CTRL_REG3 0x22          // Control register 3
#define INT_CFG_REG 0x23         // Interrupt configuration register
#define INT_SOURCE_REG 0x24       // Interrupt source register
#define THS_P_LOW_REG 0x25        // Threshold LOW register
#define THS_P_HIGH_REG 0x26       // Threshold HIGH register
```

Figure 7.99: mikroC Pro for PIC Program.

```

#define STATUS_REG 0x27          // Status register
#define PRESS_POUT_XL_REH 0x28   // Pressure output register (LSB)
#define PRESS_OUT_L 0x29         // Pressure output register (middle)
#define PRESS_OUT_H 0x2A         // Pressure output register (MSB)
#define TEMP_OUT_L 0x2B          // Temperature output register (LSB)
#define TEMP_OUT_H 0x2C          // Temperature output register (MSB)
#define AMP_CTRL 0x30            // Analog front end control register
#define Write_Addr 0xBA          // Device write register
#define Read_Addr 0xBB            // Device read register

void Pressure_Write(unsigned char address, unsigned char value)
{
    I2C1_Start();                // Send START bit
    I2C1_Wr(Write_Addr);         // Send device address
    I2C1_Wr(address);           // Send register address
    I2C1_Wr(value);             // Send data
    I2C1_Stop();                 // Send STOP bit
}

unsigned char Pressure_Read(unsigned char address)
{
    unsigned char c = 0;

    I2C1_Start();                // Send START bit
    I2C1_Wr(Write_Addr);         // Send device address
    I2C1_Wr(address);           // Send register address
    I2C1_Repeated_Start();       // Send repeated START
    I2C1_Wr(Read_Addr);         // Send device read address
    c = I2C1_Rd(0);              // Read, send no ack
    I2C1_Stop();                 // Send STOP bit
    return c;
}
//
// This function initializes the pressure chip
//
unsigned char Init_Pressure_Chip()
{
    unsigned char temp, flag = 0;

    Pressure_Write(RES_CONF, 0x78); // Select pressure and temp resolution
    Pressure_Write(CTRL_REG1, 0x04); // Configure One-shot mode
    Pressure_Write(CTRL_REG1, 0x84); // Configure chip active mode
    temp = Pressure_Read(WHO_AM_I); // Read chip identity
    if(temp != 0xBB)flag = 1;      // Error if wrong chip identity
    return flag;
}

//
// This function reads and returns the pressure (24 bits). The reading is converted into
// millibars after dividing by 4019 (shifting right 12 bits)
//

```

**Figure 7.99**

cont'd

```

//  

long int Read_Pressure_Value()  

{  

    long int outP;  

    unsigned char stat, P_DA, PressureM, PressureL;  

    Pressure_Write(CTRL_REG2, 1); // Send One-shot request  

//  

// Check if new pressure data is available, if so read it  

//  

    P_DA = 0;  

    while(P_DA == 0) // Wait until new pressure is available  

    {  

        stat = Pressure_Read(STATUS_REG); // Read the status register  

        P_DA = stat & 0x02; // Extract P_DA  

    }  

    OutP = Pressure_Read(PRESS_OUT_H); // Read high byte  

    PressureM = Pressure_Read(PRESS_OUT_L); // Read middle byte  

    PressureL = Pressure_Read(PRESS_POUT_XL_REH); // Read low byte  

    OutP = (OutP << 8); // Move to middle byte position  

    OutP = OutP | PressureM; // Add middle byte  

    OutP = (OutP << 8); // Move to upper byte position  

    OutP = (OutP | PressureL); // Add low byte  

    OutP = (OutP >> 12); // Divide by 4096 (in mbars)  

    return OutP; // Return the pressure
}  

//  

// This function reads and returns the temperature (16 bits). The reading is converted into  

// Degrees Centigrade after the following operation:  

//  

// (Value / 480) + 42.5  

//  

//  

float Read_Temperature_Value()  

{  

    int OutT;  

    unsigned char stat, T_DA, TempL;  

    float DegreesC;  

//  

// Wait until a new temperature data is available and if so get it  

//  

    T_DA = 0;  

    while(T_DA == 0) // Wait for new temperature  

    {  

        stat = Pressure_Read(STATUS_REG); // Read status register  

        T_DA = stat & 0x01; // extract T_DA bit
    }
}

```

**Figure 7.99**  
cont'd

```

OutT = Pressure_Read(TEMP_OUT_H);           // Read high byte
TempL = Pressure_Read(TEMP_OUT_L);          // Read low byte

OutT = OutT << 8;                         // Move to left byte
OutT = OutT | TempL;                        // Add low byte
DegreesC = ((OutT / 480.0) + 42.5);         // Return the temperature
return DegreesC;
}

// This function calculates the height (altimeter function) from the pressure
//
float Read_Altimeter_Value(long int Pressure)
{
    float Altitude_ft;

    Altitude_ft = Pressure/1013.25;
    Altitude_ft = pow(Altitude_ft, 0.190284);
    Altitude_ft = (1.0 - Altitude_ft)*145366.45;

    return Altitude_ft;
}

// This function displays the pressure, temperature and altitude on the GLCD
// in the following format:
//
// P(mb): nnnn
// T(C) : nn.n
// A(ft): nn.n
//
void Display_PTA(long int Pressure, float Temperature, float Altitude)
{
    unsigned char i, Txt[14];
    char *res;

    Glcd_Rectangle(5,5,120,55,1);           // Draw rectangle
    Glcd_Write_Text("P(mb): ", 7,1,1);       // Write string "Pressure (mb): "
    Glcd_Write_Text("T(C) : ", 7,3,1);
    Glcd_Write_Text("A(ft): ", 7,5,1);
    // Display Pressure
    for(i=0; i<14; i++)Txt[i] = 0;
    LongWordToStr(Pressure, Txt);
    Ltrim(Txt);
    Glcd_Write_Text(Txt, 45,1,1);
    // Display temperature
    for(i=0; i<14; i++)Txt[i] = 0;
    FloatToStr(Temperature, Txt);
    res = strrchr(Txt, '.');
    *(res + 2) = 0x0;                      // Locate "."
    *(res + 3) = 0x0;                      // Terminate after 1 digit
    Glcd_Write_Text(Txt, 45, 3, 1);
}

```

**Figure 7.99**

cont'd

```
// Display altitude
for(i=0; i<14; i++)Txt[i] = 0;
FloatToStr(Altitude, Txt);
res = strrchr(Txt,'.');
*(res + 2) = 0x0;
Glcd_Write_Text(Txt, 45, 5, 1);
}

// 
// Display Error message
//
void Error()
{
    Glcd_Write_Text("Error...", 5,3,1);                                // Wrong device ID
    while(1);
}

void main()
{
    unsigned char stat;
    long int P;
    float T, A;

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

    Glcd_Init();                                                     // Initialize GLCD
    Glcd_Fill(0x00);                                                 // Clear GLCD

    I2C1_Init(100000);                                              // Initialize I2C
    Delay_Ms(10);

    stat = Init_Pressure_Chip();                                     // Initialize pressure chip
    if(stat != 0)Error();
}

// 
// Loop to read and display the pressure, temperature and altitude
//
while(1)
{
    P = Read_Pressure_Value();                                         // Read pressure
    T = Read_Temperature_Value();                                       // Read temperature
    A = Read_Altimeter_Value(P);                                       // Read altitude
    Display_PTA(P, T, A);                                              // Display Pressure, Temp, Altitude
    Delay_Ms(5000);                                                    // Wait 5 seconds
    Glcd_Fill(0x00);                                                   // Clear GLCD
}
}
```

**Figure 7.99**  
cont'd



Figure 7.100: Sample Display from the Project.

Read\_Altimeter\_Value: Use the above formula to convert pressure into altitude.

Display\_PTA: Display the pressure, temperature, and altitude on the GLCD.

Error: Display error message if the sensor chip is not identified.

Figure 7.100 shows a sample display.

### ***Project 7.12—Plotting the Temperature Variation on the GLCD***

#### ***Project Description***

This project demonstrates how the ambient temperature can be measured and then plotted in real time on the GLCD. The temperature is measured every second using an LM35DZ-type analog sensor and is then plotted in real-time on the GLCD.

The X and Y axes are drawn on the GLCD, the axes ticks are displayed, and the Y axis is labeled as shown in Figure 7.101. The Y axis is the temperature, and the X axis is the time where every pixel corresponds to 1 s in real time.

#### ***Block Diagram***

The block diagram of the project is shown in Figure 7.102.

#### ***Circuit Diagram***

The circuit diagram of the project is as shown in Figure 7.103. The LM35DZ temperature sensor is connected to analog port RA0 (or AN0) of the microcontroller. The sensor

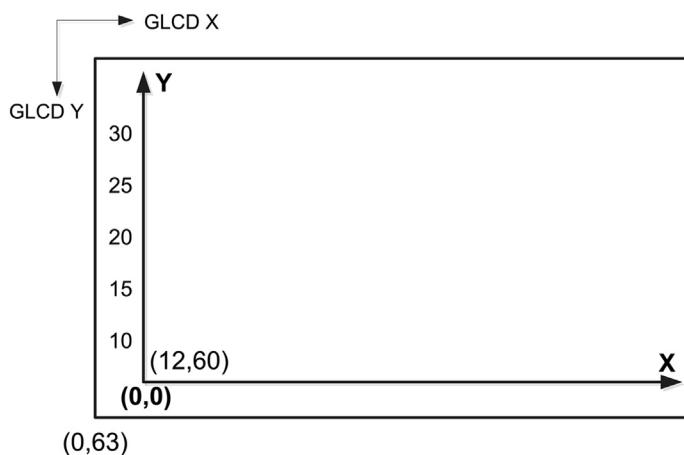


Figure 7.101: Layout of the Screen.

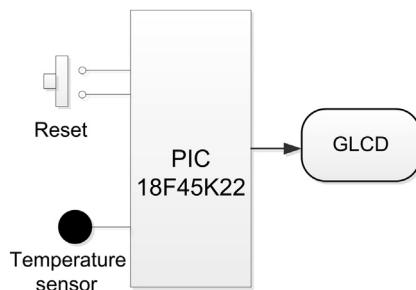


Figure 7.102: Block Diagram of the Project.

provides an output voltage directly proportional to the measured temperature. The output of the sensor is given by

$$V_o = 10 \text{ mV/}^{\circ}\text{C.}$$

PORT B and PORT D are connected to the GLCD as in the previous project.

### **Project PDL**

The PDL of this project is given in [Figure 7.104](#).

### **Project Program**

#### *mikroC Pro for PIC*

The mikroC Pro for the PIC program is given in [Figure 7.105](#) (MIKROC-GLCD3.C). The A/D converter on the PIC18F45K22 microcontroller is 10 bits wide. Thus, with a +5-V

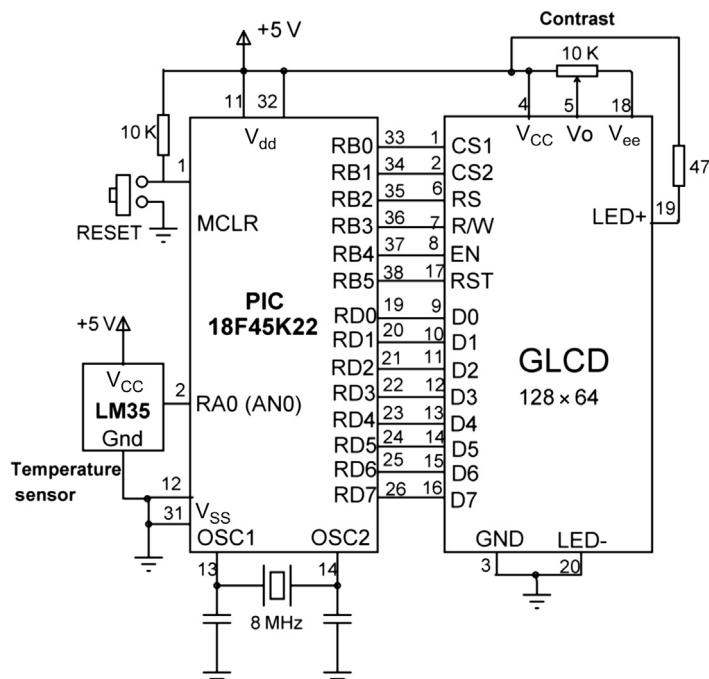


Figure 7.103: Circuit Diagram of the Project.

reference voltage the resolution will be  $5000/1024$  or 4.88 mV, which is accurate enough to measure the temperature to an accuracy of 0.5 °C.

The microcontroller is operated from an 8 MHz crystal. The PLL is disabled so that the actual running clock frequency is 8 MHz.

At the beginning of the program, the connections between the microcontroller and the GLCD are defined using *sbit* statements. The GLCD is connected to ports B and D of the microcontroller and thus both these ports are configured as digital output ports using ANSEL and TRIS statements. PORTA is configured as analog with pin RA0 (or AN0) being configured as an input.

The GLCD library is then initialized using the *Glcd\_Init* function. This function must be called before calling to any other GLCD function. The GLCD screen is then cleared using the *Glcd\_Fill(0x0)*, which turns OFF all pixels of the GLCD.

The A/D converter is initialized by calling library function *ADC\_Init*. The background of the display is drawn by calling function *PlotAxis*. This function draws the X and Y axes. The bottom left part of the screen with coordinates (12, 60) is taken as the (0, 0) coordinate of our display. Then, ticks are placed on both the X and the Y axes using

**Main Program****BEGIN**

Define the connection between the LCD and the microcontroller

Configure PORTB and PORTD as digital output

Configure PORTA as analog input

Initialise GLCD

Clear GLCD

Initialise A/D converter

**CALL** PlotAxis**DO FOREVER**

Read analog temperature from Channel 0

Convert into millivolts

Convert into Degrees centigrade

Calculate the Y co-ordinate based on temperature reading

**CALL** PlotXY to plot the temperature

Wait 1 second

**ENDDO****END****BEGIN/PlotAxis**

Draw X and Y axes

Draw axes ticks

Draw Y axis labels

**END/PlotAxis****BEGIN/PlotXY**

Draw a line to join previous and current temperature values

Update the previous X and Y values with current values

**END/PlotXY****Figure 7.104: PDL of the Project.**

Glcd\_Dot statements. The Y axis is labeled from 10 to 30 °C in steps of 5 °C using the Glcd\_Write\_Text\_Adv statements.

The program then enters an endless loop formed by a *for* statement. Inside this loop, the analog temperature is converted into digital format and stored in variable *T* by calling function ADC\_Get\_Sample with the channel number specified as 0 (RA0 or AN0). This digital value is converted into millivolts by multiplying with 5000 and dividing by 1024. The actual temperature in degrees Celsius is calculated by dividing the voltage in millivolts by 10 ( $V_o = 10 \text{ mV}/\text{°C}$ ).

The graph is drawn using the GLCD function Glcd\_Line. This function draws a line between the specified starting and ending X and Y coordinates. Variables old\_x, old\_y, new\_x, and new\_y are used to store the old and the new (current) X and Y values of the temperature, respectively. At the first iteration, the old and the current values are assumed to be the same, and this is identified by variable *flag* being cleared to 0. In all other

```
*****
        TEMPERATURE PLOTTING ON GLCD
*****
```

This project shows how the temperature can be read from an analog temperature sensor and then plotted on a GLCD in real time.

In this project an LM35DZ type analog temperature sensor is used. This sensor has 3 pins: The ground, power supply (+5 V), and the output pin. The sensor gives an output voltage which is directly proportional to the measured temperature. i.e.  $V_o = 10mV/C$ . Thus, for example at 15C the output voltage is 150 mV. Similarly, at 30C the output voltage is 300 mV and so on.

The temperature sensor is connected to analog input RA0 (or AN0) of a PIC18F45K22 type microcontroller. The microcontroller is operated from an 8 MHz crystal, with the PLL is disabled, so that the actual clock frequency is 8 MHz. The GLCD used in the project is based on KS0107/108 type controller with 128 x 64 pixels.

The program first draws the X and Y axes, axes ticks, and the Y axis labels. Then, the temperature is read from Channel 0 (RA0 or AN0), converted into digital, and then into Degrees C. The temperature is plotted in real-time every second. i.e. the horizontal axis is the time where each pixel corresponds to 1 s.

The GLCD is connected to PORTB and PORTD of the microcontroller as in the previous GLCD projects.

Author: Dogan Ibrahim  
 Date: October, 2013  
 File: MIKROC-GLCD3.C

```
******/
```

```
unsigned char stp, old_x, old_y, new_x, new_y;
```

```
// Glcd module connections
char GLCD_DataPort at PORTD;
```

```
sbit GLCD_CS1 at RB0_bit;
sbit GLCD_CS2 at RB1_bit;
sbit GLCD_RS at RB2_bit;
sbit GLCD_RW at RB3_bit;
sbit GLCD_EN at RB4_bit;
sbit GLCD_RST at RB5_bit;
```

```
sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;
// End Glcd module connections
```

**Figure 7.105: mikroC Pro for the PIC Program.**

```

// This function plots the X and Y axis. The origin is set at screen co-ordinates (12,60).
// First the two axes are drawn. Then the axes ticks are displayed for both X and Y axis.
// Finally, the Y axis labels are displayed (i.e. the temperature labels)
//
void PlotAxis()
{
    unsigned char i;

    Glcd_Line(12, 0, 12, 60, 1);                                // Draw Y axis
    Glcd_Line(12, 60, 127, 60, 1);                             // Draw X axis
    for(i=12; i<127; i += 9)Glcd_Dot(i, 61, 1);             // Display x axis ticks
    for(i=0; i<60; i += 10)Glcd_Dot(11, i, 1);            // Display y axis ticks
    Glcd_Write_Text_Adv("30",0,5);                            // Y axis label
    Glcd_Write_Text_Adv("25",0,15);                           // Y axis label
    Glcd_Write_Text_Adv("20",0,25);                           // Y axis label
    Glcd_Write_Text_Adv("15",0,35);                           // Y axis label
    Glcd_Write_Text_Adv("10",0,45);                           // Y axis label
}

//
// This function plots the temperature in real-time. The temperature is plotted by joining
// the data points with straight lines. The X axis is the time where each pixel corresponds
// to one second. The Y axis is the temperature in Degrees C
//
void PlotXY(float Temperature)
{
    Glcd_Line(old_x,old_y,new_x,new_y,1);                    // Draw temperature changes
    old_x = new_x;                                         // Update old points
    old_y = new_y;
}

//
// Start of main program
//
void main()
{
    unsigned int T;
    unsigned char flag = 0;
    float mV, C;

    ANSELA = 1;                                              // Configure PORTA as analog
    ANSELB = 0;                                              // Configure PORTB as digital
    ANSELD = 0;                                              // Configure PORTD as digital
    TRISA = 1;                                               // RA0 is input (analog)
    TRISB = 0;                                               // PORT B is output
    TRISD = 0;                                               // PORT D is output

    Glcd_Init();                                            // Initialise GLCD
    Glcd_Fill(0x0);                                         // Clear GLCD
}

```

**Figure 7.105**  
cont'd

```

ADC_Init();
          // Initialise ADC
PlotAxis();
          // Plot X-Y axes and labels

for(;;)
{
    T = ADC_Get_Sample(0);
          // Read temperature from channel 0
    mV = T*5000.0/1024.0;
          // Temperature in mV
    C = mV /10.0;
          // Temperature in Degrees C

    if(flag == 0)
    {
        new_x = 12;
          // Start from x = 12
        old_x = new_x;
        new_y = -2*C+70;
          // New temperature value
        old_y = new_y;
        flag = 1;
          // Set so that not first time
    }
    else
    {
        new_x++;
          // Inc x by 1 (1 second each pixel)
        new_y = -2*C+70;
          // New temperature value
    }
    PlotXY(C);
          // Plot the graph
    Delay_Ms(1000);
          // Wait 1 s
}
}

```

**Figure 7.105**

cont'd

iterations, variable *flag* is 1 and the *else* part of the *if* statement is executed. The X value is incremented by 1 to correspond to the next second and the new Y value is updated.

The Y coordinate (temperature) is calculated as follows:

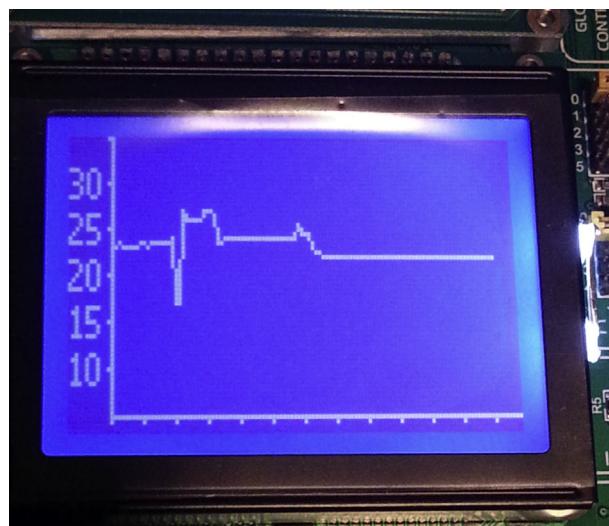
The relationship between the Y axis ticks and the Y coordinates of data values can be derived from the table:

Y Axis Ticks Pixel Coordinates	Y Axis Data Coordinate (Degrees C)
10	30
20	25
30	20
40	15
50	10

The above relationship is linear and is in the form of a straight line  $y = mx + C$ , where  $m$  is the slope of the line and  $C$  is the point where the line crosses the Y axis. The equation of this line can be found from

$$y - y_1 = m(x - x_1),$$

where  $m = (y_2 - y_1)/(x_2 - x_1)$



**Figure 7.106: Sample Display of the Temperature.**

by taking any two points on the line, we can easily find the equation. Considering the points

$$(x_1, y_1) = (30, 10) \text{ and } (x_2, y_2) = (10, 50)$$

The relationship is found to be

$$y = -2x + 70.$$

Therefore, given the temperature C in degrees Celsius, the y coordinate to be used for plotting can be calculated from

$$\text{new\_y} = -2*C + 70.$$

After plotting a point, the *new\_x* and *new\_y* are copied to *old\_x* and *old\_y*, respectively, ready for the next sample to be plotted.

Figure 7.106 shows a sample display of the temperature in real time.

### **Project 7.13—Using the Ethernet—Web Browser-Based Control**

The Ethernet has traditionally been implemented on PCs and laptops and has been used widely at homes, offices, and industries to access the worldwide Internet and companywide intranet networks. The Internet can nowadays be accessed using smaller handheld devices such as smart mobile phones, PDAs and, IPADs. Most of these devices are based on microcontrollers and use single-chip Ethernet-controller devices for connectivity. Such Ethernet controllers can easily be configured, programmed, and

incorporated into embedded systems to provide the system with Ethernet connectivity with the outside world.

### **Ethernet Connectivity**

Ethernet was originally invented by Xerox in 1972, and then developed jointly by Xerox, DEC, and Intel. It is a frame-based networking technology based on the standard IEEE 802.3. The physical medium of a typical Ethernet-based local area network (LAN) network uses coaxial cable, twisted pair wires, fiber optics, or can be in the form of Wireless LANs. Currently, the most common form of Ethernet is called 100Base-T, and it provides transmission speeds up to 100 Mbps. Slower Ethernet or 10Base-T is also commonly used in lower speed control and monitoring projects.

Devices on the Ethernet are all connected together, and the communication is based on the *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD) protocol. Only one node transmits its data while all the other nodes listen to avoid collision. In the case of a possible collision, transmitting nodes wait for a random time and attempt to retransmit, hoping to avoid the collision. The maximum length of an Ethernet cable depends on the speed of transmission and the type of cable used.

As shown in [Figure 7.107](#), an Ethernet packet consists of

- Six-byte destination address,
- Six-byte source address,
- Two-byte data type,
- Forty five to 1500 byte data,
- Four-byte CRC.

In addition, when transmitted on the Ethernet medium, a 7-byte preamble field and Start-of-Frame (SOF) delimiter byte are appended to the beginning of the Ethernet packet.

Various network communication protocols are embedded inside Ethernet packets. For example, DECnet, IP, and ARP protocols all make use of the Ethernet as the communications protocol. In this article, we will be using a Web Browser command to establish communication between the PC and the microcontroller system. Web Browser is based on the transmission control protocol (TCP) and uses port 80. TCP is an advanced protocol requiring connection and providing guaranteed packet delivery with

DESTINATION ADDRESS	SOURCE ADDRESS	TYPE	DATA	CRC
---------------------	----------------	------	------	-----

**Figure 7.107: Ethernet Packet Format.**

retransmission if an error occurs. TCP packets are acknowledged to confirm the safe packet delivery.

### ***Embedded Ethernet Controller Chips***

There are many Ethernet controller chips in the market. Although these chips can be purchased as components, in most applications, it is easier and usually cheaper to use boards with incorporated Ethernet controller chips and network connection sockets (e.g. RJ45). The ENC28J6 is a standalone 28-pin Ethernet controller chip that meets the IEEE 802.3 specifications and is controlled using the SPI interface. This chip is used in the project given later in this section. This chip has the following basic features:

- Compatible with 10Base-T networks,
- Supports both half-duplex and full-duplex operation,
- Supports automatic polarity detection and correction,
- Automatic retransmit on collision,
- Eight-kilobyte transmit/receive buffer,
- Supports unicast, multicast, and broadcast addresses,
- Link and Activity LED interface,
- Differential signal interface to RJ45 connector.

Figure 7.108 shows the block diagram and connection of the ENC28J60 Ethernet controller chip to a microcontroller. Basically, the interface requires the SPI signals SI, SO, and SCK to be connected to the microcontroller. In addition, the CS pin can also be connected to a microcontroller I/O pin.

Some high-end PIC microcontrollers incorporate Ethernet controllers. For example, the PIC18F97J60 is a microcontroller that includes a 10Base-T Ethernet controller with 8-kbyte transmit/receive buffers. The advantage of using an Ethernet-based microcontroller chip is that in addition to the Ethernet functions, the chip provides analog and digital I/O ports and many other microcontroller features.

### ***Embedded Ethernet Access Methods***

In general, there are four access methods that can be used to establish the connectivity between a PC and an embedded Ethernet controller (see Application Note AN292, Silicon Labs):

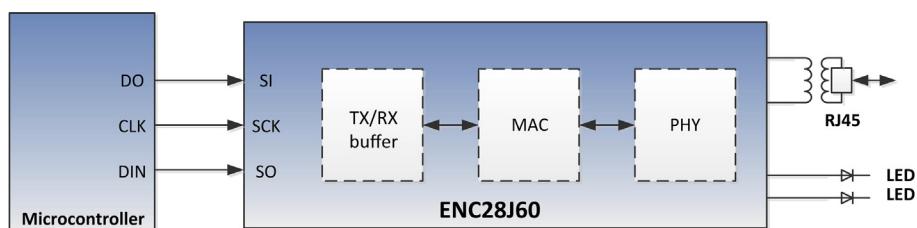


Figure 7.108: Connecting the ENC28J60 Ethernet Controller to a Microcontroller.

- Using a Web Browser on the PC,
- Using a HyperTerminal on the PC,
- Having the embedded system to send E-mail,
- Using a custom application based on developing software on both the PC and the embedded system,

#### *Using a Web Browser on the PC*

This is perhaps the easiest and the most reliable method of establishing connectivity with no software development on the PC. This method is based on HTTP, which has been in use since the 1990s as the most widely used protocol to transfer data on the internet. The aim of HTTP protocol is to allow the transfer of HTML files between a browser (usually a PC) and a Web Server where the data item is located. In this method, the PC is termed the *Client* and the microcontroller system is termed the *Server*. The client sends a request by entering the *url* of the server. Assuming that the server *url* is 192.168.10.15, then entering the following command on the PC will establish a link to the microcontroller system:

<http://192.168.10.15>

The microcontroller system, for example, can then send an HTML page as a response to the client to display a menu with buttons. By clicking a button on the menu, a command (e.g. GET) will be sent to the server with the appropriate command tail. The server can decode this command tail and take appropriate actions.

Figure 7.109 shows the connectivity using a Web Browser interface.

#### *Using a HyperTerminal*

The HyperTerminal interface is also known as the *Telnet* interface. Here, the user connects to the microcontroller system by issuing Telnet commands and specifying the IP address. This kind of interface is usually an interactive interface and requires the connectivity to be initiated and terminated by the user on the PC.

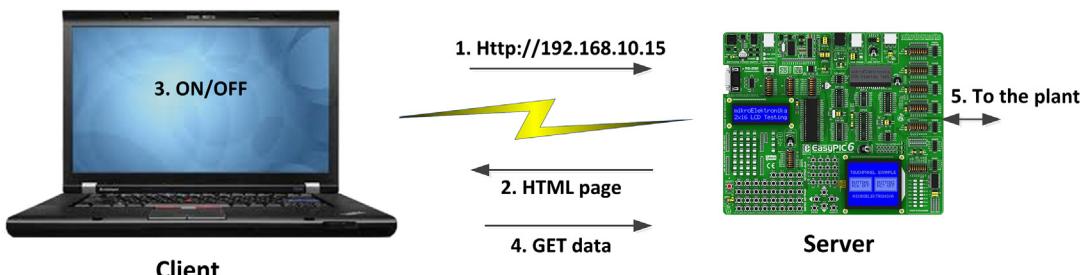


Figure 7.109: Web Browser Connectivity.



**Figure 7.110: Connectivity using the E-mail Method.**

### *Embedded System Sending E-mail*

In this method, the microcontroller system sends its data using the E-mail protocols. The outgoing E-mail is handled by SMTP, while the incoming mail is handled by mail servers such as POP, IMAP, or HTTP. Using this method has disadvantages that an incoming mail may stay in the input buffers for a long time until it is discovered and read by the user. Figure 7.110 shows the connectivity using the E-mail method.

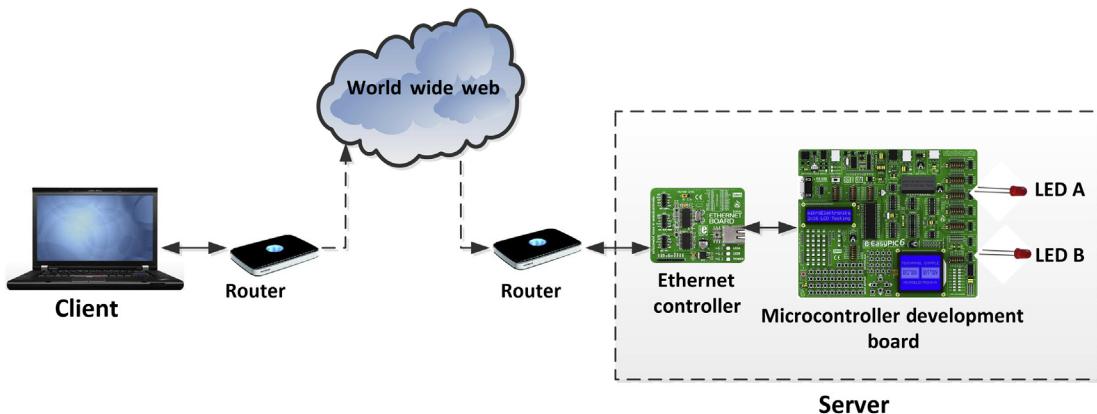
### *Using Custom Application*

The development of custom applications for network connectivity provides a highly flexible interface. This method however has the greatest disadvantage that network software should be developed on both the PC and the microcontroller system. Two protocols are usually used for custom development: user datagram package (UDP) and TCP. The TCP protocol is used in applications where a guaranteed packet delivery is required with the delivery of each packet being acknowledged. Lost packets are retransmitted. The UDP protocol on the other hand is used where a high transmission speed is more important than the safe delivery of packets. There is no acknowledgement and no retransmission if a packet is lost.

Table 7.18 compares the TCP and the UDP protocols.

**Table 7.18: Comparing the TCP and UDP Protocols.**

Feature	TCP	UDP
Speed	Slow. Packets acknowledged, lost packets retransmitted	Fast. No acknowledgement, no retransmission
Complexity	High	Low
Connectivity	Connection required between two devices	Connection is not required
Packet delivery	Guaranteed	Not guaranteed. Lost packets are not retransmitted
Packet overhead	Large	Small



**Figure 7.111: Block Diagram of the Project.**

### *Example Ethernet-Based Embedded Control Project*

This section describes the design of a simple microcontroller-based automation system using the Ethernet as the communication medium. In this project, a web browser-based communication is used where the PC is a client and the microcontroller system is the server. [Figure 7.111](#) shows the block diagram of the project. The project hardware is in two parts, connected using a network hub or a switch (or a crossed network cable for local testing): the Ethernet controller (or the Server) and the PC (or the Client).

The Ethernet controller consists of a microcontroller and an ENC28J60 Ethernet controller chip. Two LEDs (LED A and LED B) are connected to the microcontroller RD0 and RD1 output pins to simulate two lamps. These LEDs are toggled under the control of a standard Web Browser command initiated on the PC. There is no software development on the PC.

### **Project Hardware**

[Figure 7.112](#) shows the circuit diagram of the project. The microcontroller is designed around a PIC18F45K22-type microcontroller chip, operating at 8 MHz. The Ethernet controller is based on the ENC28J60 chip, operating at 25 MHz. The interface between the microcontroller and the Ethernet chip is based on the SPI bus protocol, where SI, SO, and SCK pins of the Ethernet chip are connected to SPI pins (PORTC) of the microcontroller. The Ethernet controller chip operates at 3.3 V, and thus, its output pin SO cannot drive the microcontroller input pin without a voltage translator. In [Figure 7.112](#), a 74HCT245-type buffer is used to boost the output level of pin SO. Other lower cost chips, such as 74HCT08 (AND gate), 74ACT125 (quad 3-state buffer) or other chips could also have been used.

The internal analog circuitry of the ENC28J60 chip requires that an external resistor be connected from RBIA to the ground. Some of the device's digital logic operates at 2.5 V,

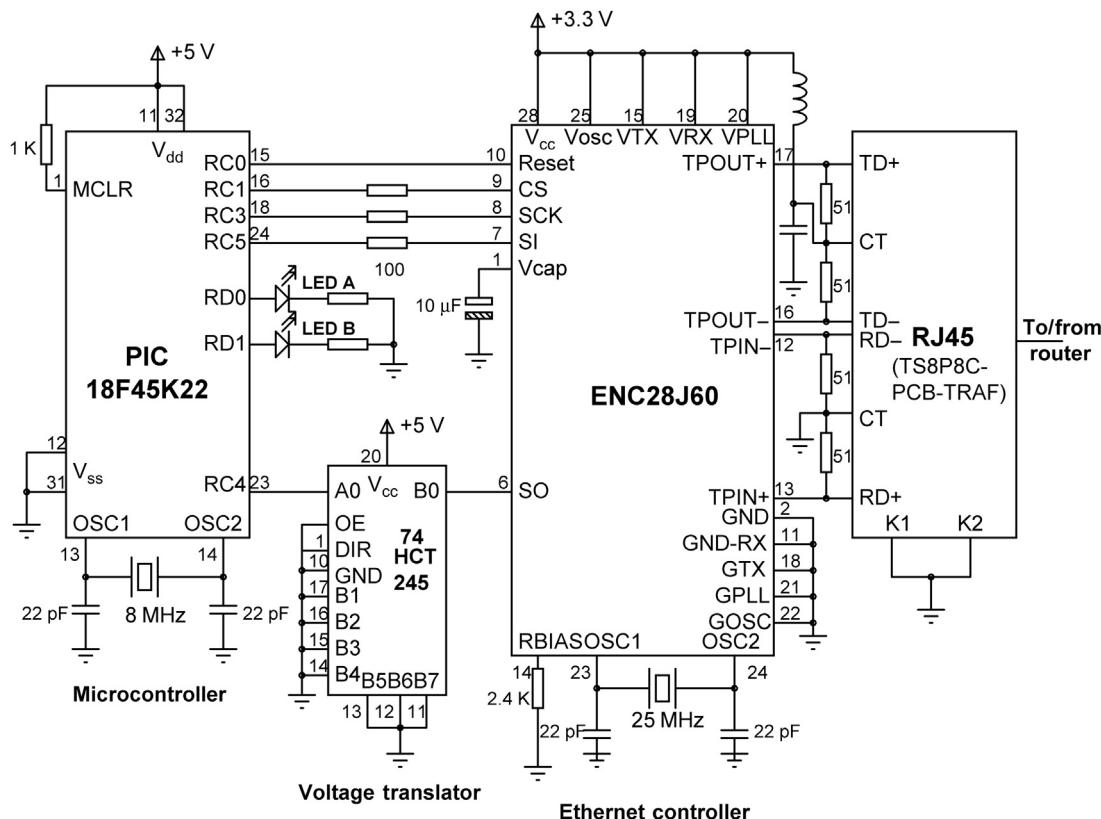


Figure 7.112: Circuit Diagram of the Project.

and an external filter capacitor should be connected from Vcap to ground. Transmit output pins of the Ethernet chip (TPOUT+ and TPOUT-) and the receive inputs (TPIN+ and TPIN-) are connected to an RJ45 network socket with an integrated Ethernet transformer (T58P8C-PCB-TRAF). Two LEDs on the Serial Ethernet board provide visual indication of the Link and Activity on the line (the RJ45 socket has a pair of built-in internal LEDs, but are not used in this project). A 5- to 3.3-V power supply regulator chip (e.g. MC33269DT-3.3) is used to provide power to the Ethernet chip. If the PC and the Ethernet controller are on the same network and close to each other, then the two can be connected together using a crossed network cable; otherwise, a hub or a switch may be required. If the PC and the Ethernet controller are located on different networks and are not close to each other, then routers may be required to establish connectivity between the two.

Two LEDs, LED A and LED B, are connected to pins RD0 and RD1 of the microcontroller, respectively. These LEDs are toggled under the control of a Web Browser command issued from the PC.

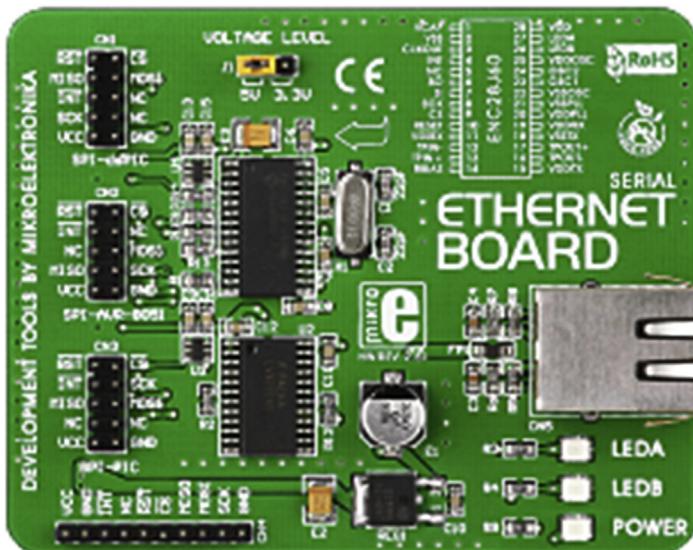


Figure 7.113: The Serial Ethernet Board.

### ***The Construction***

The project was constructed using the EasyPIC V7 development board and the mikroElektronika Serial Ethernet Board ([Figure 7.113](#)). This is a small board that plugs in directly to PORTC of the EasyPIC V7 development board via a 10-way IDC plug ([Figure 7.114](#)) simplifying the development of embedded Ethernet projects. The board is equipped with an EC28J60 Ethernet controller chip, a 74HCT245 voltage translation chip, three LEDs, a 5- to 3.3-V voltage regulator, and an RJ45 socket with an integrated transformer.

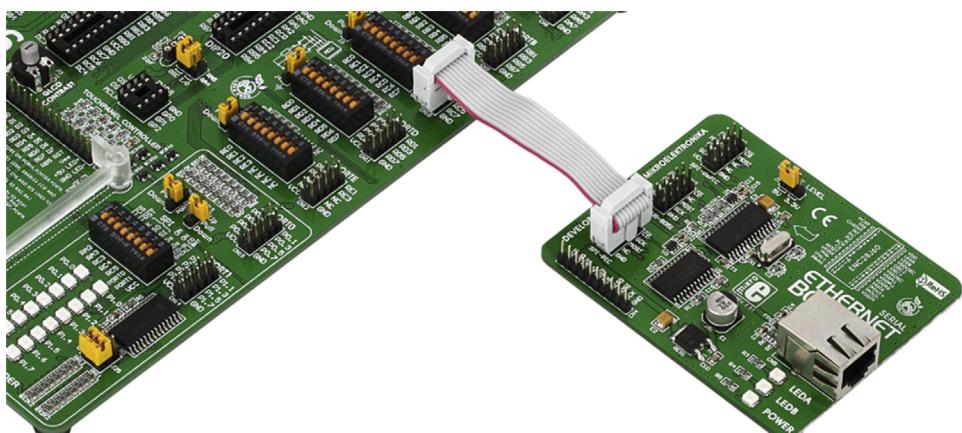


Figure 7.114: Connecting the Serial Ethernet Board to EasyPIC7 V7.

If you are using the EasyPIC V7 development board for this project, use a 10-way ribbon cable and make sure that you plug in one side of the cable to PORTC on the development board, and the other side to the bottom connector on the Serial Ethernet board (Figure 7.114).

### **Project PDL**

Figure 7.115 shows the project PDL.

#### **Project Software**

##### *mikroC Pro for PIC*

The mikroC Pro for PIC program listing is given in Figure 7.116 (MIKROC-ETHER.C). At the beginning of the main program, PORTC and PORTD are configured as digital, and the SPI bus is initialized by calling built-in library function *SPI\_Init*. Then, the Serial Ethernet module is initialized by calling built-in function *SPI\_Ethernet\_Init* and specifying the MAC address of the ethernet board, the IP address to be used, and the mode of operation as full duplex.

MAIN program

```
BEGIN
    Configure I/O ports
    Initialise SPI bus
    Initialise Serial Ethernet Library
    DO FOREVER
        Check for packets
    ENDDO
END

BEGIN/SPI_Ethernet_UserTCP
    IF a GET packet is received THEN
        IF two characters starting at index 6 are "TA"
            Toggle LED A
        ELSE IF two characters starting at index 6 are "TB"
            Toggle LED B
        ENDIF
    ENDIF
    Send HTML response page to the Client
END/SPI_Ethernet_UserTCP

BEGIN/SPI_Ethernet_UserUDP
END/SPI_Ethernet_UserUDP
```

Figure 7.115: Project PDL.

```
=====
WEB BROWSER BASED ETHERNET CONTROL PROJECT
=====

This project shows how the ETHERNET can be used in microcontroller based projects. In this
project a Serial Ethernet Board (www.mikroe.com) is connected to the EasyPIC V7 development
board.

The project uses the Web Browser method to establish Ethernet based communication between
a PC and the microcontroller system.

The PC is the client and the microcontroller system is the server.

Two LEDs (LED A and LED B) are connected to the microcontroller system. These LEDs are toggled
remotely by entering commands on the PC. The HTTP protocol is used in the project

Author: Dogan Ibrahim
Date: October, 2013
File: MIKROC-ETHER.C
=====

const char HTTPHeader[] = "HTTP/1.1 200 OK\nContent-type:";
const char HTTPMimeTypeHTML[] = "text/html\n\n";
const char HTTPMimeTypeScript[] = "text/plain\n\n";
//
// Define the HTML page to be sent to the PC
//
char StrtPage[] =
"<html><body>\n<form name=\"input\" method=\"get\"><table align=center width=500 \
bgcolor=Red border=4><tr><td align=center colspan=2><font size=7 \
color=white face=\"verdana\"><b>LED CONTROL</b></font></td></tr>\n<tr><td align=center bgcolor=Blue><input name=\"TA\" type=\"submit\" \
value=\"TOGGLE LED A\"></td><td align=center bgcolor=Green> \
<input name=\"TB\" type=\"submit\" value=\"TOGGLE LED B\"></td></tr>\n</table></form></body></html>";

//
// Ethernet NIC interface definitions
//
sfr sbit SPI_Ethernet_Rst at RCO_bit;
sfr sbit SPI_Ethernet_CS at RC1_bit;
sfr sbit SPI_Ethernet_Rst_Direction at TRISCO_bit;
sfr sbit SPI_Ethernet_CS_Direction at TRISC1_bit;
//
// Define Serial Ethernet Board MAC Address, and IP address to be used for the communication
//
unsigned char MACAddr[6] = {0x00, 0x14, 0xA5, 0x76, 0x19, 0x3f} ;
unsigned char IPAddr[4] = {192,168,1,15};
unsigned char getRequest[10];

typedef struct
```

**Figure 7.116: Program Listing of the Project.**

```
{  
    unsigned canCloseTCP:1;  
    unsigned isBroadcast:1;  
}TEthPktFlags;  
  
//  
// TCP routine. This is where the user request to toggle LED A or LED B are processed  
//  
//  
unsigned int SPI_Ethernet_UserTCP(unsigned char *remoteHost,  
                                  unsigned int remotePort, unsigned int localPort,  
                                  unsigned int reqLength, TEthPktFlags *flags)  
{  
    unsigned int Len;  
    for(len=0; len<10; len++) getRequest[len]=SPI_Ethernet_getByte();  
    getRequest[len]=0;  
    if(memcmp(getRequest,"GET /",5))return(0);  
  
    if(!memcmp(getRequest+6,"TA",2))RD0_bit = ~ RD0_bit;  
    else if(!memcmp(getRequest+6,"TB",2))RD1_bit = ~ RD1_bit;  
  
    if(localPort != 80)return(0);  
    Len = SPI_Ethernet_putConstString(HTTPheader);  
    Len += SPI_Ethernet_putConstString(HTTPMimeTypeHTML);  
    Len += SPI_Ethernet_putString(StrtPage);  
    return Len;  
}  
  
//  
// UDP routine. Must be declared even though it is not used  
//  
unsigned int SPI_Ethernet_UserUDP(unsigned char *remoteHost,  
                                  unsigned int remotePort, unsigned int destPort,  
                                  unsigned int reqLength, TEthPktFlags *flags)  
{  
    return(0);  
}  
  
//  
// Start of MAIN program  
//  
void main()  
{  
    ANSELc = 0;                                // Configure PORTC as digital  
    ANSELD = 0;                                // Configure PORTD as digital  
    TRISD = 0;                                 // Configure PORTD as output  
    PORTD = 0;  
    SPI1_Init();                               // Initialize SPI module  
    SPI_Ethernet_Init(MACAddr, IPAddr, 0x01);   // Initialize Ethernet module  
  
    while(1)                                    // Do forever  
    {  
        SPI_Ethernet_doPacket();                // Process next received packet  
    }  
}
```

**Figure 7.116**  
cont'd

```

<html>
<body>
<form name="input" method="get">
<table align=center width=500 bgcolor=Red border=4>
<tr>
<td align=center colspan=2><font size=7 color=white face="verdana"><b>LED CONTROL</b></font></td>
</tr>
<tr>
<td align=center bgcolor=Blue><input name="TA" type="submit" value="TOGGLE LED A"></td>
<td align=center bgcolor=Green><input name="TB" type="submit" value="TOGGLE LED B"></td>
</tr>
</table>
</form>
</body>
</html>

```

**Figure 7.117: HTML Code Sent to the Web Browser.**

The MAC address of the Serial Ethernet Board used in this project is set at factory to “0x00, 0x14, 0xA5, 0x76, 0x19, 0x3F”. The program sets the IP address of the board to “192.168.1.15”. The main program then enters an infinite loop where built-in library function *SPI\_Ethernet\_doPacket* is called to check for the arrival of packets and also to send any outstanding packets to their destinations. The Ethernet library requires both the UDP and TCP functions to be present in the program even though they may not be used. Only the TCP is used in this example as the Web Browser communication is based on TCP. Inside the TCP function, any received packets are checked, and the function continues if the packets are of type “GET/”. Then, the command tail is checked and the LEDs are toggled as required. The transmit buffer is loaded with the HTML response and the length of the buffer is returned from the function which then sends the buffer to the client.

The array StrtPage at the beginning of the program defines the HTML page to be sent to the PC so that the PC can display it. This page is made up of the following commands. This HTML script basically displays a form (Figure 7.118) on the PC screen with two buttons. Clicking TOGGLE LED A toggles the state of LED A (if the LED is ON it turns OFF, and vice versa), and similarly, clicking TOGGLE LED B button toggles the state of LED B:



**Figure 7.118: Form Displayed by the Web Browser on the PC.**

```
char StrtPage[] =  
<html><body>\n<form name=\"input\" method=\"get\"><table align=center width=500 \\  
bgcolor=Red border=4><tr><td align=center colspan=2><font size=7 \\  
color=white face=\"verdana\">LED CONTROL</font></td></tr>\n<tr><td align=center bgcolor=Blue><input name=\"TA\" type=\"submit\" \\  
value=\"TOGGLE LED A\"></td><td align=center bgcolor=Green> \\  
<input name=\"TB\" type=\"submit\" value=\"TOGGLE LED B\"></td></tr>\n</table></form></body></html>;
```

The connectivity of the system can be checked by using the PING command to send packets from the PC to the Ethernet controller. If everything is working as expected, then PING replies should be displayed on the PC screen. To use the PING command, Click the START button (Windows 7) and type CMD, followed by the Enter key. Then enter the command

PING 192.168.1.15

You should get a response similar to the following lines:

```
Pinging 192.168.1.15 with 32 bytes of data:  
Reply from 192.168.1.15: bytes = 32 time = 12 ms TTL = 128  
Reply from 192.168.1.15: bytes = 32 time = 6 ms TTL = 128  
Reply from 192.168.1.15: bytes = 32 time = 6 ms TTL = 128  
Reply from 192.168.1.15: bytes = 32 time = 6 ms TTL = 128  
  
Ping statistics for 192.168.1.15:  
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss)  
    Approximate round trip time in milliseconds:  
        Minimum = 6 ms, Maximum = 12 ms, Average = 7 ms
```

The operation of the system is described below in steps:

- Compile and load the program to the microcontroller. Connect the Serial Ethernet board to PORTC of the development board. Connect the PC and the Serial Ethernet board together using a hub, switch, or router, or using a crossed network cable for local testing.
- Open a web browser on the PC (e.g. Microsoft Internet Explorer or Firefox) and send an HTTP request by entering the following url: <http://192.168.1.15>
- Upon receipt of this request, the Ethernet controller sends the HTML code shown in [Figure 7.117](#) to the Web Browser, together with the HTTP header. The *Content-Type* field is used by the browser to tell which format the document it receives is in. HTML is identified with “text/html”, and ordinary text is identified with “text/plain”.
- The Web Browser then displays the form shown in [Figure 7.118](#).
- The user can toggle LED A or LED B by clicking on the appropriate buttons. Assuming that button LED A is clicked, the Web Browser sends the following command to the Ethernet controller:

GET /? TA=TOGGLE\_LED+A

Similarly, if button B is pressed, the Web Browser sends the following command to the Ethernet controller:

GET /? TB = TOGGLE\_LED + B

- The Ethernet controller checks the received command (inside function TCP) and toggles LED A or LED B as required.

### ***Project 7.14—Using the Ethernet—UDP-Based Control***

This is another project using the embedded Ethernet. In this project, communication is established between the PC and the microcontroller system using the UDP protocol. Eight LEDs are connected to PORTD of the microcontroller. A Graphical User Interface (GUI) program is developed on the PC using the Visual Studio, Visual Basic program (VB.NET). The user specifies which bits of PORTD should be turned ON by clicking the appropriate parts of a GUI form. The PC program establishes UDP communication with the microcontroller system and sends a packet about the PORTD bits that should be turned ON. The microcontroller system uses the UDP protocol to receive this packet and then turns ON the required bits of PORTD.

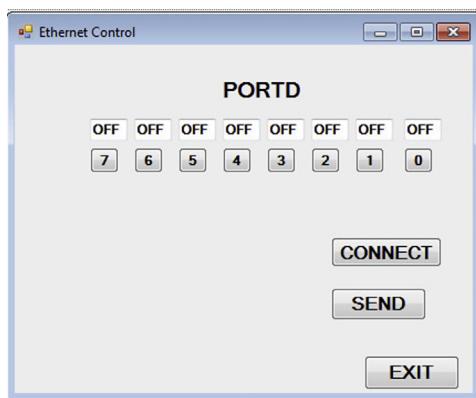
#### ***The Hardware***

The circuit diagram of the project is similar to the one given in [Figure 7.112](#), but here, eight LEDs are connected to PORTD instead of just 2.

#### ***The PC Program***

The PC program is based on VB.NET. When the program is run, the form shown in [Figure 7.119](#) is displayed. The form consists of 8 Textboxes (called TextBox0 to TextBox7) and 8 Buttons (called Button0 to Button7). The Textboxes correspond to the LEDs connected to the microcontroller, and they can take two values: ON or OFF. Clicking a button under a textbox toggles the contents of the Textbox from ON to OFF, or from OFF to ON. When in state ON, the corresponding LED is ON.

After selecting which bits of PORTD should be ON, the user should click the CONNECT button to establish UDP communication with the microcontroller system. After this, the SEND button should be clicked to send a packet to the microcontroller system so that the required LEDs can be turned ON. Byte array PortArray has elements as either 0 or 1, and it stores the required state of each individual bit. The UDP packet consists of 1 byte only which is the byte stored in variable PortValue, corresponding to



**Figure 7.119: The PC Form.**

the PORTD bits to be turned ON. Function ClientSocket.Send is used to send this byte to the microcontroller.

The PC program listing is shown in [Figure 7.120](#).

### ***The Microcontroller Program***

#### ***mikroC Pro for PIC***

The microcontroller program is shown in [Figure 7.121](#) (MIKRO-UDP.C). At the beginning of the program, the connections between the Serial Ethernet board and the microcontroller are defined. Then the Ethernet board MAC address and the IP address to be used in the UDP communication are defined.

This program uses the UDP, but the function SPI\_Ethernet\_UserTCP must be declared with a return statement, even though the, TCP protocol is not used in the program.

The SPI\_Ethernet\_UserUDP is very simple. It checks the port number, and if the remote port number is invalid (i.e. is not 10001), then the function returns. Otherwise, the received packet is stored in array Txt using function SPI\_Ethernet\_getByte, and the contents of Txt[0], which is the byte corresponding to user's selection is sent to PORTD.

### ***Project 7.15—Digital Signal Processing—Low Pass FIR Digital Filter Project***

Digital filters are very important in many digital signal processing applications. The theory of digital filters is complex and is beyond the scope of this book. It is assumed that the

```
Imports System.Net.Sockets
Imports System.Text
Imports System.Threading

Public Class Form1
    Public Txt(1) As Byte
    Public PortArray(8) As Byte
    Public ClientSocket As New UdpClient

    Public ServerAddress As String = "192.168.1.15"      ' Set the IP address of the server
    Public PortNumber As Integer = 10001                  ' Set port number

    Private Sub Button0_Click(sender As System.Object, e As System.EventArgs) Handles Button0.Click
        If TextBox0.Text = "OFF" Then
            TextBox0.Text = "ON"
            PortArray(0) = 1
        Else
            TextBox0.Text = "OFF"
            PortArray(0) = 0
        End If
    End Sub

    Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles Button1.Click
        If TextBox1.Text = "OFF" Then
            TextBox1.Text = "ON"
            PortArray(1) = 1
        Else
            TextBox1.Text = "OFF"
            PortArray(1) = 0
        End If
    End Sub

    Private Sub Button2_Click(sender As System.Object, e As System.EventArgs) Handles Button2.Click
        If TextBox2.Text = "OFF" Then
            TextBox2.Text = "ON"
            PortArray(2) = 1
        Else
            TextBox2.Text = "OFF"
            PortArray(2) = 0
        End If
    End Sub

    Private Sub Button3_Click(sender As System.Object, e As System.EventArgs) Handles Button3.Click
        If TextBox3.Text = "OFF" Then
            TextBox3.Text = "ON"
            PortArray(3) = 1
        Else
            TextBox3.Text = "OFF"
            PortArray(3) = 0
        End If
    End Sub

    Private Sub Button4_Click(sender As System.Object, e As System.EventArgs) Handles Button4.Click
        If TextBox4.Text = "OFF" Then

```

Figure 7.120: The PC Program.

```
        TextBox4.Text = "ON"
        PortArray(4) = 1
    Else
        TextBox4.Text = "OFF"
        PortArray(4) = 0
    End If
End Sub

Private Sub Button5_Click(sender As System.Object, e As System.EventArgs) Handles
Button5.Click
    If TextBox5.Text = "OFF" Then
        TextBox5.Text = "ON"
        PortArray(5) = 1
    Else
        TextBox5.Text = "OFF"
        PortArray(5) = 0
    End If
End Sub

Private Sub Button6_Click(sender As System.Object, e As System.EventArgs) Handles
Button6.Click
    If TextBox6.Text = "OFF" Then
        TextBox6.Text = "ON"
        PortArray(6) = 1
    Else
        TextBox6.Text = "OFF"
        PortArray(6) = 0
    End If
End Sub

Private Sub Button7_Click(sender As System.Object, e As System.EventArgs) Handles
Button7.Click
    If TextBox7.Text = "OFF" Then
        TextBox7.Text = "ON"
        PortArray(7) = 1
    Else
        TextBox7.Text = "OFF"
        PortArray(7) = 0
    End If
End Sub

Private Sub ButtonConnect_Click(sender As System.Object, e As System.EventArgs)
Handles ButtonConnect.Click
    ClientSocket.Connect(ServerAddress, PortNumber)
End Sub

Private Sub ButtonSend_Click(sender As System.Object, e As System.EventArgs) Handles
ButtonSend.Click
    Dim PortValue As Byte
    PortValue = 0

    For i = 0 To 7
        PortValue = PortValue + PortArray(i) * 2 ^ i
    Next
    Txt(0) = PortValue
    ClientSocket.Send(Txt, 1)
End Sub
```

Figure 7.120

cont'd

```

Private Sub ButtonExit_Click(sender As System.Object, e As System.EventArgs) Handles
ButtonExit.Click
    ClientSocket.Close()
End
End Sub
End Class

```

**Figure 7.120**

cont'd

readers have a sufficient knowledge on finding the filter coefficients and the various digital filtering structures for a given design specifications. There are many books and references on the theory of digital filters that may be helpful.

In this project, we will be designing a low-pass FIR type digital filter with the following specifications:

Window Type	No Windowing
Cut-off frequency	50 Hz
Sampling frequency	1000 Hz
Filter order	10 (11 taps)

The block diagram of the project is shown in [Figure 7.122](#). In this project, a Velleman PCSGU250 is used ([Figure 7.123](#)). This is a device that operates as a frequency generator, oscilloscope, transient recorder, spectrum analyzer, and frequency plotter (Bode plotter).

The analog signal generated by the PCSGU250 is applied to one of the analog channels of the microcontroller. The filtered signal is converted into the analog signal using a DAC chip and is then fed to the PCSGU250 for plotting the frequency response.

### ***The Filter Structure***

There are many software packages that could be used to find the filter coefficients. In this book, the highly popular ScopeFIR program is used.

The steps in finding the filter parameters using the ScopeFIR program are given below:

- Start the program.
- Create a new project by selecting the filter type as low-pass, Windowed Sinc, 11 taps, sampling frequency 100 Hz, and the cut-off frequency of 50 Hz ([Figure 7.124](#)).
- Click Design to design the filter. The frequency response, filter coefficients, and the impulse response of the filter to be designed will be displayed by the program. [Figure 7.125](#) shows the frequency response and [Figure 7.126](#) shows the required filter coefficients.

```
*****
    UDP ETHERNET CONTROL PROJECT
=====
*****
```

This project shows how the ETHERNET can be used in microcontroller based projects. In this project a Serial Ethernet Board ([www.mikroe.com](http://www.mikroe.com)) is connected to the EasyPIC V7 development board.

The project uses the UDP method to establish Ethernet based communication between a PC and the microcontroller system.

The PC is the client and the microcontroller system is the server. The PC program is written using the Visual Studio, Visual Basic programming language (VB.NET). The PC sends a packet to the microcontroller system in the form of a UDP packet.

8 LEDs are connected to PORTD. The user interactively enters the LEDs to be turned ON using a GUI type window. This message is then passed to the microcontroller system via the UDP and then the required LEDs are turned ON by the microcontroller.

Port 10001 and IP address 192.168.1.15 are used in the project.

Author: Dogan Ibrahim

Date: October, 2013

File: MIKROC-UDP.C

```
*****
// Ethernet NIC interface definitions
//
sfr sbit SPI_Ethernet_Rst at RC0_bit;
sfr sbit SPI_Ethernet_CS at RC1_bit;
sfr sbit SPI_Ethernet_Rst_Direction at TRISCO_bit;
sfr sbit SPI_Ethernet_CS_Direction at TRISC1_bit;
//
// Define Serial Ethernet Board MAC Address, and IP address to be used for the communication
//
unsigned char MACAddr[6] = {0x00, 0x14, 0xA5, 0x76, 0x19, 0x3f} ;
unsigned char IPAddr[4] = {192,168,1,15};
unsigned char getRequest[10];

typedef struct
{
    unsigned canCloseTCP:1;
    unsigned isBroadcast:1;
}TEthPktFlags;

//
// TCP routine. This is where the user request to toggle LED A or LED B are processed
//
//
unsigned int SPI_Ethernet_UserTCP(unsigned char *remoteHost,
                                  unsigned int remotePort, unsigned int localPort,
                                  unsigned int reqLength, TEthPktFlags *flags)
```

**Figure 7.121: mikroC Pro for PIC Program.**

```

{
    return (0);
}

// UDP routine. Must be declared even though it is not used
//
unsigned int SPI_Ethernet_UserUDP(unsigned char *remoteHost,
                                  unsigned int remotePort, unsigned int destPort,
                                  unsigned int reqLength, TEthPktFlags *flags)
{
    char Txt[10];
    char len=0;

    if(destport != 10001) return(0);                                // Check that correct port is used
    while(reqLength--)
    {
        Txt[len++] = SPI_Ethernet_getByte();                      // Extract the received bytes
    }

    PORTD = Txt[0];                                              // Turn ON required LEDs
    return;
}

// Start of MAIN program
//
void main()
{
    ANSELc = 0;                                                 // Configure PORTC as digital
    ANSELD = 0;                                                 // Configure PORTD as digital
    TRISD = 0;                                                 // Configure PORTD as output
    PORTD = 0;                                                 // Clear PORTD to start with

    SPI1_Init();                                                 // Initialize SPI module
    SPI_Ethernet_Init(MACAddr, IPAddr, 0x01);                  // Initialize Ethernet module

    while(1)                                                     // Do forever
    {
        SPI_Ethernet_doPacket();                                // Process next received packet
    }
}

```

**Figure 7.121**  
cont'd

The filter coefficients are

```

h[0] = h[10] = 0.0681
h[1] = h[9] = 0.0810
h[2] = h[8] = 0.0918
h[3] = h[7] = 0.1001
h[4] = h[6] = 0.1052
h[5] = 0.1070

```

The filter structure is shown in [Figure 7.127](#).

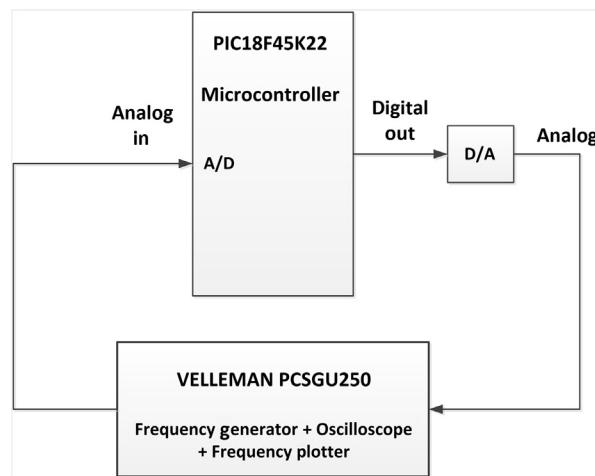


Figure 7.122: Block Diagram of the Project.



Figure 7.123: The Velleman PCSGU250 Device.

### The Hardware

The circuit diagram of the project is shown in [Figure 7.128](#). The MCP4921 12-bit serial DAC chip, controlled by the SPI bus, is used for the DAC.

### Project PDL

The project PDL is shown in [Figure 7.129](#).

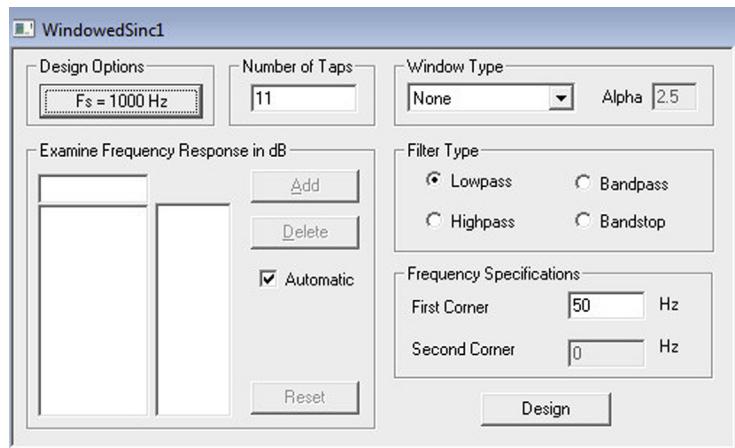


Figure 7.124: ScopeFIR Filter Design Program.

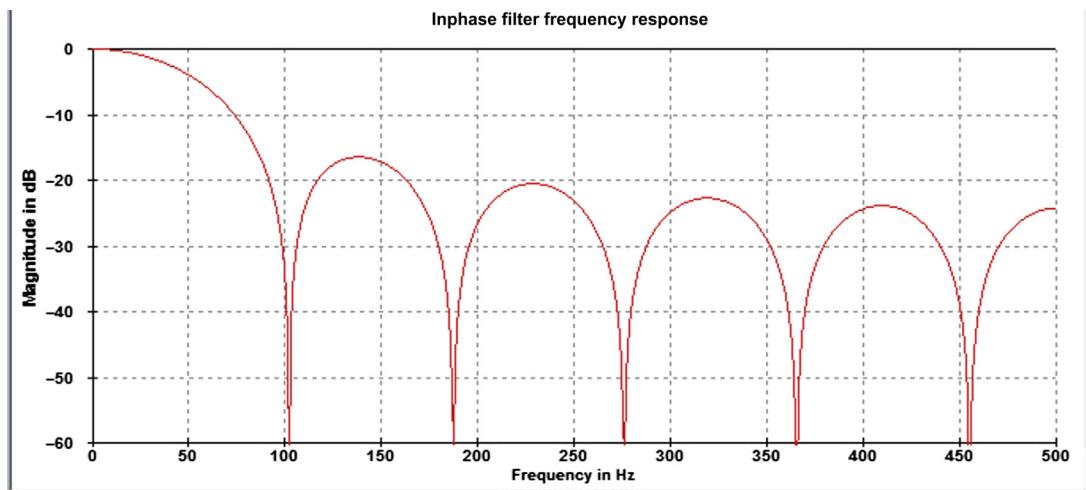


Figure 7.125: Frequency Response of the Filter.

### **Project Program**

#### *mikroC Pro for PIC*

The mikroC Pro for the PIC program listing is given in [Figure 7.130](#) (MIKROC-FIR.C). At the beginning of the program, the D/A converter chip select pin connection is defined. The 11 filter coefficients are then stored in a floating point array called *h*:

```
float h[N] = {0.0681, 0.0810, 0.0918, 0.1001, 0.1052, 0.1070, 0.1052, 0.1001,
0.0918, 0.0810, 0.0681};
```

```
0.068146550155253907  
0.081014025735235917  
0.091886195306041848  
0.100138842939668960  
0.105292210529990840  
0.107044350667617030  
0.105292210529990840  
0.100138842939668960  
0.091886195306041848  
0.081014025735235917  
0.068146550155253907
```

Figure 7.126: Filter Coefficients.

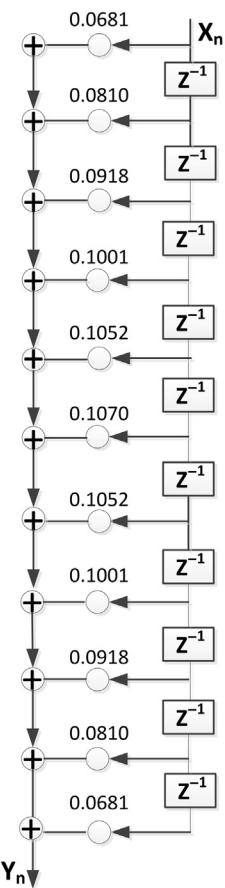


Figure 7.127: The 10th Order (11-tap) FIR Filter Structure.

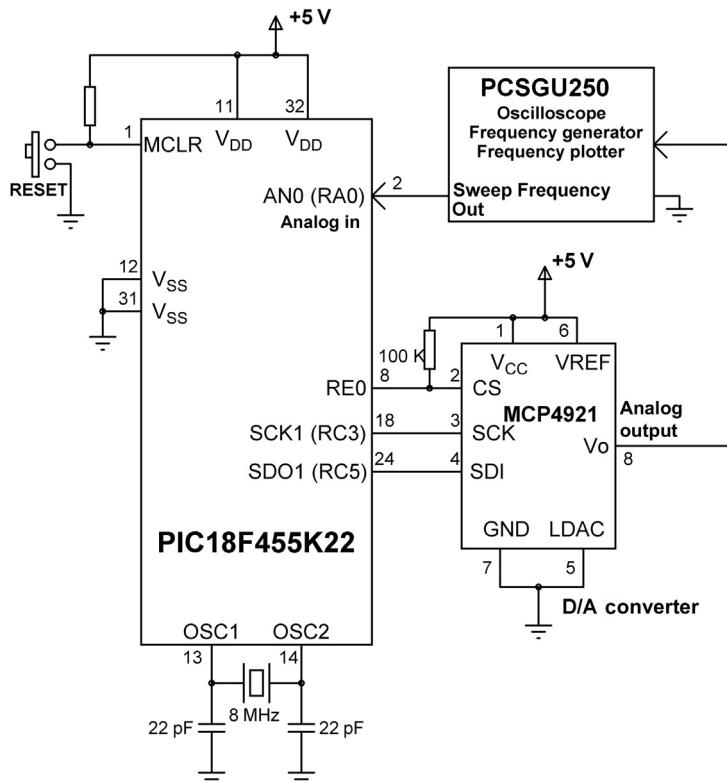


Figure 7.128: Circuit Diagram of the Project.

**BEGIN/MAIN**

- Configure the D/A chip select pin
- Store filter coefficients in an array
- Configure PORTC and PORTE as digital
- Configure AN0 (Channel 0) as analog input
- Initialize the SPI library
- Initialize A/D converter module
- Initialize Timer 1 for 1ms interrupts
- Wait for Timer 0 interrupts (TMR)

**END/MAIN****BEGIN/TMR**

- Get a new signal sample
- Calculate the output sample
- Send the output sample to the D/A converter
- Delay the input signals by one sample time
- Clear Timer 0 interrupt flag

**END/TMR**

Figure 7.129: Project PDL.

```
*****
    FINITE IMPULSE RESPONSE FILTER DESIGN
=====
```

This project shows how a FIR type digital filter can be designed.

Analog sine wave signal is fed to the AN0 (RA0) analog input of the microcontroller. A D/A converter is connected to the microcontroller through the SPI bus so that the filtered signal is in analog form and can be plotted using a frequency plotter.

In this example a LOW-Pass filter is designed with the following specifications:

Filter Type:	No windowing
Sampling Frequency:	1000 Hz
Passband Upper Frequency:	50 Hz
Filter Order:	10 (11 taps)

The FIR filter coefficients are obtained using the ScopeFIR software package.

The filter response is plotted using a Velleman PCGU250 type oscilloscope+frequency generator + frequency plotter device.

The PIC18F45K22 microcontroller is used in this project. The external clock is supplied using an 8 MHz crystal, but internally the PLL is used to increase the clock frequency to 32 MHz (this is done during the programming by enabling 4xPLL and setting the clock frequency to 32 MHz in Project -> Edit Project window).

The D/A converter used is the MCP4921 SPI bus based converter with 12-bit resolution, operating with +5 V reference voltage.

The connection between the microcontroller and the D/A converter is as follows:

```
Author:      Dogan Ibrahim
Date:       October 2013
File:        MIKROC-FIR.C
*****
// DAC module connections
sbit Chip_Select at LATE0_bit;
sbit Chip_Select_Direction at TRISE0_bit;
// End DAC module connections

#define N 10                                // Filter order=10, having 11 taps

float Sample,xn, yn, x[N];
unsigned ADC;
unsigned char temp;
unsigned int DAC;
float h[N+1] = {0.0681, 0.0810, 0.0918, 0.1001, 0.1052, 0.1070, 0.1052, 0.1001, 0.0918, 0.0810,
0.0681};
//
// Timer 1 interrupt service routine. The program jumps here every 1000us (the sampling
```

**Figure 7.130: mikroC Pro for PIC Program.**

```

// frequency is 1 kHz, i.e. Period = 1ms = 1000us). Here, a new output is calculated and sent
// to the D/A converter
//
void interrupt()
{
    unsigned char i;

    TMROL = 6;                                // Re-load TMRO
    ADC = ADC_Get_Sample(0);                   // Get new input Sample from AN0
    x[0]=ADC;
    yn = 0.0;

    //
    // Calculate a new output yn
    //
    for(i = 0; i <= N; i++)
    {
        yn = yn + h[i]*x[i];
    }
    //
    // Output the new Sample via the D/A converter
    //
    DAC = yn;
    Chip_Select = 0;                          // Select DAC chip

    // Send High Byte
    temp = (DAC >> 8) & 0x0F;                // Store DAC[11..8] to temp[3..0]
    temp |= 0x30;                            // Define D/A setting
    SPI1_Write(temp);                        // Send high byte via SPI

    // Send Low Byte
    temp = DAC;                             // Store DAC[7..0] to temp[7..0]
    SPI1_Write(temp);                        // Send low byte via SPI

    Chip_Select = 1;                         // Deselect D/A converter chip
    //
    // Shift the input samples for the delay action
    //
    for(i = 0; i < N; i++)
    {
        x[N-i] = x[N-i-1];
    }
    //
    // Re-enable Timer 0 interrupts
    //
    INTCON.TMROIF = 0;                      // Clear Timer 0 interrupt flag
}

//
//***** MAIN PROGRAM *****
// Start of MAIN program. In the main program the I/O ports are configured, SPI bus

```

**Figure 7.130**  
cont'd

```

// library is initialized, and the A/D converter module library is initialized. In addition,
// Timer 0 is configured to interrupt at every one millisecond and interrupts are enabled.
//
void main()
{
    TRISA0_bit = 1;                                // AN0 (RA0) is input
    ANSELE = 0;                                     // RE0 is digital I/O
    ANSELA = 1;                                     // RA0 is analog I/O
    Chip_Select_Direction = 0;                      // Configure CS pin as output
    Chip_Select = 1;                                 // Disable D/A converter
    SPI1_Init();                                    // Initialize SPI1
    ADC_Init();                                     // Initialize A/D converter

    //
    // Configure Timer 0 for 1000us (1ms) interrupts
    //
    TOCON = 0x44;                                   // Disable TMRO, 8-bit, prescaler-32
    INTCON.TMROIF = 0;                             // Clear Timer 0 interrupt flag
    INTCON.TMROIE = 1;                            // Enable Timer 0 interrupts
    TMROL = 6;                                      // Load TMRO
    TOCON.TMROON = 1;                            // Enable Timer 0
    INTCON.GIE = 1;                                // Enable global interrupts

    for(;;)                                         // Wait for Timer 0 interrupts
    {
    }
}

```

**Figure 7.130**  
cont'd

Inside the main program, analog input AN0 (Channel 0, or pin RA0) is configured as an analog input, D/A converter is disabled, and the SPI library and A/D converter modules are initialized. Timer 0 is then configured to interrupt at every sampling time (1000  $\mu$ s).

With a clock frequency of 32 MHz, the clock period is 0.03125  $\mu$ s (note that the actual crystal frequency is 8 MHz, but the internal PLL module is used to multiply the external clock frequency by 4 to give an operating clock frequency of 32 MHz. This is done by enabling the 4xPLL option in the Project → Edit Project window before the microcontroller is programmed. In addition, the oscillator frequency should be selected as 32 MHz in this window). Since the instruction cycle time is four clock periods, the actual timer clock frequency is 8 MHz, or the actual timer clock period is 0.125  $\mu$ s. The timer prescaler is set to 32, giving a value of 6 for the timer register TMROL. Thus, when Timer 0 is loaded with 6 and timer and global interrupts are enabled, the timer will generate interrupts at every millisecond and the program will jump to the ISR declared by the programmer:

$$\text{TMROL} = 256 - \text{Delay}/(\text{Clock period} \times \text{prescaler value})$$

or,

$$\text{TMROL} = 256 - 1000 \mu\text{s}/(0.125 \mu\text{s} \times 32) = 6$$

The main program then enters a loop and waits for timer interrupts to occur. The digital filtering operation is performed inside the timer ISR, which is entered every millisecond. Here, a new input sample is obtained from analog channel AN0 by calling function *ADC\_Get\_Sample* with channel number 0, and the output sample is calculated.

The program then sends the output sample to the D/A converter. The D/A converter is 12 bits wide and the high nibble (bits 8–11) is sent first, followed by the low byte (bits 0–7). Function *SPI1\_Write* sends data to the D/A converter over the SPI bus. Note that the PIC18F45K22 microcontroller supports two SPI bus I/O pins, and in this project, only the first SPI bus is used. This is why the SPI statements are terminated with number 1. The following statements are used to send the processed data to the D/A converter:

```
Chip_Select = 0;           // Enable DAC chip
// Send High Byte
temp = (DAC >> 8) & 0x0F; // Store DAC[11..8] to temp[3..0]
temp |= 0x30;             // Define D/A setting
SPI1_Write(temp);         // Send high byte via SPI
// Send Low Byte
temp = DAC;               // Store DAC[7..0] to temp[7..0]
SPI1_Write(temp);         // Send low byte via SPI
Chip_Select = 1;           // Disable D/A chip
```

The input samples are then shifted (delayed) by one sampling time using the following code:

```
for(i = 0; i < N; i++)
{
    x[N-i] = x[N - i - 1];
}
```

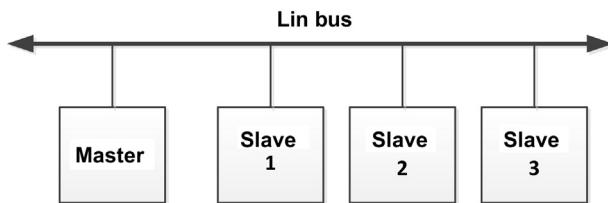
Just before exiting from the ISR, the timer interrupt flag TMR0IF is cleared so that further timer interrupts can be accepted by the processor.

Note that the A/D and D/A converters used in the design are unipolar (accept only positive voltages), and therefore, it is necessary to introduce the DC level to the input signal to shift it up so that it is always positive. This is done by clicking the *Offset* button at the bottom right-hand corner of the PCSGU250 screen.

In addition, the PCSGU250 device will not give an accurate graph in the stop band if the input signal is very small as may be the case in the stop band of low-pass filters.

### ***Project 7.16—Automotive Project—Local Interconnect Network Bus Project***

The Local Interconnect Network (LIN) is a serial network protocol developed for the automotive industry. The CAN bus was too expensive to implement for every electronic component in a car and the need for a cheap serial network arose as a result of this.



**Figure 7.131: LIN Bus with One Master and Three Slaves.**

The LIN bus is a low-cost serial protocol that can easily be implemented with microcontrollers having UART modules. The bus consists of a master and typically up to 16 slaves. All messages are initiated by the master with an identifier. The slave with the matching identifier replies to the message. As a result of this two-way communication, there is no collision on the bus. In typical applications, the master is a microcontroller requesting information or sending commands to a slave. The slaves are typically sensors or actuators that respond to the commands sent by the master.

Compared to the CAN bus, the LIB bus has the following advantages and disadvantages:

- The LIN bus has maximum data rate of 19,200 bps, while the CAN bus is much faster, up to 1 Mbps.
- The LIN bus is limited to 16 nodes (1 master and up to 15 slaves). The CAN bus can have up to 128 nodes.
- The LIN can is a single-wire bus (plus the chassis), while the CAN bus is a two-wire bus.
- The LIN bus is much cheaper to implement than the CAN bus.

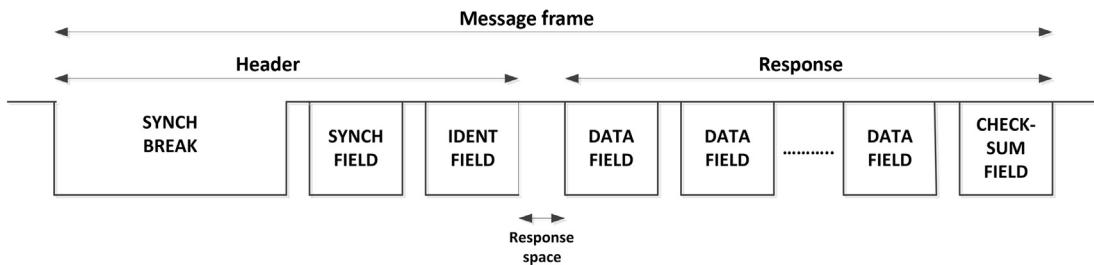
The LIN bus can be up to 40 m long and is typically used in the following components of vehicles:

- Vehicle doors, central locking system, mirrors.
- Vehicle light sensor, sun roof, light control.
- Seat heater, occupancy sensor, seat motors.
- Small engine controls, such as cruise control, wiper, turn indicator, climate control, small motors, sensors, steering wheel.

[Figure 7.131](#) shows a typical LIN bus implementation with one master and three slaves.

### ***The LIN Protocol***

The LIN protocol can be implemented with any microcontroller that supports a UART module. The protocol consists of frames, where each frame has two



**Figure 7.132: LIN Protocol.**

parts: the Header, and the Response. The protocol consists of the following fields ([Figure 7.132](#)):

- Synch Break,
- Synch Field,
- Identifier Field,
- Data Field,
- Checksum Field.

Each byte, except the Synch Break, is transmitted (or received) in standard serial format with one start bit, eight data bits, and one stop bit.

**Synch Break:** The Synch Break is always initiated by the master, and it signifies the start of a frame. It is identified by a start bit, and at least 13 bits of 0s, followed by the stop bit.

**Synch Field:** This field is sent by the master as the Synch delimiter, and it allows the slaves to synchronize. The data sent are one start bit, eight data bits corresponding to hexadecimal number 0x55 (bit pattern 01010101), and one stop bit.

**Identifier Field:** This byte is sent by the master, and it consists of the start bit, eight identifier bits, and one stop bit in the following format:

- Bits 0 to 3—LIN ID,
- Bits 4 and 5—Data Length,
- Bits 6 and 7—Parity.

ID0	ID1	ID2	ID3	ID4	ID5	P0	P1
-----	-----	-----	-----	-----	-----	----	----

The LIN ID bits (ID0 to ID3) represent the identifier of the slave node who is to respond in the Response part of the frame.

The Data Length bits specify the number of bytes in the Data Field:

ID5	ID4	No of Bytes
0	0	2
0	1	2
1	0	4
1	1	8

The last 2 bits of the identifier Field are parity bits that are used to detect possible errors (there is no error correction). The parity is calculated using the following algorithm:

$$P_0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4$$

$$P_1 = ID1 \oplus ID3 \oplus ID4 \oplus ID5$$

Identifiers in the range 0x00 to 0x3B are known as Unconditional frame identifiers, and there is only one sender of these frames. The identifiers, known as Command Frame Identifiers and having codes 0x60, 0x3C, and 0x3D are known as Diagnostic Frames and are reserved for diagnostic purposes. The Command Frame with the first byte set to 0x00 is used to put all slaves into the Sleep mode.

Data Field: the Data Field can contain 2, 4, or 8 bytes, each having one start bit, eight data bits, and one stop bit.

Checksum Field: This is the last byte in a frame. This byte contains the inverted modulo – 256 sum of all bytes within the Data Field. The sum is calculated by adding all data bytes with any carry bits and then inverting the answer (the property of inverted modulo-256 sum is that if the resultant number is added to the sum of all data bytes, the result will be 0xFF). For example,  $0xFF + 0x01 = 0x01$  and not 0x00.

### **Project Description**

In this project, a master and a slave node are used. The slave node is connected to a temperature sensor. Temperature readings are sent to the master on request and are displayed on an LCD connected to the master.

[Figure 7.133](#) shows the block diagram of the project.

### **Project Hardware**

The circuit diagram of the project is shown in [Figure 7.134](#). The MCP 201 LIN bus transceiver chip is used for the LIN bus to microcontroller interface. An LM35DZ-type temperature sensor is connected to the slave node.

The MCP 201 chip has the following features:

- Support up to 19,200 bps communication speed;
- Six- to 18-V supply voltage;

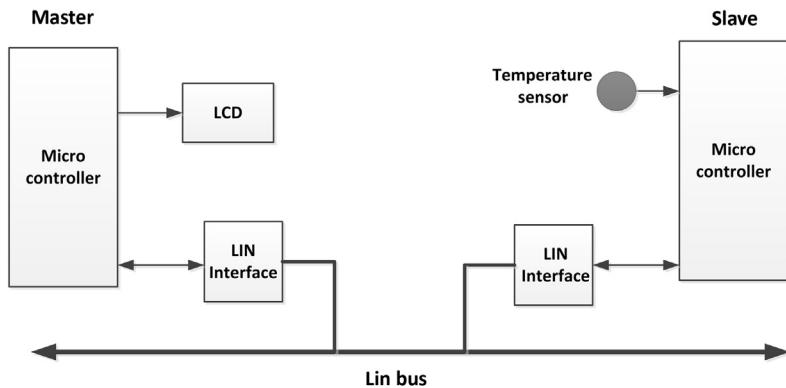


Figure 7.133: Block Diagram of the Project.

- Eight-pin DIL housing;
- Standard UART interface;
- Internal pull-up resistor and diode;
- A 40-mA current drive;
- Short-circuit current limit;
- Internal 5 V, 50-mA regulator.

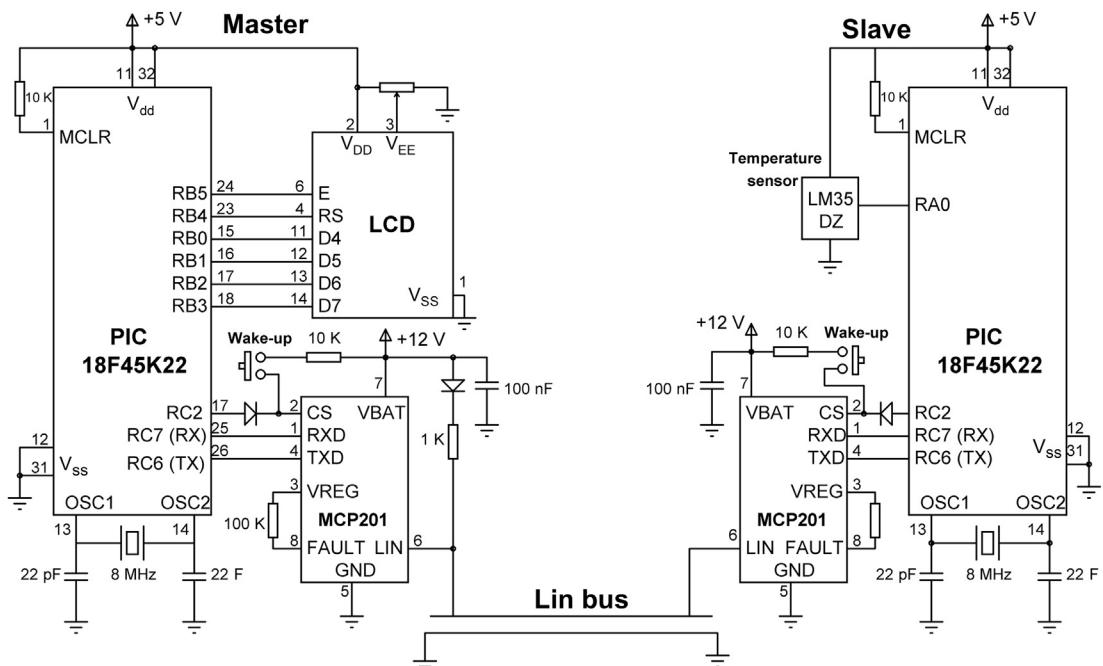


Figure 7.134: Circuit Diagram of the Project.

The MCP 201 provides half-duplex, bidirectional communications interface between a microcontroller and the LIN bus. The device translates the microcontroller logic levels to LIN logic, and vice versa.

The MCP 201 has the following pin definitions:

Pin 1, RXD: Receive data output.

Pin 2, CS/WAKE: Chip select (logic 1 to activate chip).

Pin 3, VREG: +5-V output.

Pin 4, TXD: Transmit data output.

Pin 5, VSS: Ground.

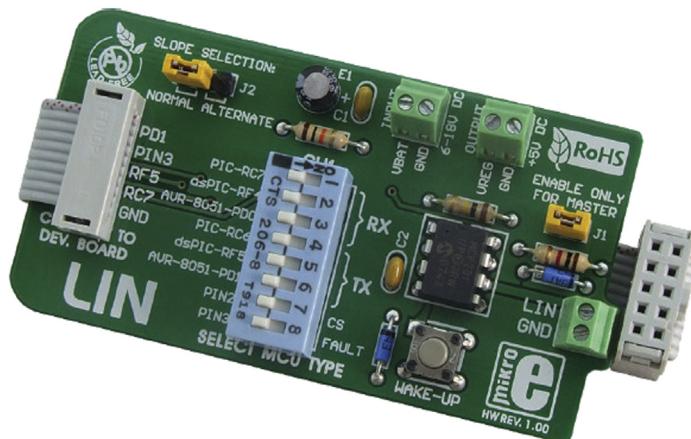
Pin 6, LIN: Lin bus connection.

Pin 7, VBAT: 6- to 18-V input.

Pin 8, FAULT: Fault detect output.

Two EasyPIC V7 development boards and two mikroElektronika LIN Bus Boards were used in the project development. These LIN Bus boards ([Figure 7.135](#)) have the following features:

- Baud rates up to 19,200 bps;
- Supply voltage 6–18 V;
- Conforming to LIN bus standards;
- Compatible with mikroC Pro for PIC compiler;
- Compatible with the EasyPIC V7 development board.



**Figure 7.135:** mikroElektronika LIN Bus Board.

If you are using the LIN Bus board together with the EasyPIC V7 development board, then plug in the board to PORTC connector at the edge of the development board and set the following jumpers on both the MASTER and the SLAVE boards (these jumper settings connect the UART pins RC6 and RC7 to the board. Also, the CS is connected to the RC2 pin of the microcontroller):

- Set DIL switch SW1 1, 4, and 7 ON.
- Leave J1 on the MASTER node.
- Remove J1 from the SLAVE node.
- Apply external a 6- to 18-V DC supply to the VBAT connectors.
- Establish the LIN bus between the MASTER and SLAVE by connecting LIN and GND connectors of both boards together.

### ***Project PDL***

The project PDL is shown in [Figure 7.136](#).

### ***Project Program***

*mikroC Pro for PIC*

MASTER Node

The mikroC Pro for the PIC program listing of the MASTER node is shown in [Figure 7.137](#) (MIKROC-LINMSTR.C). At the beginning of the program, the connections between the LCD and the microcontroller are defined. Also, the CS connection of the MCP 201 chip is defined as pin RC2 of the microcontroller. Symbols SYNCH\_FIELD, SLAVE\_NODE, and No\_Of\_Bytes are assigned values.

Inside the main program, PORTB and PORTC are configured as digital, MCP 201 chip is disabled, LCD is initialized, and message “LIN BUS PROJECT” is displayed for 2 s. The remainder of the program is executed in an endless loop, formed using a while statement. Inside this loop, the HEADER is sent to the LIN bus using function Send\_Header, response is received from the slave using function Get\_Response, and the received data (temperature in this project) are displayed on the LCD.

#### **Send\_Header Function**

This function sends the BREAK sequence, SYNCH\_FIELD, and the IDENT FIELD. The BREAK sequence is accomplished by forcing the UART to generate a frame error (i.e. the

**MASTER:**

**Main Program**

**BEGIN**

- Define connections between the LCD and microcontroller
- Configure PORTB and PORTC as digital
- Initialize LCD
- Send a message to the LCD
- Initialize UART

**DO FOREVER**

- CALL** Send\_Header
- CALL** Get\_Response
- Clear LCD
- Display temperature
- Wait 1 second

**ENDDO**

**END**

**BEGIN/Send\_Header**

- Send BREAK sequence
- Send SYNCH FIELD
- Send IDENT FIELD

**END/Send\_Header**

**BEGIN/Get\_Response**

- Get response bytes from the slave

**END/Get\_Response**

**SLAVE:**

**Main Program**

**BEGIN**

- Define connections between the LCD and microcontroller
- Configure RA0 as analog and PORTC as digital
- Initialize UART

**DO FOREVER**

- Wait for BREAK sequence
- Get SYNC FIELD
- Get IDENT FIELD
- Extract identifier
- IF** the request is for this node
  - Read temperature from channel 0
  - Send data to MASTER over the LIN bus
  - CALL** Calc\_Checksum
  - Send Checksum to MASTER over the LIN bus

**ENDIF**

**ENDDO**

**END**

**Begin/Calc\_Checksum**

- Add all data bytes including carry bit
- Invert the sum
- Return the sum to the caller

**END/Calc\_Checksum**

Figure 7.136: The Project PDL

---

```
*****
LIN BUS CONTROL PROJECT
=====
```

This project shows how to use the Automotive LIN bus in microcontroller projects.

In this project 2 LIN bus nodes, named MASTER and SLAVE communicate with each other at 9600 bps over the LIN BUS.

An LCD is connected to the MASTER node. An LM35DZ type temperature sensor is connected to the SLAVE node. The MASTER node requests the temperature every second from the SLAVE node. The SLAVE node reads the temperature and sends to the MASTER node which then displays on the LCD.

This project is based on the EasyPIC V7 development board and the mikroElektronika LIN bus boards. Two development boards are used, one as the MASTER, the other one as the SLAVE. LIN bus boards are attached to PORTC of each development board and the two boards are connected to each other with two cables (LIN bus and ground). The CS pins of the LIN bus boards are connected to RC2 pin of the microcontroller.

There is one MASTER and one SLAVE in this project.

This is the MASTER node program. The program works with 8 MHz clock (clock PLL is disabled)

Author: Dogan Ibrahim  
 Date: October, 2013  
 File: MIKROC-LINMSTR.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 MCP201_CS PORTC.RC2           // MCP201 CS connection
#define SYNCH_FIELD 0x55              // SYNCH FIELD
#define SLAVE_NODE_ID 0x01            // Slave node identifier
#define No_Of_Bytes 2                 // Number of bytes to receive
//
// This function sends HEADER to the SLAVE. The Header consists of:
// SYNCH BREAK, SYNCH FIELD, IDENT FIELD
```

**Figure 7.137: mikroC Pro for PIC MASTER Program.**

```

//
void Send_Header()
{
    bit P0, P1, ID0, ID1, ID2, ID3, ID4, ID5;
    unsigned char IDENT_FIELD, temp, Cnt, c;

//
// Enable BREAK sequence
//
    TXSTA1.TXEN = 1;                                // Set TXEN bit
    TXSTA1.SENDB = 1;                                // Set SENDB bit
    TXREG1 = 0x0;                                    // send dummy data to start the sequence
    Uart1_Write(SYNCH_FIELD);                        // Send SYNCH FIELD character

//
// The Node Identifier is set to 1, and number of bytes is 2. Thus, the IDENT FIELD has the
// format P1 P0 00 0001. Find parity bits P1 and P0 and add to the IDENT field
    if(No_Of_Bytes == 2)                            // No of bytes
        Cnt = 0;                                    // 2-bit field for the number of bytes
    else if(No_Of_Bytes == 4)
        Cnt = 2;                                    // Cnt is 0,2, or 3 depending on byte count
    else if(No_Of_Bytes == 8)
        Cnt = 3;

    IDENT_FIELD = SLAVE_NODE_ID;
    Cnt = Cnt << 4;                                // Add No of Bytes to IDENT field

    ID0 = IDENT_FIELD.F0;                           // Bit 0
    ID1 = IDENT_FIELD.F1;                           // Bit 1
    ID2 = IDENT_FIELD.F2;                           // Bit 2
    ID3 = IDENT_FIELD.F3;                           // Bit 3
    ID4 = IDENT_FIELD.F4;                           // Bit 4
    ID5 = IDENT_FIELD.F5;                           // Bit 5
    P0 = ID0 ^ ID1 ^ ID2 ^ ID4;                   // Find P0
    P1 = ID1 ^ ID3 ^ ID4 ^ ID5;                   // Find P1

    Temp = 0;
    Temp = P1 | P0;
    Temp = Temp << 6;
    IDENT_FIELD = IDENT_FIELD | Temp;              // Add the parity bits to IDENT field
    while(Uart1_Tx_Idle() == 0);                  // Wait until ready
    Uart1_Write(IDENT_FIELD);                      // Send IDENT_FIELD
}

//
// This function reads data from the SLAVE node. The last byte received is the Checksum
//
void Get_Response(unsigned char c[])
{
    unsigned char i, Checksum;

```

**Figure 7.137**  
cont'd

```

for(i = 0; i < No_Of_Bytes; i++)
{
    while(Uart1_Data_Ready() == 0);           // Wait until UART is ready
    c[i] = Uart1_Read();                     // get data from UART
}
c[i] = 0x0;                                // Insert NULL terminator
while(Uart1_Data_Ready() == 0);               // Wait until UART has data
Checksum = Uart1_Read();                    // Read the Checksum
}

// Start of MAIN program
//
void main()
{
    unsigned char Txt[3];

    ANSELB = 0;                            // Configure PORTB as digital
    ANSELC = 0;                            // Configure PORTC as digital
    MCP201_CS = 0;                         // Disable MCP201 to stat with
    TRISC.RC2 = 0;                          // Configure RC2 as output

    Lcd_Init();                           // Initialize LCD
    Lcd_Cmd(_LCD_CURSOR_OFF);            // LCD cursor OFF
    Lcd_Cmd(_LCD_CLEAR);                // Clear LCD
    Lcd_Out(1,1,"LIN BUS PROJECT");     // Write message on row 1
    Delay_Ms(2000);                     // Wait to see the message

    Uart1_Init(9600);                   // Set UART baud rate to 9600bps
    MCP201_CS = 1;                      // Activate MCP201

    while(1)                            // Do FOREVER
    {
        Send_Header();                  // Send LIN bus Header over the LIN bus
        Get_Response(Txt);            // Get Response from the SLAVE
        Lcd_Cmd(_LCD_CLEAR);          // Clear LCD
        Lcd_Out(2,1,Txt);             // Display data (temperature)
        Delay_Ms(1000);               // Wait 1 s
    }
}

```

**Figure 7.137**  
cont'd

condition where the STOP bit is not received at the expected time). There are basically two ways that we can generate the BREAK sequence:

1. By sending a logic 0 (START bit) on pin RC6 and keeping the line low for at least 13 bit times, and then sending a STOP bit. For example, if the baud rate is 9600 bps, then

13 bits will take approximately 1.3 ms. The following statements will generate a BREAK sequence:

```
PORTC.RC6 = 0; // Send START bit  
Delay_Ms(2); // Wait for 2 ms  
PORTC.RC6 = 1; // send STOP bit
```

2. Alternatively, we can force the UART module to send a BREAK sequence. The extended UART, found in most PIC18F series microcontrollers, supports sending the BREAK sequence. The following statements force the UART to send a BREAK sequence:

```
TXSTA1.TXEN = 1; // Set TXEN bit  
TXSTA1.SENDB = 1; // Set send-break (SENDB) bit  
TXREG1 = 0x0; // Send dummy data to start the sequence
```

After sending the BREAK sequence, the SYNCH\_FIELD character (0x55) is sent to the slave over the bus. Next, the IDENT field is formed by combining the slave node identifier, number of data bytes expected, and the parity bits P1 and P0, formed by Exclusive-OR'ing the appropriate bits.

#### *Get\_Response Function*

This function reads the data bytes from the slave device, including the Checksum byte. Although the Checksum is received, it is not validated here for simplicity. The program finally displays the temperature on the LCD. The above process is repeated every second.

#### SLAVE Node

The mikroC Pro for the PIC program listing of the SLAVE node is shown in [Figure 7.138](#) (MIKROC-LINSLAVE.C). At the beginning of the main program, PORTC is configured as a digital input, and RA0 is configured as an analog input. The UART is initialized to 9600 bps and the MCP 201 chip is activated. The remainder of the program is executed in an endless loop formed using a while statement. Inside this loop, the program waits until the BREAK sequence is received. There are several ways that the BREAK sequence can be detected:

1. By starting a timer when the START bit is detected and stopping the timer when the STOP bit is detected. If the timer value is equal or greater than 13 bit times (1.3 ms at 9600 bps), then it is assumed that the BREAK sequence is received and terminated.
2. By looking for the START bit and then a STOP bit, and assuming that the BREAK sequence is terminated when the STOP bit is received. Since when a slave device is waiting the only communication on the bus is the BREAK sequence, this is perhaps the simplest method of detecting the BREAK sequence.
3. By checking the framing error bit of the UART (RCSTA1.FERR).

---

```
*****
LIN BUS CONTROL PROJECT
=====
```

This project shows how to use the Automotive LIN bus in microcontroller projects.

In this project 2 LIN bus nodes, named MASTER and SLAVE communicate with each other at 9600 bps over the LIN BUS.

An LCD is connected to the MASTER node. An LM35DZ type temperature sensor is connected to the SLAVE node. The MASTER node requests the temperature every second from the SLAVE node. The SLAVE node reads the temperature and sends to the MASTER node which then displays on the LCD.

This project is based on the easyPIC V7 development board and the mikroElektronika LIN bus boards. Two development boards are used, one as the MASTER, the other one as the SLAVE. LIN bus boards are attached to PORTC of each development board and the two boards are connected to each other with two cables (LIN bus and ground). The CS pins of the LIN bus boards are connected to RC2 pin of the microcontroller.

There is one MASTER and one SLAVE in this project.

This is the SLAVE node program. The program works with 8MHz clock (clock PLL is disabled)

Author: Dogan Ibrahim

Date: October, 2013

File: MIKROC-LINSLAVE.C

```
*****
#define MCP201_CS PORTC.RC2                                // MCP201 CS bit
#define SLAVE_NODE_ID 0x01                                  // Our Identifier
#define SYNCH_FIELD 0x55                                    // SYNCH FIELD

// This function waits if UART is busy and then writes a character
//
void Write_Uart(unsigned char c)
{
    while(Uart1_Tx_Idle() == 0);                          // Wait if UART is busy
    Uart1_Write(c);                                      // Write the character
}

// This function checks if a character is ready in UART and then reads it
//
unsigned char Read_Uart()
{
    unsigned char c;

    while(Uart1_Data_Ready() == 0);                      // Wait to receive from the MASTER
    c = Uart1_Read();
    return c;
```

**Figure 7.138: mikroC Pro for PIC SLAVE Program.**

```
}

// This function calculates the Checksum byte and returns to the calling program.
// The Checksum is calculated by adding all the data bytes (including carry bit)
// and then inverting the result
//
unsigned char Calc_Checksum(unsigned char N, unsigned char c[])
{
    unsigned char i, Checksum = 0;

    for(i = 0; i < N; i++)
    {
        Checksum = Checksum + c[i] - '0';                                // Add data bytes (not ASCII)
        if(Checksum > 255)Checksum++;                                     // Add carry (if any)
    }
    Checksum = ~Checksum;                                              // Invert the data
    return Checksum;
}

// Start of MAIN program
//
void main()
{
    unsigned char c, i, Checksum, No_Of_Bytes, ID, MYID, Txt[7];
    unsigned int temp, mV;

    ANSELA = 1;                                                        // RA0 is analog
    ANSELC = 0;                                                       // Configure PORTC as digital
    TRISA.RAO = 1;                                                    // RA0 is input
    MCP201_CS = 0;                                                   // Disable MCP201
    TRISC.RC2 = 0;                                                    // Configure RC2 as output

    Uart1_Init(9600);                                                 // Set UART baud rate to 9600 bps
    Delay_Ms(10);                                                     // Wait until UART is settled
    MCP201_CS = 1;                                                    // Activate MCP201

    //

    // START OF LOOP
    //
    while(1)                                                          // Do FOREVER
    {
        //
        // Wait to receive the BREAK sequence. The BREAK sequence is identified when Framing
        // error occurs
        //
        TRISC.RC7=1;                                                 // Configure RC7 as input
        while(PORTC.RC7 == 1);                                         // Wait for START bit
```

**Figure 7.138**  
cont'd

```

        while(PORTC.RC7 == 0);                                // Wait for STOP bit

        //
        // Receive the SYNCH_FIELD byte
        //
        c = Read_Uart();                                     // Read a character
        if(c == SYNCH_FIELD)                                 // If SYNCH FIELD
        {
            ID = Read_Uart();                               // Read the IDENT FIELD
            MYID = ID & 0x0F;                             // Extract ID nibble

            if(MYID == SLAVE_NODE_ID)                      // If this node is requested
            {
                No_Of_Bytes = ID & 0x30;                  // Extract No Of Bytes to send
                No_Of_Bytes = No_Of_Bytes >> 4;

                if(No_Of_Bytes == 0 || No_Of_Bytes == 1)
                    No_Of_Bytes = 2;
                if(No_Of_Bytes == 2)
                    No_Of_Bytes = 4;
                if(No_Of_Bytes == 3)
                    No_Of_Bytes = 8;

                temp = ADC_READ(0);                         // Read the analog temperature
                mV = temp*5;                                // In millivolts (approximate)
                mV = mV / 10;                               // Temperature in Degrees C
                IntToStr(mV, Txt);                         // Convert to string
                Ltrim(Txt);                               // Remove leading spaces

                for(i = 0; i < No_Of_Bytes; i++)           // Do for all requested bytes
                {
                    Write_Uart(Txt[i]);                   // Send temperature to MASTER
                }
                Checksum = Calc_Checksum(No_Of_Bytes, Txt); // Calculate the Checksum
                Write_Uart(Checksum);                     // Send the Checksum byte
            }
        }
    }
}

```

**Figure 7.138**  
cont'd

In this program, option 2 is used to detect the end of the BREAK sequence:

```

TRISC.RC7 = 1;          // Configure RC7 as input
while(PORTC.RC7 == 1);  // Wait for START bit
while(PORTC.RC7 == 0);  // Wait for STOP bit

```

The program then reads a byte and checks to make sure that the received byte is actually a SYNCH byte (0x55). Then, the IDENT FIELD is read and the program extracts the ID to check if the request is for this node. If so, the temperature is read from analog channel 0 (RA0, AN0) of the microcontroller, converted into a string, and sent to the MASTER

device over the LIN bus. Finally, the Checksum is calculated and sent over the bus. Although the Checksum is sent, it is not validated by the MASTER for simplicity. The program then waits for the next request.

### ***Project 7.17—Automotive Project—Can Bus Project***

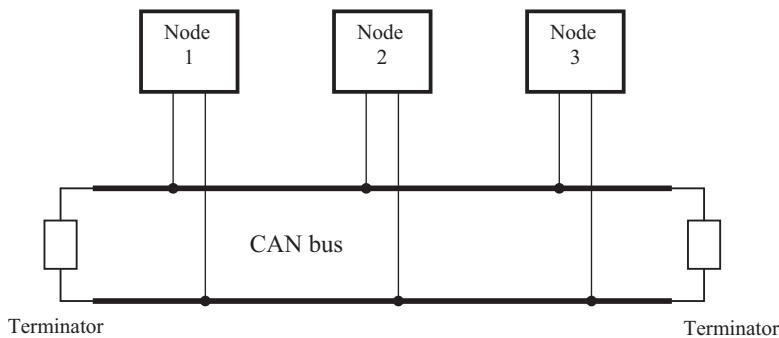
CAN is a serial bus communications protocol developed by Bosch (an electrical equipment manufacturer in Germany) in the early 1980s. Thereafter CAN was standardized in ISO-11898 and ISO-11519, establishing itself as the standard protocol for in-vehicle networking in car industry. CAN defines an efficient communication protocol between sensors, actuators, controllers, and other nodes in real time applications. The early CAN development was mainly supported by the vehicle industry, and it was used in passenger cars, boats, trucks, and other types of vehicles. Today, the CAN protocol is also used in many other fields requiring networked embedded control, such as industrial automation, medical applications, building automation, weaving machines, and production machinery.

CAN is widely accepted for its simplicity, high performance, and reliability. In the early days of the automotive industry various actuators and electromechanical subsystems were controlled using standalone, localized controllers. By networking all the electronics in vehicles, it became possible to control them from a central point, the engine control unit, and this has increased the functionality, added modularity, and made it possible to carry out diagnostics more efficiently.

CAN is based on a bus topology, and only two wires are needed for communication over the bus. The bus has a multimaster structure where each device on the bus is capable of sending or receiving data. Only one device can send data at any time while all the other devices listen. If two or more devices attempt to send data at the same time, then the device with the higher priority is allowed to send its data while the others return into the receive mode.

The use of CAN in the automotive industry has caused the mass production of CAN controllers. Today, it is estimated that >400 million CAN modules are sold every year, and CAN controllers are integrated on many microcontrollers (e.g. PIC microcontrollers) and are available at a low cost.

[Figure 7.139](#) shows a CAN bus with three nodes. The CAN protocol is based on CSMA/CD + AMP (Carrier Sense Multiple Access/Collision Detection with Arbitration on Message Priority) protocol, which is similar to the protocol used in an Ethernet LAN. When Ethernet detects a collision the sending nodes stop transmitting. They wait a random time before trying to send again. The CAN protocol solves the collision problem with the principle of arbitration where only the higher priority node is given the right to send its data.



**Figure 7.139: Example CAN Bus.**

There are basically two types of CAN protocols: standard CAN 2.0A and CAN 2.0B. CAN 2.0A is the earlier standard with 11 bits of identifier, while CAN 2.0B is the new extended standard with 29 bits of identifier. The 2.0B controllers are completely backward compatible with 2.0A controllers and can receive and transmit messages in either format. There are two types of 2.0A controllers: the first is capable of sending and receiving 2.0A messages only and reception of any 2.0B message will flag an error. The second type of 2.0A controller (known as 2.0B passive) can also send and receive 2.0A messages, but in addition, they will acknowledge receipt of 2.0B messages and then ignore them.

Some of the features of CAN protocol are as follows:

- CAN bus is a multimaster. When the bus is free, any device attached to the bus can start sending a message.
- CAN bus protocol is flexible. The devices connected to the bus have no addresses. This means that messages are not transmitted from one node to another node based on addresses. All the nodes in the system receive every message transmitted on the bus and it is up to each node in the system to decide whether the message received should be kept or discarded. A single message can be destined for one particular node to receive, or many nodes based on the way the system is designed. Another advantage of not having addresses is that, when a new device is added or an existing device is removed from the bus, there is no need to change any configuration data, that is, the bus is “hot pluggable”.
- Another feature of the CAN protocol is the ability for a node to request information from other nodes on the bus. This is called **Remote Transmit Request** (RTR). Thus, instead of waiting for information to be sent continuously by a node, a request can be sent to the node to get its information. For example, in a vehicle, the engine temperature is an important parameter. The system can be designed such that the temperature can be sent periodically over the bus. A more elegant solution would be to request the temperature whenever required. This approach would minimize the bus traffic while still maintaining the integrity of the network.

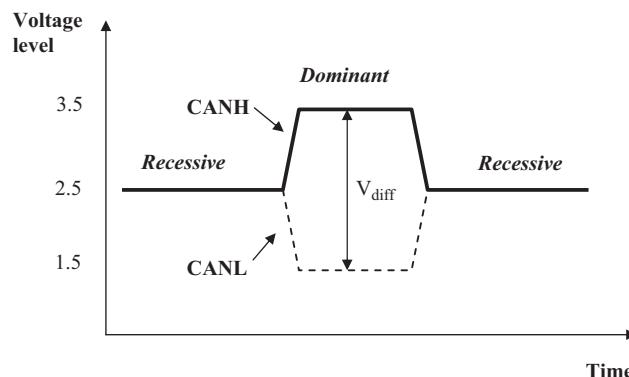


Figure 7.140: CAN Logic States.

- CAN bus communication speed is not fixed. Any communication speed can be set for the devices attached to a bus.
- All devices on the bus can detect an error. The device that has detected an error immediately notifies all other devices.
- Multiple devices can be connected to the bus at the same time. There are no logical limits to the number of devices that can be connected to the bus. In practice, the number of units that can be attached to the bus is limited by the delay time and electrical load of the bus.

The data on the CAN bus are differential, and there can be two states: **dominant** state and **recessive** state. Figure 7.140 shows the state of voltages on the bus. The bus defines a logic bit 0 as a dominant bit and a logic bit 1 as a recessive bit. When there is arbitration on the bus, a dominant bit state always wins arbitration over a recessive bit state. In the recessive state, the differential voltage CANH and CANL is less than the minimum threshold, that is,  $<0.5\text{-V receiver input and } <1.5\text{-V transmitter output}$ . In the dominant state, the differential voltage CANH and CANL is greater than the minimum threshold.

ISO-11898 specifies that a device on the CAN bus must be able to drive a 40-m cable at 1 Mb/s. A much longer bus length can usually be achieved by lowering the bus speed. A CAN bus is terminated to minimize signal reflections on the bus. ISO-11898 requires that the bus have a characteristic impedance of  $120\ \Omega$ . The bus can be terminated in one of the following methods:

- Standard termination,
- Split termination,
- Biased split termination.

In standard termination, a  $120\text{-}\Omega$  resistor is used at each end of the bus as shown in Figure 7.141(a), and this is the most commonly used termination method. In split

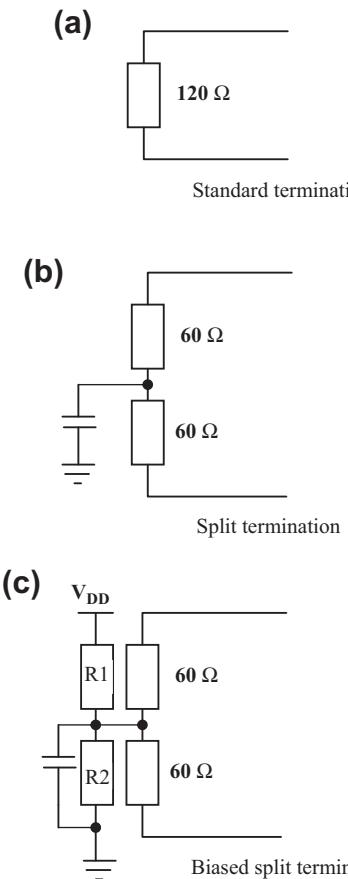


Figure 7.141: Bus Termination Methods.

termination, the ends of the bus is split and a single 60- $\Omega$  resistor is used as shown in Figure 7.141(b). Split termination provides reduced emission, and this method is gaining popularity. The biased split termination is similar to the split termination, but here, a voltage divider circuit and a capacitor are used at each end of the bus. This method increases the EMC performance of the bus (Figure 7.141(c)).

There are basically four message frames in CAN: **data**, **remote**, **error**, and **overload** frames. The data and remote frames need to be set by the user. Other frames are set by the CAN hardware.

### Data Frame

The data frame is in two formats: standard and extended. The standard format has a 11-bit ID, and the extended format has a 29-bit ID. The data frame is used by the

transmitting device to send data to the receiving device, and this is the most important frame handled by the user. A standard data frame starts with the SOF bit. It is then followed by an 11-bit identifier and the remote transmit request (RTR) bit. The identifier and the RTR form the arbitration field. The control field is 6 bits wide and it indicates how many bytes of data there are in the data field. The data field can be 0–8 bytes. The data field is followed by the CRC field, which checks whether or not the received bit sequence is corrupted. The ACK field is 2 bits, and it is used by the transmitter to receive an acknowledgement of a valid frame from any receiver. The end of the message is indicated by a 7-bit end-of-frame (EOF) field. In an extended data frame, the arbitration field is 29 bits wide.

The data frame consists of the following fields:

#### *Start of Frame*

This field indicates the beginning of a data frame and is common to both standard and extended formats.

#### *Arbitration Field*

Arbitration is used to resolve bus conflicts when more than one device starts sending a message on the bus. This field indicates the priority of a frame and differs between the standard and extended formats. In the standard format, there are 11 bits, and up to 2032 IDs can be set. The extended format ID consists of 11 base IDs and 18 extended IDs. Up to  $2032 \times 2^{18}$  discrete IDs can be set.

During the arbitration phase, each transmitting device transmits its identifier and compares it with the level on the bus. If the levels are equal, the device continues to transmit. If the device detects a dominant level on the bus while it was trying to transmit a recessive level, then it quits transmitting and becomes a receiving device. When the arbitration is over, there is only one transmitter left on the bus, and this transmitter continues to control field, data field etc.

#### *Control Field*

The control field is 6 bits wide, and it indicates the number of data bytes in a message to be transmitted. This field consists of two reserved bits and four data length code (DLC) bits. The control field is coded as shown in [Table 7.19](#), where up to eight transmit bytes can be coded with 6 bits.

#### *Data Field*

This field indicates the actual content of data. The data size can vary from 0 to 8 bytes. The data are transmitted with the MSB first.

**Table 7.19: Coding the Control Field.**

No of Data Bytes	DLC3	DLC2	DLC1	DLC0
0	D	D	D	D
1	D	D	D	R
2	D	D	R	D
3	D	D	R	R
4	D	R	D	D
5	D	R	D	R
6	D	R	R	D
7	D	R	R	R
8	R	D or R	D or R	D or R

D: Dominant level, R: Recessive level.

### CRC Field

The CRC field is used to check the frame for a transmission error. This field consists of a 15-bit CRC sequence and a 1-bit CRC delimiter. The CRC calculation includes the SOF, arbitration field, control field, and data field. The calculated CRC and the received CRC sequence are compared, and if they do not match, an error is assumed.

### ACK Field

The ACK field indicates that the frame has been received normally. This field consists of 2 bits, one for ACK slot and one for ACK delimiter.

### Remote Frame

This frame is used by the receive unit to request transmission of a message from the transmitting unit. The remote frame consists of six fields: **start of frame**, **arbitration field**, **control field**, **CRC field**, **ACK field**, and **end of frame field**. The remote field is the same as a data frame except that it does not have a data field.

### Error Frame

The error frame is used to notify an error that has occurred during transmission. This field consists of an **error flag** and an **error delimiter**. Error frames are generated and transmitted by the CAN hardware. There are two types of error flags: active error flag, and passive error flag. Active error flag consists of six dominant bits. Passive error flag consists of six recessive bits. The error delimiter consists of eight recessive bits.

### Overload Frame

The overload frame is used by the receive unit to notify that it is not ready to receive frames yet. This frame consists of an **overload flag** and an **overload delimiter**. The

overload flag consists of six dominant bits, and it is structured the same way as the active error flag of the error frame. The overload delimiter consists of eight recessive bits and this field is structured the same way as the error delimiter of the error frame.

### **Bit Stuffing**

CAN bus uses bit stuffing technique that is used to periodically synchronize transmit–receive operations to prevent timing errors between receive nodes. If five consecutive bits with the same level appear, 1 bit of inverted data is added to the sequence. During sending of a data or a remote frame, the same level occurs in five consecutive bits during the SOF to CRC sequence, an inverted level preceding 5 bits is inserted in the next (i.e. the sixth) bit. During receiving a data or a remote frame, the same level occurs in five consecutive bits during the SOF to CRC sequence, the next (sixth) bit is deleted from the received frame. If the deleted sixth bit is at the same level as the preceding fifth bit, an error (stuffing error) is detected.

### **Nominal Bit Timing**

The CAN bus nominal bit rate (NMR) is defined as the number of bits transmitted every second without resynchronization. The inverse of the NMR is the nominal bit time. All devices on the CAN bus must use the same bit rate, even though each device can have its own different clock frequency. One message bit consists of four nonoverlapping time segments:

- Synchronization segment (Sync\_Seg),
- Propagation time segment (Prop\_Seg),
- Phase buffer segment 1 (Phase\_Seg1),
- Phase buffer segment 2 (Phase\_Seg2).

The **Sync\_Seg** segment is used to synchronize various nodes on the bus, and an edge is expected to lie within this segment. The **Prop\_Seg** segment compensates for the physical delay times within the network. The **Phase\_Seg1** and **Phase\_Seg2** segments are used to compensate for edge phase errors. These segments can be lengthened or shortened by synchronization. The sample point is the point in time where the actual bit value is located. The sample point is at the end of **Phase\_Seg1**. A CAN controller can be configured to sample three times and use a majority function to determine the actual bit value.

Each segment is divided into units known as **Time Quantum**, or  $T_Q$ . A desired bit timing can be set by adjusting the number of  $T_Q$ 's that comprise one message bit and the number of  $T_Q$ 's that comprise each segment in it. The  $T_Q$  is a fixed unit derived from the oscillator

period and the time quantum of each segment can vary from 1 to 8. The length of each time segment is

- **Sync\_Seg** is one time quantum long.
- **Prop\_Seg** is programmable to be one to eight time quanta long.
- **Phase\_Seg1** is programmable to be one to eight time quanta long.
- **Phase\_Seg2** is programmable to be two to eight time quanta long.

By setting the bit timing, it is possible to set a sampling point so that multiple units on the bus can sample messages with the same timing.

The nominal bit time is programmable from a minimum of eight time quanta to a maximum of 25 time quanta. By definition, the minimum nominal bit time is  $1 \mu\text{s}$  corresponding to a maximum 1 Mb/s rate. The nominal bit time ( $T_{\text{BIT}}$ ) is given by

$$T_{\text{BIT}} = T_Q * (\text{Sync\_Seg} + \text{Prop\_Seg} + \text{Phase\_Seg1} + \text{Phase\_Seg2}) \quad (7.1)$$

and the NMR is

$$\text{NBR} = 1/T_{\text{BIT}}.$$

The time quantum is derived from the oscillator frequency and the programmable baud rate prescaler, with integer values from 1 to 64. The time quantum can be expressed as

$$T_Q = 2 * (\text{BRP} + 1)/F_{\text{OSC}},$$

Where

$T_Q$  is in microseconds, and  $F_{\text{OSC}}$  is in megahertz, and BRP is the baud rate prescaler (0–63).

We can also write

$$T_Q = 2 * (\text{BRP} + 1) * T_{\text{OSC}},$$

Where  $T_{\text{OSC}}$  is in microseconds.

An example is given below for the calculation of the NMR.

### Example 7.1

Assuming a clock frequency of 20 MHz, a baud rate prescaler value of 1, and a nominal bit time of  $T_{\text{BIT}} = 8 * T_Q$ , determine the NMR.

#### Solution 7.1

From the above equations,

$$T_Q = 2 * (1 + 1)/20 = 0.2 \mu\text{s}.$$

Also,

$$T_{BIT} = 8 * T_Q = 8 * 0.2 = 1.6 \mu s$$

and

$$NBR = 1/T_{BIT} = 1/1.6 \mu s = 625,000 \text{ bits/s or } 625 \text{ Kb/s.}$$

To compensate for phase shifts between the oscillator frequencies of the nodes on the bus, each CAN controller must synchronize to the relevant signal edge of the received signal.

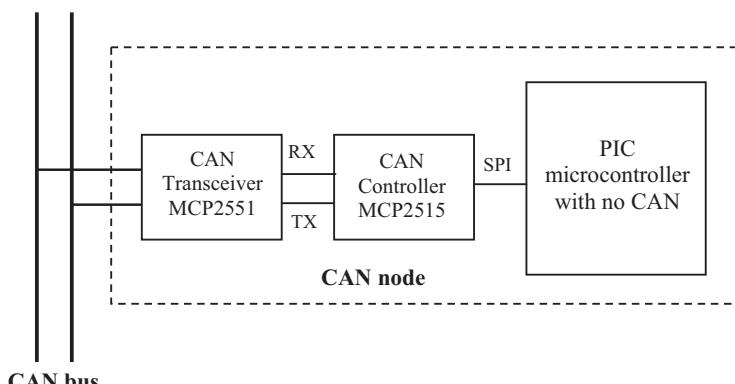
Two types of synchronization are defined: *hard synchronization* and *resynchronization*. Hard synchronization is done only at the beginning of a message frame, when each CAN node aligns the **Sync\_Seg** of its current bit time to the recessive or dominant edge of the transmitted SOF. According to the rules of synchronization, if a hard synchronization occurs, there will not be a resynchronization within that bit time.

With resynchronization, the **Phase\_Seg1** may be lengthened or **Phase\_Seg2** may be shortened. The amount of change of the phase buffer segments has an upper bound given by the **Synchronization Jump Width** (SJW). The SJW is programmable between 1 and 4, and its value is added to **Phase\_Seg1**, or subtracted from **Phase\_Seg2**.

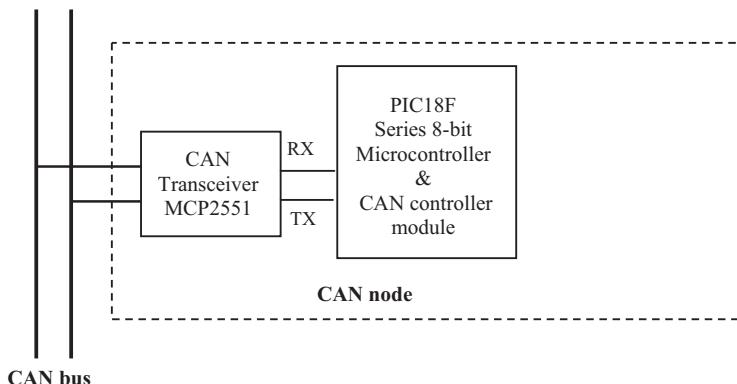
### **PIC Microcontroller CAN Interface**

PIC microcontrollers can be used in CAN bus-based projects. In general, any type of PIC microcontroller can be used, but some PIC microcontrollers (e.g. PIC18F258) have built-in CAN modules that simplify the design of CAN bus-based systems. Microcontrollers with no built-in CAN modules can also be used in CAN bus applications, but additional hardware and software are required, making the design costly and also more complex.

Figure 7.142 shows the block diagram of a PIC microcontroller-based CAN bus application. Here, a PIC microcontroller with no built-in CAN module is used.



**Figure 7.142: CAN Node with Any PIC Microcontroller.**



**Figure 7.143: CAN Node with an Integrated CAN Module.**

The microcontroller is connected to the CAN bus using an external MCP2515 CAN controller chip and an MCP2551 CAN bus transceiver chip. This configuration is suitable for a quick upgrade to an existing design using any PIC microcontroller.

For new CAN bus-based designs, it is easier to use a PIC microcontroller with a built-in CAN module. As shown in [Figure 7.143](#), such devices include built-in CAN controller hardware on the chip. All that is required to make a CAN node is then to add a CAN transceiver chip.

In this project, the PIC18F258 microcontroller is used in a CAN bus-based project. The description and operating principles given in this section are in general applicable to other PIC microcontrollers with CAN modules.

### **PIC18F258 Microcontroller**

PIC18F258 is a high performance 8-bit microcontroller with an integrated CAN module. The device has the following features:

- A 32 k flash program memory,
- A 1536-byte RAM data memory,
- A 256-byte EEPROM memory,
- Twenty two I/O ports,
- Five-channel 10-bit A/D converters,
- Three timers/counters,
- Three external interrupt pins,
- High current (25-mA) sink/source,
- Capture/compare/Pulse Width Modulation (PWM) module,
- SPI/I<sup>2</sup>C module,

- CAN 2.0A/B module,
- Power-on reset and power-on timer,
- Watchdog timer,
- Priority level interrupts,
- DC to 40-MHz clock input,
- An  $8 \times 8$  hardware multiplier,
- Wide operating voltage (2.0–5.5 V),
- Power saving sleep mode.

It is important to understand the architecture of the CAN module. In this section, we shall be looking at the CAN module features of the PIC18F258 microcontroller.

PIC18F258 microcontroller CAN module has the following features:

- Compatible with CAN 1.2, CAN 2.0A and CAN 2.0B,
- Supports standard and extended data frames,
- Programmable bit rate up to 1 Mbit/s,
- Double buffered receiver,
- Three transmit buffers,
- Two receive buffers,
- Programmable clock source,
- Six acceptance filters,
- Two acceptance filter masks,
- Loop-back mode for self-test,
- Low-power sleep mode,
- Interrupt capabilities.

The CAN module uses port pins RB3/CANRX and RB2/CANTX for CAN bus receive and transmit functions, respectively. These pins are connected to the CAN bus via an MCP2551 type CAN bus transceiver chip.

PIC18F258 microcontroller supports the following frame types:

- Standard data frame,
- Extended data frame,
- Remote frame,
- Error frame,
- Overload frame,
- Interframe space.

A node uses filters to decide whether or not to accept a received message. Message filtering is applied to the whole identifier field, and mask registers are used to specify which bits in the identifier are to be examined with the filters.

The CAN module in the PIC18F258 microcontroller has six modes of operation:

- Configuration mode,
- Disable mode,
- Normal operation mode,
- Listen-only mode,
- Loop-back mode,
- Error recognition mode.

### ***Configuration Mode***

The CAN module is initialized in the configuration mode. The module is not allowed to enter the configuration mode while a transmission is taking place. In the configuration mode, the module will not transmit or receive, the error counters are cleared, and the interrupt flags remain unchanged.

### ***Disable Mode***

In the disable mode, the module will not transmit or receive. In this mode, the internal clock is stopped unless the module is active. If the module is active, the module will wait for 11 recessive bits on the CAN bus, detect that condition as an IDLE bus, and then accept the module disable command. The WAKIF interrupt (wake-up interrupt) is the only CAN module interrupt that is active in the disable mode.

### ***Normal Operation Mode***

This is the standard operating mode of the CAN module. In this mode, the module monitors all bus messages and generates acknowledge bits, error frames, etc. This is the only mode that will transmit messages.

### ***Listen-only Mode***

This mode allows the CAN module to receive messages, including messages with errors. This mode can be used to monitor the bus activities or for detecting the baud rate on the bus. For autobaud detection, there must be at least two other nodes communicating with each other. The baud rate can be determined by testing different values until valid messages are received. No messages can be transmitted in the listen-only mode.

### ***Loop-back Mode***

In this mode, messages can be directed from internal transmit buffers to receive buffers, without actually transmitting messages on the CAN bus. This mode can be used during system developing and testing.

### Error Recognition Mode

This mode is used to ignore all errors and receive all messages. In this mode, all messages, valid or invalid, are received and copied to the receive buffer.

### CAN Message Transmission

PIC18F258 microcontroller implements three dedicated transmit buffers: TXB0, TXB1, and TXB2. Pending transmittable messages are in a priority queue. Prior to sending the SOF, the priority of all buffers that are queued for transmission is compared. The transmit buffer with the highest priority will be sent first. If two buffers have the same priorities, the buffer with the highest buffer number will be sent first. There are four levels of priority.

### CAN Message Reception

Reception of a message is a more complex process. PIC18F258 microcontroller includes two receive buffers RXB0 and RXB1 with multiple acceptance filters for each (Figure 7.144). All received messages are assembled in Message Assembly Buffer (MAB). Once a message is received, regardless of the type of identifier and the number of data bytes received, the entire message is copied into MAB.

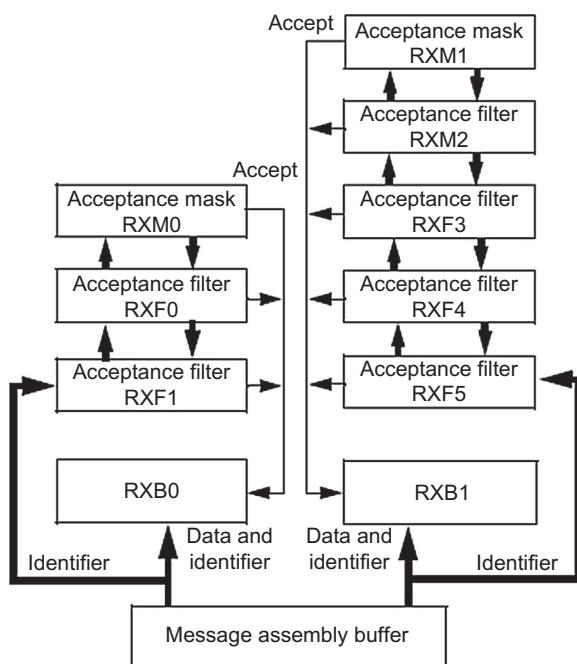


Figure 7.144: Receive Buffer Block Diagram.

Received messages have priorities. RXB0 is the higher priority buffer, and it has two message acceptance filters RXF0 and RXF1. RXB1 is the lower priority buffer, and it has four acceptance filters RXF2, RXF3, RXF4, and RXF5. There are also two programmable acceptance filter masks RXM0 and RXM1 available, one for each receive buffer.

The CAN module uses message acceptance filters and masks to determine if a message in the MAB should be loaded into a receive buffer. Once a valid message is received by the MAB, the identifier field of the message is compared to the filter values. If there is a match, that message will be loaded into the appropriate receive buffer. The filter masks are used to determine which bits in the identifier are examined with the filters.

### ***Calculating the Timing Parameters***

Setting the timing parameters of the nodes are very important for the reliable operation of the bus. Given the microcontroller clock frequency and the required CAN bus bit rate, we need to calculate the values of the following timing parameters:

- Baud rate prescaler value,
- Prop\_Seg value,
- Phase\_Seg1 value,
- Phase\_Seg2 value,
- SJW value.

Correct timing requires that

- Prop\_Seg + Phase\_Seg1  $\geq$  Phase\_Seg2,
- Phase\_Seg2  $\geq$  SJW.

An example is given below to illustrate how the various timing parameters can be calculated.

### ***Example 7.2***

Assuming that the microcontroller oscillator clock rate is 20 MHz, and the required CAN bit rate is 125 kHz, calculate the timing parameters.

#### **Solution 7.2**

With a 20-MHz clock rate, the clock period is 50 ns. Choosing a baud rate prescaler value of 4, from [Eqn \(7.4\)](#),

$$T_Q = 2 * (BRP + 1) * T_{OSC}$$

gives a time quantum of  $T_Q = 500$  ns. To obtain a NMR of 125 kHz, the nominal bit time must be

$$T_{BIT} = 1 / 0.125 \text{ MHz} = 8 \mu\text{s, or } 16T_Q.$$

The **Sync\_Segment** is  $1T_Q$ . Choosing  $2T_Q$  for the **Prop\_Seg**, and  $7T_Q$  for **Phase\_Seg1** leaves  $6T_Q$  for **Phase\_Seg2** and places the sampling point at  $10T_Q$  (at the end of **Phase\_Seg1**).

By the rules given above, the SJW could be the maximum allowed (i.e. 4). However, a large SJW is only necessary when the clock generation of different nodes is not stable or accurate (e.g. if using ceramic resonators). Typically an SJW of 1 is enough. In summary, the required timing parameters are as follows:

Baud rate prescaler (BRP)	= 4
Sync_Seg	= 1
Prop_Seg	= 2
Phase_Seg1	= 7
Phase_Seg2	= 6
SJW	= 1

The sampling point is at  $10T_Q$  that corresponds to 62.5% of the total bit time.

There are several tools available on the Internet for calculating the CAN bus timing parameters accurately. Interested readers should refer to the excellent book of the author on this topic, entitled “Controller Area Network Project, Elektor Int. Media, ISBN: 978-1-907920-04-2”

---

### ***mikroC Pro for PIC CAN Functions***

mikroC Pro for the PIC language provides two sets of libraries for CAN bus applications: the library for PIC microcontrollers with built-in CAN modules, and the library based on the use of SPI bus for PIC microcontrollers having no built-in CAN modules. In this project, we shall only be looking at the library functions available for PIC microcontrollers with built-in CAN modules. Similar functions are available for PIC microcontrollers with no built-in CAN modules.

mikroC CAN functions are supported only by PIC18XXX8 microcontrollers with MCP2551 or similar CAN transceivers. Both standard (11 identifier bits) and extended format (29 identifier bits) messages are supported.

The following mikroC functions are provided:

- CANSetOperationMode,
- CANGetOperationMode,
- CANInitialize,
- CANSetBaudRAtE,
- CANSetMask,
- CANSetFilter,
- CANRead,
- CANWrite.

These functions are described below.

### 1. CANSetOperationMode

This function is used to set the CAN operation mode. The function prototype is

```
void CANSetOperationMode(char mode, char wait_flag)
```

Parameter **wait\_flag** is either 0 or 0xFF. If set to 0xFF, the function blocks and will not return until the requested mode is set. If 0, the function returns as a nonblocking call.

The mode can be one of the following:

- **\_CAN\_MODE\_NORMAL** - normal mode of operation,
- **\_CAN\_MODE\_SLEEP** - SLEEP mode of operation,
- **\_CAN\_MODE\_LOOP** - Loop-back mode of operation,
- **\_CAN\_MODE\_LISTEN** - Listen Only mode of operation,
- **\_CAN\_MODE\_CONFIG** - Configuration mode of operation,

### 2. CANGetOperationMode

This function returns the current CAN operation mode. The function prototype is

```
char CANGetOperationMode(void)
```

### 3. CAN\_Initialize

This function initializes the CAN module. All mask registers are cleared to 0 to allow all messages. Upon execution of this function, the Normal mode is set. The function prototype is

```
void CANInitialize(char SJW, char BRP, char PHSEG1,
                  char PHSEG2, char PROPEG, char CAN_CONFIG_FLAGS)
```

where

- |         |                                     |
|---------|-------------------------------------|
| SJW     | is the Synchronization Jump Width,  |
| BRP     | is the Baud Rate prescaler,         |
| PHSEG1  | is the Phase_Seg1 timing parameter, |
| PHSEG2  | is the Phase_Seg2 timing parameter, |
| PROPSEG | is the Prop_Seg.                    |

CAN\_CONFIG\_FLAGS can be one of the following configuration flags:

Value	Meaning
<b>_CAN_CONFIG_DEFAULT</b>	Default flags
<b>_CAN_CONFIG_PHSEG2_PRG_ON</b>	Use supplied PHSEG2 value
<b>_CAN_CONFIG_PHSEG2_PRG_OFF</b>	Use a maximum of PHSEG1 or information processing Time whichever is greater

*Continued*

Value	Meaning
_CAN_CONFIG_LINE_FILTER_ON	Use the CAN bus line filter for wake-up
_CAN_CONFIG_FILTER_OFF	Do not use a CAN bus line filter
_CAN_CONFIG_SAMPLE_ONCE	Sample bus once at the sample point
_CAN_CONFIG_SAMPLE_THRICE	Sample bus three times prior to the sample point
_CAN_CONFIG_STD_MSG	Accept only standard identifier messages
_CAN_CONFIG_XTD_MSG	Accept only extended identifier messages
_CAN_CONFIG_DBL_BUFFER_ON	Use double buffering to receive data
_CAN_CONFIG_DBL_BUFFER_OFF	Do not use double buffering
_CAN_CONFIG_ALL_MSG	Accept all messages including invalid ones
_CAN_CONFIG_VALID_XTD_MSG	Accept only valid extended identifier messages
_CAN_CONFIG_VALID_STD_MSG	Accept only valid standard identifier messages
_CAN_CONFIG_ALL_VALID_MSG	Accept all valid messages

The above configuration values can be bitwise AND'ed to form complex configuration values.

#### 4. CANSetBaudRate

This function is used to set the CAN bus baud rate. The function prototype is

```
void CANSetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2,
                     char PROPSSEG, char CAN_CONFIG_FLAGS)
```

the arguments of the function are as in function *CANInitialize*.

#### 5. CANSetMask

This function sets the mask for filtering of messages. The function prototype is

```
void CANSetMask(char CAN_MASK, long value, char
                CAN_CONFIGFLAGS)
```

CAN\_MASK can be one of the following:

- \_CAN\_MASK\_B1—Receive Buffer 1 mask value,
- \_CAN\_MASK\_B2—Receive Buffer 2 mask value.

**value** is the mask register value. CAN\_CONFIG\_FLAGS can be either \_CAN\_CONFIG\_XTD (extended message), or \_CAN\_CONFIG\_STD (standard message).

#### 6. CANSetFilter

This function sets filter values. The function prototype is

```
void CANSetFilter(char CAN_FILTER, long value, char
                  CAN_CONFIGFLAGS)
```

CAN\_FILTER can be one of the following:

- \_CAN\_FILTER\_B1\_F1—Filter 1 for Buffer 1,
- \_CAN\_FILTER\_B1\_F2—Filter 2 for Buffer 1,
- \_CAN\_FILTER\_B2\_F1—Filter 1 for Buffer 2,
- \_CAN\_FILTER\_B2\_F2—Filter 2 for Buffer 2,
- \_CAN\_FILTER\_B2\_F3—Filter 3 for Buffer 2,
- \_CAN\_FILTER\_B2\_F4—Filter 4 for Buffer 2.

CAN\_CONFIG\_FLAGS can be either \_CAN\_CONFIG\_XTD (extended message), or \_CAN\_CONFIG\_STD (standard message).

## 7. CANRead

This function is used to read messages from the CAN bus. Zero is returned if no message is available. The function prototype is

```
char CANRead(long *id, char *data, char *datalen, char
            *CAN_RX_MSG_FLAGS)
```

**id** is the CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended). **data** is an array of bytes up to eight where the received data are stored. **datalen** is the length of the received data (1–8).

CAN\_RX\_MSG\_FLAGS can be one of the following:

- \_CAN\_RX\_FILTER\_1—Receive Buffer Filter 1 accepted this message.
- \_CAN\_RX\_FILTER\_2—Receive Buffer Filter 2 accepted this message.
- \_CAN\_RX\_FILTER\_3—Receive Buffer Filter 3 accepted this message.
- \_CAN\_RX\_FILTER\_4—Receive Buffer Filter 4 accepted this message.
- \_CAN\_RX\_FILTER\_5—Receive Buffer Filter 5 accepted this message.
- \_CAN\_RX\_FILTER\_6—Receive Buffer Filter 6 accepted this message.
- \_CAN\_RX\_OVERFLOW—Receive buffer overflow occurred.
- \_CAN\_RX\_INVALID\_MSG—Invalid message received.
- \_CAN\_RX\_XTD\_FRAME—Extended Identifier message received.
- \_CAN\_RX\_RTR\_FRAME—RTR frame message received.
- \_CAN\_RX\_DBL\_BUFFERED—This message was double-buffered.

The above flags can be bitwise AND'ed if desired.

## 8. CANWrite

This function is used to send a message to the CAN bus. A zero is returned if the message cannot be queued (buffer full). The function prototype is

```
char CANWrite(long id, char *data, char datalen, char
            CAN_TX_MSG_FLAGS)
```

**id** is the CAN message identifier. Only 11 or 29 bits may be used depending on the message type (standard or extended). **data** is an array of bytes up to eight where the data to be sent are stored. **datalen** is the length of the data (1–8).

CAN\_TX\_MSG\_FLAGS can be one of the following:

- \_CAN\_TX\_PRIORITY\_0 - Transmit priority 0
- \_CAN\_TX\_PRIORITY\_1 - Transmit priority 1
- \_CAN\_TX\_PRIORITY\_2 - Transmit priority 2
- \_CAN\_TX\_PRIORITY\_3 - Transmit priority 3
- \_CAN\_TX\_STD\_FRAME - Standard Identifier message
- \_CAN\_TX\_XTD\_FRAME - Extended Identifier message
- \_CAN\_TX\_NO\_RTR\_FRAME - Non-RTR message,
- \_CAN\_TX\_RTR\_FRAME - RTR message.

The above flags can be bitwise AND'ed if desired.

### **CAN Bus Programming**

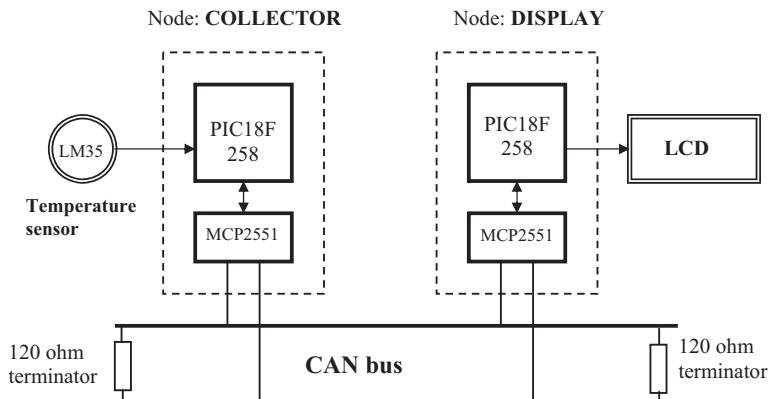
To operate the PIC18F258 microcontroller on the CAN bus, the following steps should be carried out:

- Configure CAN bus I/O port directions (RB2 and RB3).
- Initialize the CAN module (CANInitialize).
- Set CAN module to the CONFIG mode (CANSetOperationMode).
- Set Mask registers (CANSetMask).
- Set Filter registers (CANSetFilter).
- Set CAN module to the Normal mode (CANSetOperationMode).
- Write/Read data (CANWrite/CANRead).

### **CAN Bus Project Description—Temperature Sensor and Display**

This is a simple two-node CAN bus based project. The block diagram of the project is shown in [Figure 7.145](#). The system is made up of two CAN nodes. One node (called the **COLLECTOR** node) reads the temperature from an external semiconductor temperature sensor. The other node (called the **DISPLAY** node) requests the temperature every second and then displays it on an LCD. This process is repeated continuously.

The circuit diagram of the project is given in [Figure 7.146](#). Two CAN nodes are connected together using a 2-m twisted pair cable, terminated with 120- $\Omega$  resistors at both ends.



**Figure 7.145: Block Diagram of the Project.**

### ***The COLLECTOR Processor***

The COLLECTOR processor consists of a PIC18F258 microcontroller with a built-in CAN module and an MCP2551 transceiver chip. The microcontroller is operated from an 8-MHz crystal. The MCLR input is connected to an external reset button. Analog input AN0 of the microcontroller is connected to a LM35DZ-type semiconductor temperature sensor. The sensor can measure temperature in the range 0–100 °C and generates an analog voltage directly proportional to the measured temperature, that is, the output is 10 mV/°C. For example, at 20 °C, the output voltage is 200 mV.

The CAN outputs (RB2/CANTX and RB3/CANRX) of the microcontroller are connected to the TXD and RXD inputs of an MCP2551-type CAN transceiver chip. The CANH and CANL outputs of this chip are connected directly to a twisted cable terminated CAN bus. MCP2551 is an eight-pin chip that supports data rates up to 1 Mb/s. The chip can drive up to 112 nodes. An external resistor connected to pin 8 of the chip controls the rise and fall times of CANH and CANL so that EMI can be reduced. For high-speed operation, this pin should be connected to the ground. A reference voltage equal to  $V_{DD}/2$  is output from pin 5 of the chip.

### ***The DISPLAY Processor***

As in the COLLECTOR processor, the DISPLAY processor consists of a PIC18F258 microcontroller and an MCP2551 transceiver chip. The microcontroller is operated from an 8-MHz crystal. The MCLR input is connected to an external reset button. The CAN outputs (RB2/CANTX and RB3/CANRX) of the microcontroller are connected to the TXD and RXD inputs of the MCP2551. Pins CANH and CANL of the transceiver chip are

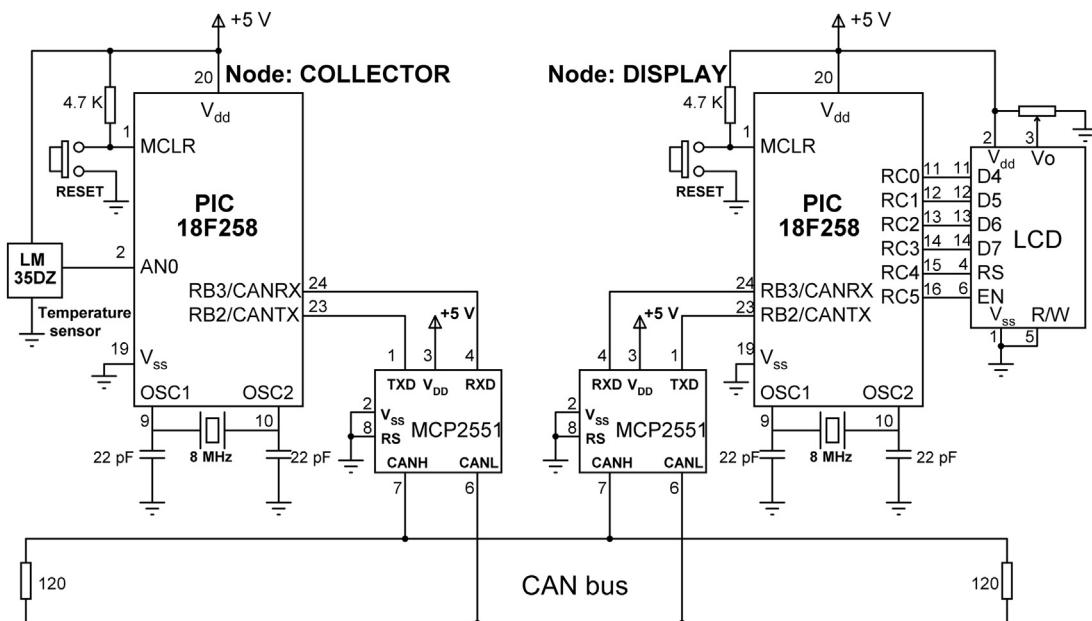


Figure 7.146: Circuit Diagram of the Project.

connected to the CAN bus. An LCD is connected to PORTC of the microcontroller to display the temperature values.

The program listing is in two parts: the COLLECTOR program and the DISPLAY program. The operation of the system is as follows:

- The DISPLAY processor requests the current temperature from the COLLECTOR processor over the CAN bus.
- The COLLECTOR processor reads the temperature, formats it, and sends to the DISPLAY processor over the CAN bus.
- The DISPLAY processor reads the temperature from the CAN bus and then displays it on the LCD.
- The above process is repeated every second.

### **DISPLAY Program**

Figure 7.147 shows the program listing of the DISPLAY program, called CAN-DISPLAY.C. At the beginning of the program, PORTC pins are configured as outputs, RB3 is configured as input (CANRX), and RB2 is configured as output (CANTX). In this project, the CAN bus bit rate is selected as 100 kb/s. With a microcontroller clock frequency of 8 MHz, the timing parameters were calculated to be

---

```
*****
CAN BUS EXAMPLE - NODE: DISPLAY
=====
```

This is the DISPLAY node of the CAN bus example. In this project a PIC18F258 type microcontroller is used. An MCP2551 type CAN bus transceiver is used to connect the microcontroller to the CAN bus. The microcontroller is operated from an 8 MHz crystal with an external reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins RXD and TXD of the transceiver chip respectively. Pins CANH and CANL of the transceiver chip are connected to the CAN bus.

An LCD is connected to PORT C of the microcontroller. The ambient temperature is read from another CAN node and is displayed on the LCD.

The LCD is connected to the microcontroller as follows:

Microcontroller	LCD
RC0	D4
RC1	D5
RC2	D6
RC3	D7
RC4	RS
RC5	EN

CAN speed parameters are:

Microcontroller clock:	8 MHz
CAN Bus bit rate:	100 Kb/s
Sync_Seg:	1
Prop_Seg:	6
Phase_Seg1:	6
Phase_Seg2:	7
SJW:	1
BRP:	1
Sample point:	65%

Author: Dogan Ibrahim

Date: October 2013

File: CAN-DISPLAY.C

---

```
*****// LCD module connections
sbit LCD_RS at RC4_bit;
sbit LCD_EN at RC5_bit;
sbit LCD_D4 at RCO_bit;
sbit LCD_D5 at RC1_bit;
sbit LCD_D6 at RC2_bit;
sbit LCD_D7 at RC3_bit;
```

**Figure 7.147: DISPLAY Program Listing.**

```
sbit LCD_RS_Direction at TRISC4_bit;
sbit LCD_EN_Direction at TRISC5_bit;
sbit LCD_D4_Direction at TRISCO_bit;
sbit LCD_D5_Direction at TRISC1_bit;
sbit LCD_D6_Direction at TRISC2_bit;
sbit LCD_D7_Direction at TRISC3_bit;
// End LCD module connections

void main()
{
    unsigned char temperature, dat[8];
    unsigned char init_flag, send_flag, dt, len, read_flag;
    char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, txt[4];
    long id, mask;

    TRISC = 0;                                // PORTC are outputs (LCD)
    TRISB = 0x08;                            // RB2 is output, RB3 is input
    //
    // CAN BUS Parameters
    //
    SJW = 1;
    BRP = 1;
    Phase_Seg1 = 6;
    Phase_Seg2 = 7;
    Prop_Seg = 6;

    init_flag = _CAN_CONFIG_SAMPLE_THRICE  &
                _CAN_CONFIG_PHSEG2_PRG_ON &
                _CAN_CONFIG_STD_MSG      &
                _CAN_CONFIG_DBL_BUFFER_ON &
                _CAN_CONFIG_VALID_XTD_MSG &
                _CAN_CONFIG_LINE_FILTER_OFF;

    send_flag = _CAN_TX_PRIORITY_0      &
                _CAN_TX_XTD_FRAME     &
                _CAN_TX_NO_RTR_FRAME;

    read_flag = 0;
    //
    // Initialise CAN module
    //
    CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, init_flag);
    //
    // Set CAN CONFIG mode
    //
    CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);

    mask = -1;
    //
    // Set all MASK1 bits to 1's
    //
}
```

**Figure 7.147**  
cont'd

```

        CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_XTD_MSG);
//
// Set all MASK2 bits to 1's
//
        CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_XTD_MSG);
//
// Set id of filter B2_F3 to 3
//
        CANSetFilter(_CAN_FILTER_B2_F3,3,_CAN_CONFIG_XTD_MSG);
//
// Set CAN module to NORMAL mode
//
        CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);

//
// Configure LCD
//
        Lcd_Init();                                // Initializd LCD
        Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD
        Lcd_Out(1,1,"CAN BUS");                  // Display heading on LCD
        Delay_ms(1000);                          // Wait for 2 seconds

//
// Program loop. Read the temperature from Node:COLLECTOR and display
// on the LCD continuously
//
        for(;;)                                  // Endless loop
        {
            Lcd_Cmd(_LCD_CLEAR);                // Clear LCD
            Lcd_Out(1,1,"Temp = ");           // Display "Temp = "
            //
            // Send a message to Node:COLLECTOR and ask for data
            //
            dat[0] = 'T';                      // Data to be sent
            id = 500;                         // Identifier
            CANWrite(id, dat, 1, send_flag);   // Send 'T'
            //
            // Get temperature from node:COLLECT
            //
            dt = 0;
            while(!dt)dt = CANRead(&id, dat, &len, &read_flag);
            if(id == 3)
            {
                temperature = dat[0];
                ByteToStr(temperature,txt);      // Convert to string
                Lcd_Out(1,8,txt);               // Output to LCD
                Delay_ms(1000);                // Wait 1 second
            }
        }
    }
}

```

**Figure 7.147**  
cont'd

```
SJW = 1  
BRP = 1  
Phase_Seg1 = 6  
Phase_Seg2 = 7  
Prop_Seg = 6
```

mikroC Pro for PIC CAN bus function CANInitialize is used to initialize the CAN module. The timing parameters and the initialization flag are specified as arguments in this function. The initialization flag was made up from the bitwise AND of

```
init_flag = _CAN_CONFIG_SAMPLE_THRICE &  
           _CAN_CONFIG_PHSEG2_PRG_ON &  
           _CAN_CONFIG_STD_MSG &  
           _CAN_CONFIG_DBL_BUFFER_ON &  
           _CAN_CONFIG_VALID_XTD_MSG &  
           _CAN_CONFIG_LINE_FILTER_OFF;
```

Where it was specified to sample the bus three times, standard identifier was specified, double buffering was turned on, and the line filter was turned off.

Then, the operation mode was set to CONFIGURATION, and the filter masks and filter values were specified. Both mask 1 and mask 2 were set to all 1's (-1 is a short hand of writing 0xFFFFFFFF, that is, setting all mask bits to 1's) so that matching of all filter bits was required.

Filter 3 for Buffer 2 was set to value 3 so that identifiers having values 3 will be accepted by the receive buffer.

The operation mode is then set to NORMAL. The program then configures the LCD and displays message “CAN BUS” for 1 s on the LCD.

The main program loop executes continuously and starts with a **for** statement. Inside this loop, the LCD is cleared and text “TEMP = ” is displayed on the LCD. Then character “T” is sent over the bus with identifier equal to 500 (the COLLECTOR node filter is set accept identifier 500). This is a request to the COLLECTOR node to send the temperature reading. The program then reads the temperature from the CAN bus, converts it to a string in array **txt**, and then displays it on the LCD. The above process is repeated after a 1-s delay.

### ***COLLECTOR Program***

Figure 7.148 shows the program listing of the COLLECTOR program, called CAN-COLLECTOR.C. The initial part of this program is the same as the DISPLAY program. Here, the receive filter is set to 500 so that messages with identifier 500 can be accepted by the program.

```
*****
CAN BUS EXAMPLE - NODE: COLLECTOR
=====
```

This is the COLLECTOR node of the CAN bus example. In this project a PIC18F258 type microcontroller is used. An MCP2551 type CAN bus transceiver is used to connect the microcontroller to the CAN bus. The microcontroller is operated from an 8 MHz crystal with an external reset button.

Pin CANRX and CANTX of the microcontroller are connected to pins RXD and TXD of the transceiver chip respectively. Pins CANH and CANL of the transceiver chip are connected to the CAN bus.

An LM35DZ type analog temperature sensor is connected to port AN0 of the microcontroller. The microcontroller reads the temperature when a request is received and then sends the temperature value as a byte to Node:DISPLAY on the CAN bus.

CAN speed parameters are:

Microcontroller clock:	8 MHz
CAN Bus bit rate:	100 Kb/s
Sync_Seg:	1
Prop_Seg:	6
Phase_Seg1:	6
Phase_Seg2:	7
SJW:	1
BRP:	1
Sample point:	65%

Author: Dogan Ibrahim  
 Date: October 2013  
 File: CAN-COLLECTOR.C

```
******/
```

```
void main()
{
    unsigned char temperature, dat[8];
    unsigned short init_flag, send_flag, dt, len, read_flag;
    char SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, txt[4];
    unsigned int temp;
    unsigned long mV;
    long id, mask;

    TRISA = 0xFF;                                // PORT A are inputs
    TRISB = 0x08;                                // RB2 is output, RB3 is input
    //
    // Configure A/D converter
    //
    ADCON1 = 0x80;
    //
    // CAN BUS Timing Parameters
```

**Figure 7.148: COLLECT Program Listing.**

```
//  
SJW = 1;  
BRP = 1;  
Phase_Seg1 = 6;  
Phase_Seg2 = 7;  
BRP = 1;  
Prop_Seg = 6;  
  
init_flag = _CAN_CONFIG_SAMPLE_THRICE &  
           _CAN_CONFIG_PHSEG2_PRG_ON &  
           _CAN_CONFIG_STD_MSG &  
           _CAN_CONFIG_DBL_BUFFER_ON &  
           _CAN_CONFIG_VALID_XTD_MSG &  
           _CAN_CONFIG_LINE_FILTER_OFF;  
  
send_flag = _CAN_TX_PRIORITY_0 &  
           _CAN_TX_XTD_FRAME &  
           _CAN_TX_NO_RTR_FRAME;  
  
read_flag = 0;  
//  
// Initialise CAN module  
//  
    CANInitialize(SJW, BRP, Phase_Seg1, Phase_Seg2, Prop_Seg, init_flag);  
//  
// Set CAN CONFIG mode  
//  
    CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);  
  
mask = -1;  
//  
// Set all MASK1 bits to 1's  
//  
    CANSetMask(_CAN_MASK_B1, mask, _CAN_CONFIG_XTD_MSG);  
//  
// Set all MASK2 bits to 1's  
//  
    CANSetMask(_CAN_MASK_B2, mask, _CAN_CONFIG_XTD_MSG);  
//  
// Set id of filter B2_F3 to 500  
//  
    CANSetFilter(_CAN_FILTER_B2_F3, 500, _CAN_CONFIG_XTD_MSG);  
//  
// Set CAN module to NORMAL mode  
//  
    CANSetOperationMode(_CAN_MODE_NORMAL, 0xFF);  
  
//  
// Program loop. Read the temperature from analog temperature  
// sensor  
//
```

**Figure 7.148**  
cont'd

```

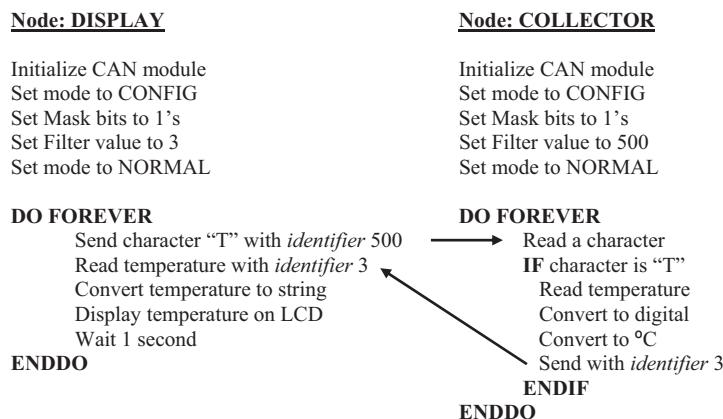
for(;;)                                // Endless loop
{
    //
    // Wait until a request is received
    //
    dt = 0;
    while(!dt) dt = CANRead(&id, dat, &len, &read_flag);
    if(id == 500 && dat[0] == 'T')
    {
        //
        // Now read the temperature
        //
        temp = Adc_Read(0);           // Read temp
        mV = (unsigned long)temp * 5000 / 1024; // in mV
        temperature = mV/10;          // in degrees C
        //
        // send the temperature to Node:Display
        //
        dat[0] = temperature;
        id = 3;                      // Identifier
        CANWrite(id, dat, 1, send_flag); // Send temperature
    }
}
}

```

**Figure 7.148**  
cont'd

Inside the program loop, the program waits until it receives a request to send the temperature. Here, the request is identified by the reception of character “T”. Once a valid request is received, the temperature is read and converted into degrees centigrade (stored in variable **temperature**) and then sent to the CAN bus as a byte with identifier value equal to 3. The above process repeats forever.

Figure 7.149 summarizes the operation of both nodes.



**Figure 7.149: Operation of Both Nodes.**

## **Project 7.18 Multitasking**

Most complex real-time systems consist of a number of tasks running independently. This requires some form of scheduling and task control mechanisms. For example, consider an extremely simple real-time system that must flash an LED at required intervals and at the same time look for a key input from a keypad. One solution would be to scan the keypad in a loop at regular intervals while flashing the LED at the same time. Although this approach may work for simple systems, in most complex real-time systems, a real-time operating system (RTOS) or a multiprocessing approach are usually employed. Multiprocessing is beyond the scope of this project.

An RTOS is a program that manages system resources, scheduling the execution of various tasks in the system and provides services for intertask synchronization and messaging. There are many books and other sources of reference that describe the operation and principles of various RTOS systems.

Every RTOS consists of a kernel that provides the low-level functions, mainly the scheduling, creation of tasks and intertask resource management. Most complex RTOSs also provide file-handling services, disk input–output operations, interrupt servicing, network management, user management, and so on.

A task is an independent thread of execution in a multitasking system, usually with its own local set of data. A multitasking system consists of a number of independent tasks, each running its own code and communicating with each other to have orderly access to shared resources. The simplest RTOS consists of a scheduler that determines the order in which the tasks should run. This scheduler switches from one task to the next by performing a context switching where the context of the running task is stored and context of the next task is loaded so that execution can continue properly with the next task. Tasks are usually in the form of endless loops, executed in an order determined by the scheduler.

Although there exists many variations of scheduling algorithms in use, the three most commonly used algorithms are as follows:

- Cooperative scheduling,
- Round-robin scheduling,
- Preemptive scheduling.

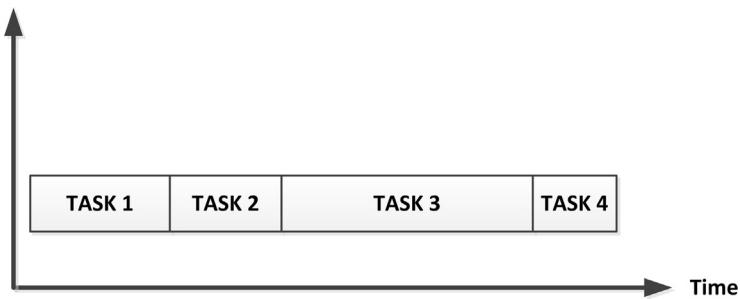


Figure 7.150: Cooperative Scheduling.

### ***Cooperative Scheduling***

This is perhaps the simplest algorithm (Figure 7.150) where tasks voluntarily give up the central processing unit (CPU) usage when they have nothing useful to do, or when they are waiting for some resources to become available (e.g. a key to be pressed and a timer to expire). This algorithm has the disadvantage that certain tasks can use excessive CPU times, and thus not allow some other important tasks to run when needed. Cooperative scheduling is used in simple multitasking systems with no time critical applications. A variation of the pure cooperative scheduling is to prioritize the tasks and run the highest priority computable task when the CPU becomes available. As we shall see in an example project later, cooperative scheduling can easily be implemented in microcontrollers using the *switch* statement.

### ***Round-robin Scheduling***

Round-robin scheduling (Figure 7.151) allocates each task an equal share of the CPU time. In its simplest form, tasks are in a circular queue and when a task's allocated CPU time

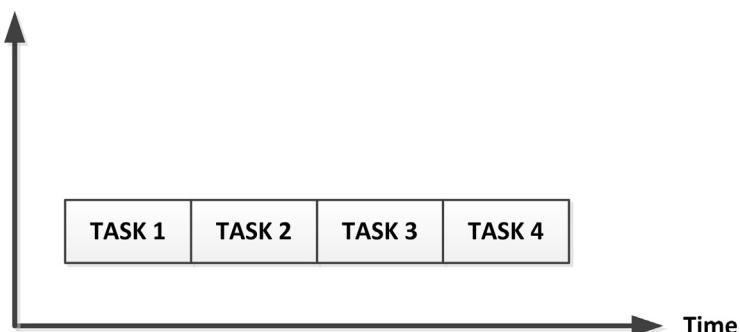


Figure 7.151: Round-robin Scheduling.

expires, the task is put to the end of the queue and the new task is taken from the front of the queue. Round-robin scheduling is not very satisfactory in many real-time applications where each task can have varying amounts of CPU requirements depending upon the complexity of processing required. One variation of the pure round-robin scheduling is to provide priority-based scheduling, where tasks with the same priority levels receive equal amounts of CPU time. It is also possible to allocate different maximum CPU times to each task. An example project is given later on the use of round-robin scheduling.

### **Preemptive Scheduling**

This is the most commonly used and the most complex scheduling algorithm used in real-time systems. Here, the tasks are prioritized, and the task with the highest priority among all other tasks gets the CPU time (Figure 7.152). If a task with a priority higher than the currently executing task becomes ready to run, the scheduler saves the context of the current task and switches to the higher priority task by loading its context.

Usually, the highest priority task runs to completion or until it becomes noncomputable (e.g. waiting for a resource to be available). Although the preemptive scheduling is very powerful, care is needed as an error in programming can place a high priority task in an endless loop and thus not release the CPU to other tasks. Some real-time systems employ a combination of round-robin and preemptive scheduling. In such systems, time critical tasks are usually prioritized and run under preemptive scheduling, whereas less time-critical tasks run under the round-robin scheduling, sharing the left CPU time among themselves.

There are many commercially available, shareware, and open-source RTOS software for the PIC microcontrollers. Brief details of some popular RTOS systems are given below.

Salvo ([www.pumpkininc.com](http://www.pumpkininc.com)) is a low-cost, event driven, priority-based, multitasking RTOS designed for microcontrollers with limited data and program memories. Salvo

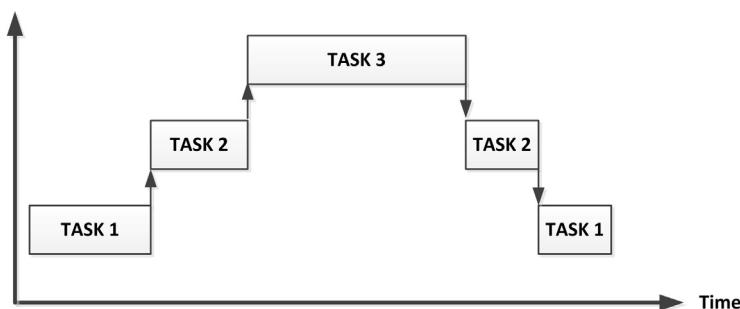


Figure 7.152: Preemptive Scheduling.

can be used for many microcontroller families and it supports large number of compilers, such as Keil C51, Hi-Tech 8051, Hi-Tech PICC-18, MPLAB C18, and many others. A demo version (Salvo Lite) is available for evaluation purposes. The Pro version is the top model aimed for professional applications, supporting unlimited number of tasks with priorities, event flags, semaphores, message queues, and many more features.

CCS compiler ([www.ccsinfo.com](http://www.ccsinfo.com)) from Custom Computer services Inc supports a cooperative RTOS with a number of functions to start and terminate tasks, to send messages between tasks, to synchronize tasks using semaphores, and so on.

CMX-Tiny+ ([www.cmx.com](http://www.cmx.com)) supports large number of microcontrollers. This is a preemptive RTOS with a large number of features such as event-flags, cyclic timers, message queues, and semaphores. Although CMX-Tiny+ is a sophisticated RTOS, it has the disadvantage that the cost is relatively high.

PICos18 ([www.picos18.com](http://www.picos18.com)) is an open-source preemptive RTOS for the PIC18 microcontrollers. The full documentation and the source code are provided free of charge for people wishing to use the product.

MicroC/OS-II (<http://micrium.com>) is a preemptive RTOS, which has been ported to many microcontrollers, including the PIC family. This is a very sophisticated RTOS, providing semaphores, mailboxes, event-flags, timers, memory management, message queues, and many more.

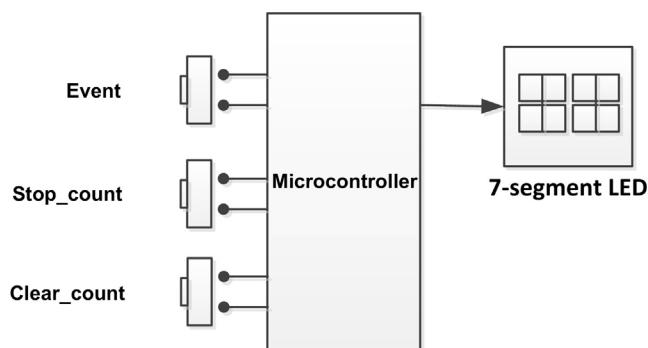
FreeRTOS ([www.freertos.org](http://www.freertos.org)) is an open-source RTOS that can be used in microcontroller-based projects. This is a preemptive RTOS but can be configured for cooperative or hybrid operations.

Finally, OSA-RTOS (<http://picosa.narod.ru>) is freeware RTOS for PIC microcontrollers. The full source code and documentation are available and can be downloaded. OSA is a cooperative multitasking RTOS, offering many features such as semaphores, data queues, mutexes, memory pools, system services, and many more.

## ***Project 1—Using Cooperative Multitasking***

This is a simple project demonstrating how cooperative multitasking can be implemented easily using the C language. This is an example of an event counter with two-digit seven-segment LEDs. The project counts external events and displays the count on the LEDs. The following tasks are used in this project:

Task 1 (REFRESH\_COUNT): This task refreshes the seven-segment LEDs every 3 ms and displays the current count.



**Figure 7.153: Block Diagram of the Project.**

**Task 2 (EVENT):** This is the event counter task. An event is assumed to occur when a push-button switch goes from logic 1 to logic 0.

**Task 3 (CLEAR\_COUNT):** This task clears the count. A 1 to 0 transition of a push-button switch clears the count.

**Task 4 (STOP\_COUNT):** This task stops the count. A 1 to 0 transition of a push-button switch clears the count.

The block diagram of the project is shown in [Figure 7.153](#).

### ***Project Hardware***

The circuit diagram of the project is shown in [Figure 7.154](#). A two-digit seven-segment display is used to display the event count. Three push-button switches are used to initiate an event, clear the count, and to stop the count.

### ***Project PDL***

The project PDL is shown in [Figure 7.155](#).

### ***Project Program***

*mikroC Pro for PIC*

Cooperative scheduling can easily be implemented using the *switch* statement. For example, a system with three tasks can be implemented as follows:

```
for(;;)
{
    state = 1;
    switch(state)
    {
```

```

case 1:
    Implement TASK 1
    state = 2;
    break;
case 2:
    Implement TASK 2
    state = 3;
    break;
case 3:
    Implement TASK 3
    state = 1;
    break;
}
}

```

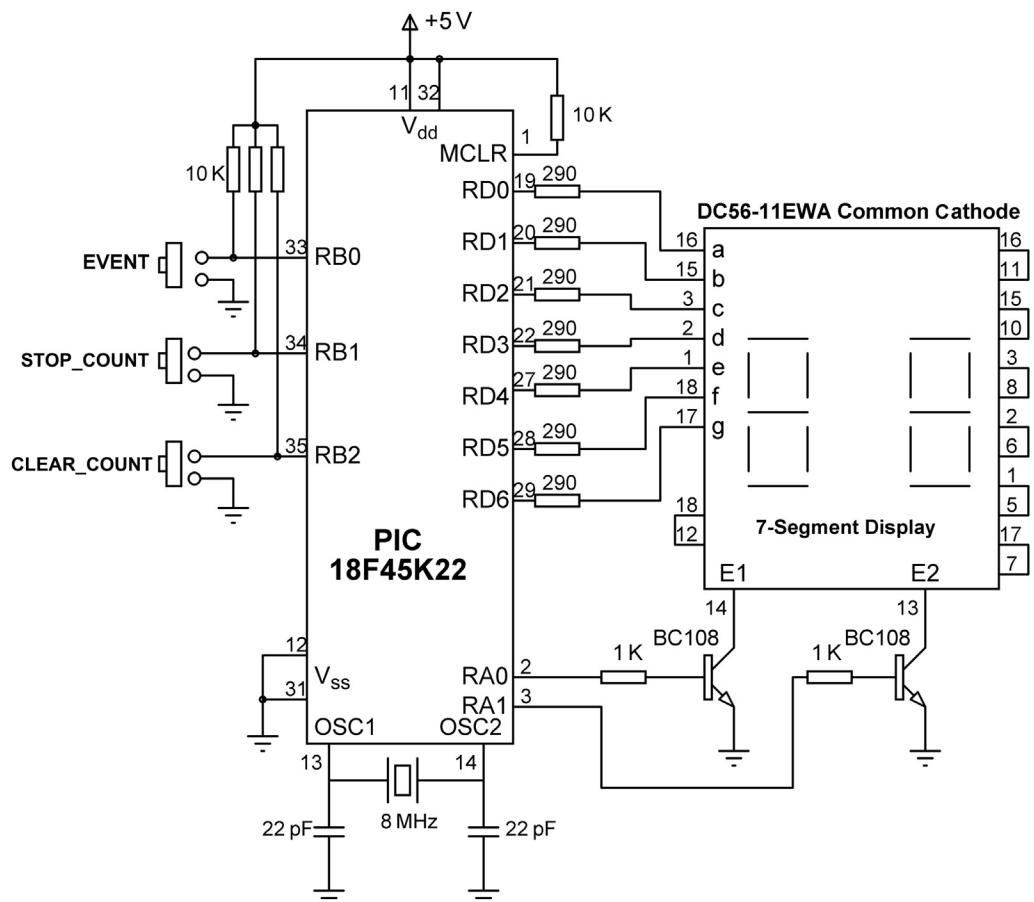


Figure 7.154: Circuit Diagram of the Project.

```
BEGIN
    Configure PORTA, PORTB, PORTD as digital
    Configure PORTA, PORTD as outputs, RB0:RB2 as inputs
    Disable the display
    Configure TIMER0 to interrupt at every millisecond
    DO FOREVER
        CALL TASK1
        CALL TASK2
        CALL TASK3
        CALL TASK4
    ENDDO
END

BEGIN/Timer0 Interrupt
    Re-load TMROL
    Decrement variable Timer by 1
    Re-enable Timer0 interrupts
END/Timer0 Interrupt

BEGIN/DISPLAY
    Extract the bit pattern of the number to be displayed
    Return the bit pattern
END/DISPLAY

BEGIN/TASK1
    IF variable Timer equals 0
        Set variable Timer to 5
        Display variable Cnt
    ENDIF
END/TASK1

BEGIN/TASK2
    IF STOP_COUNT button is pressed
        Clear Count_Flag to 0
```

**Figure 7.155: Project PDL.**

```
        ENDIF

    END/TASK2

BEGIN/TASK3

    IF Clear_Count button is pressed
        Clear Cnt to 0
    ENDIF

    END/TASK3

BEGIN/TASK4

    IF EVENT occurred
        Wait until event is removed
        Increment variable Cnt
    ENDIF

    END/TASK4
```

**Figure 7.155**  
cont'd

The mikroC Pro for the PIC program listing is shown in [Figure 7.156](#) (MIKROC-RTOS1.C). At the beginning of the program, PORTA, PORTB, and PORTD are configured as digital. PORTB and PORTD are configured as outputs and RB0, RB1, and RB2 are configured as inputs. Then, Timer 0 is configured to generate interrupts at every millisecond. The remaining part of the main program implements the multitasking loop where the tasks are executed in order one after the other.

TASK 1 refreshes the seven-segment display every 5 ms. Global variable Timer is loaded with five and is decremented by one every time a timer interrupt is generated. When the variable Timer is 0, the display is refreshed. First, the MSD digit is refreshed, followed by the LSD digit by enabling the corresponding digit enable bits.

```
*****
Dual 7-SEGMENT DISPLAY EVENT COUNTER
=====
```

In this project two common cathode 7-segment LED displays are connected to PORTD of a PIC18F45K22 microcontroller and the microcontroller is operated from an 8 MHz crystal. Digit 1 (right digit) enable pin is connected to port pin RA0 and digit 2 (left digit) enable pin is connected to port pin RA1 of the microcontroller.

The program is an event counter. 3 push-button switches are connected to PORTB as follows:

- RB0 Event push button switch
- RB1 Stop\_Count push button switch
- RB2 Clear\_Count push button switch

Events are assumed to occur when RB0 goes from 1 to 0. When an event occurs, variable Cnt is incremented by 1. The Display is refreshed every 5 ms and the count is displayed. Pressing Stop Count button disables the counting. Pressing the Clear Count button clears the count to 0.

This program uses co-operative multitasking, implemented using a switch statement. There are 4 tasks in the program:

- TASK1: Refreshes the Display every 5ms
- TASK2: Disables counting
- TASK3: Clears counting to zero
- TASK4: Increments counting when an event is detected

Author: Dogan Ibrahim  
 Date: October 2013  
 File: MIKROC-RTOS1.C

```
*****
#define DIGIT1 PORTA.RA0          // Display DIGIT 1 enable
#define DIGIT2 PORTA.RA1          // Display DIGIT 2 enable
#define EVENT PORTB.RB0           // Event input (push button)
#define STOP_COUNT PORTB.RB1      // STOP_COUNT push button
#define CLEAR_COUNT PORTB.RB2     // CLEAR_COUNT push button

unsigned char state = 1;           // Initial value of state variable
unsigned char Count_Flag = 1;       // Set when counting is enables
unsigned char refresh = 0;          // Refreshing time (ms)
unsigned char Cnt = 0;             // Initial value of count
unsigned int Timer = 5;            // Re-load Timer0 for 1ms interrupts
unsigned char Sbutton = 0;          // Decrement variable Timer

// Generate Timer interrupts every milliseconds
void interrupt()
{
    TMROL = 225;                  // Re-load Timer0 for 1ms interrupts
    Timer--;                      // Decrement variable Timer
}
```

**Figure 7.156: mikroC Pro for the PIC Program.**

```

INTCON = 0x20;                                // Re-enable Timer0 interrupts
}

// 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);
}

// This task refreshes the 7-segment LEDs every 5 milliseconds
//
void TASK1()
{
    unsigned char Msd, Lsd;

    if(Timer == 0)                            // If 5 ms has elapsed
    {
        Timer = 5;                          // Re-load variable Timer
        if(refresh == 0)                    // Time to refresh MSD digit ?
        {
            refresh = 1;
            DIGIT1 = 0;
            Msd = Cnt / 10;                // MSD digit
            PORTD = Display(Msd);          // Send to PORTD
            DIGIT2 = 1;                   // Enable digit 2
        }
        else if(refresh == 1)              // Time to refresh LSD digit ?
        {
            refresh = 0;
            DIGIT2 = 0;                  // Disable digit 2
            Lsd = Cnt % 10;              // LSD digit
            PORTD = Display(Lsd);          // Send to PORTD
            DIGIT1 = 1;                   // Enable digit 1
        }
    }
}

// This task stops the count
//
void TASK2()

```

**Figure 7.156**  
cont'd

```
{  
    if(STOP_COUNT == 0)Count_Flag = 0;           // Clear count flag to stop counting  
}  
  
//  
// This task clears the count  
//  
void TASK3()  
{  
    if(CLEAR_COUNT == 0)Cnt = 0;                 // Clear the count to 0  
}  
  
//  
// This task increments the event counter  
//  
void TASK4()  
{  
    if(EVENT == 0 && Count_Flag == 1)Sbutton = 1;      // If event and counting enabled  
    if(EVENT == 1 && Sbutton == 1 && Count_Flag == 1)      // If event has been removed  
    {  
        Cnt++;                                         // Increment count  
        Sbutton = 0;  
    }  
}  
  
}  
  
//  
// Start of MAIN Program  
//  
void main()  
{  
    ANSELA = 0;                                     // Configure PORTA as digital  
    ANSELB = 0;                                     // Configure PORTB as digital  
    ANSELD = 0;                                     // Configure PORTD as digital  
    TRISA = 0;                                      // Configure PORTA as outputs  
    TRISB = 7;                                       // Configure RB0, RB1, RB2 as input  
    TRISD = 0;                                      // Configure PORTD as outputs  
  
    DIGIT1 = 0;                                     // Disable digit 1  
    DIGIT2 = 0;                                     // Disable digit 2  
}  
//  
// Set Timer0 to interrupt at every millisecond  
//  
    TOCON = 0xC5;                                    // Configure T0CON register  
    TMROL = 225;                                     // Load TMROL register  
    INTCON = 0xA0;                                    // Enable Timer0 interrupts  
}  
//  
// Start of multitasking loop  
//
```

Figure 7.156  
cont'd

```

while(1)
{
    switch(state)
    {
        case 1:
            TASK1();                                // Do TASK1
            state = 2;                             // Next task is TASK2
            break;
        case 2:
            TASK2();                                // Do TASK2
            state = 3;                             // Next task is TASK3
            break;
        case 3:
            TASK3();                                // Do TASK3
            state = 4;                             // Next task is TASK4
            break;
        case 4:
            TASK4();                                // Do TASK4
            state = 1;                             // Next task is TASK1
            break;
    }
}
}
}

```

**Figure 7.156**  
cont'd

TASK2 clears flag Count\_Flag so that counting stops, that is, no count is generated when an event occurs.

TASK3 clears the count by clearing variable Cnt.

TASK4 checks the state of the EVENT button (RB0). This button is normally at logic 1. When the button is pressed, it goes to logic 0 and is back at logic 1 when the button is released. It is important that we generate only one count when the button is pressed and released. This task initially sets variable Sbutton to 1 when an event occurs (EVENT = 0) and also when the counting is enabled (Count\_Flag = 1). At this point, the button is in the pressed state, and we have to wait until the button is released before incrementing the count; otherwise, the count increment while the button is kept pressed. The following program code detects when the button is pressed and then it increments the count only when the button is released:

```

if(EVENT == 0 && Count_Flag == 1)Sbutton = 1;
if(EVENT == 1 && Sbutton == 1 && Count_Flag == 1)
{
    Cnt++;
    Sbutton = 0;
}

```

## Project 2—Using Round-Robin Multitasking With Variable CPU Time Allocation

In this project, we will be developing and using a round robin-type multitasking software with variable CPU time allocation. The software, called RTOS.C, can be included in multitasking programs. Each task is allocated maximum processing time selected by the user. The scheduler terminates a task when this time is reached, or when the task voluntarily gives-up CPU time by calling a function. The scheduler is interrupt driven and activates the tasks in order.

Each task in the user program is organized as a C function, running forever in a loop. The first thing a task does is to call scheduler function InitTask, which saves the task return address in an array called TStack. In addition, the maximum allocated duration of each task (in milliseconds) is also stored in array TTime. The program counter of a PIC18F microcontroller is 24 bits wide and is stored in three 8-bit stack registers TOSL, TOSH, and TOSU after a function call or an interrupt (Figure 7.157). These registers are accessed by the scheduler during the saving and restoring of task return addresses.

Figure 7.158 shows the program listing of the multitasking scheduler. In an application, the main program initially calls all the tasks in turn so that their return addresses can be saved. Then, function StartTasks is called. This function calls to SetUpTmrInt in order to configure timer TMR0 so that timer interrupts can be

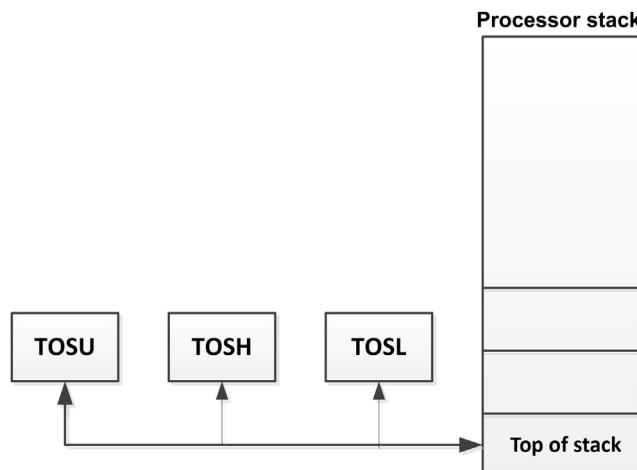


Figure 7.157: PIC18F Microcontroller Stack Structure.

generated every milliseconds for the scheduler. In addition, the return address of Task 0 is pushed onto the stack and a RETURN is executed so that task execution starts from Task 0. At the core of the scheduler, we have the timer ISR. The ISR determines the next task to run and performs the necessary context switching. The following operations are carried out within the ISR:

- Timer register TMR0 is reloaded for 1-ms interrupts.
- Current CPU registers W, STATUS, and BSR are saved.
- If the allocated duration of current task has not expired, then timer interrupts are reenabled and the ISR passes control back to the same task with no context changing.
- Otherwise, the return address of the current task is saved in array TStack.
- Task number of the next task is determined, and its return address is pushed onto processor stack.
- CPU registers W, STATUS, and BSR of the next task are restored.
- Timer interrupts are reenabled, and the ISR passes the CPU control to the next task.

Three timer registers called TimerA, TimerB, and TimerC are decremented every millisecond inside the ISR, and these registers can be used in task timing applications.

### ***Project Description***

This project is the same as the previous project where a two-digit seven-segment LED is used with four tasks:

Task 1 (REFRESH\_COUNT): This task refreshes the seven-segment LEDs every 3 ms and displays the current count.

Task 2 (EVENT): This is the event counter task. An event is assumed to occur when a push-button switch goes from logic 1 to logic 0.

Task 3 (CLEAR\_COUNT): This task clears the count. A 1–0 transition of a push-button switch clears the count.

Task 4 (STOP\_COUNT): This task stops the count. A 1–0 transition of a push-button switch clears the count.

The block diagram of the project is as shown in [Figure 7.153](#).

### ***Project Hardware***

The circuit diagram of the project is as given in [Figure 7.154](#).

```

#pragma disablecontextsaving
#define freq 8                                // Clock frequency
#define Prescale 64                            // Timer prescaler
#define T 1000                                 // 1000 us (1 ms) timer
#define Timervalue 256-(T*freq/(4*Prescale))
#define StopTask while(1){Swapp = 1; INTCON.F2 = 1;}
#define SwapTask {Swapp = 1; INTCON.F2 = 1;}
unsigned char TMRO = Timervalue;
unsigned char Temp, Twreg, Tstatus, Tbsr, Swapp = 0;
unsigned char Saved[MaxTsk][3];
unsigned char TaskNumber = 0;
unsigned char TStack[MaxTsk][4];
unsigned int TCount[MaxTsk];
unsigned int TTime[MaxTsk];
signed char TimerA, TimerB, TimerC;

//  

// The program jumps here every 1 ms  

//  

void interrupt()
{
    TMROL = TMRO;                           // Reload timer register
    TimerA--;                               // Two general purpose timers
    TimerB--;
    TimerC--;

    Twreg = WREG; Tstatus = STATUS; Tbsr = BSR;           // Get current context

    TCount[TaskNumber]++;
    if((Swapp == 1) || (TCount[TaskNumber] >= TTime[TaskNumber]))
    {
        TCount[TaskNumber] = 0;
        if(Swapp == 1)Swapp = 0;

        Saved[TaskNumber][0] = Twreg;     Saved[TaskNumber][1] = Tstatus;
        Saved[TaskNumber][2] = Tbsr;
    }
    // Save return address of current task
    //
    TStack[TaskNumber][0] = TOSL;      TStack[TaskNumber][1] = TOSH;
    TStack[TaskNumber][2] = TOSU;
    asm POP
    //
    // Get next task, and save its return address on TOS
    //
    TaskNumber++;
    if(TaskNumber > MaxTsk - 1)TaskNumber = 0;
    asm PUSH
    Temp = TStack[TaskNumber][0];      TOSL = Temp;
    Temp = TStack[TaskNumber][1];      TOSH = Temp;

```

Figure 7.158: Scheduler (RTOS.C) Program Listing.

```

        Temp = TStack[TaskNumber][2];      TOSU = Temp;
    //
    // Restore task registers and return from interrupt
    //
    INTCON = 0x20;
    Temp = Saved[TaskNumber][1];      BSR = Saved[TaskNumber][2];
    WREG = Saved[TaskNumber][0];      STATUS = Temp;
}
else
{
    INTCON = 0x20;                  WREG = Twreg;
    BSR = Tbsr;                   STATUS = Tstatus;
}
asm retfie 0

//
// Set up Timer0 interrupts every 100 microseconds
//
void SetUpTmrInt(void)
{
    TOCON = 0xC5;                      // Prescaler = 64
    TMROL = TMRO;
    INTCON = 0xA0;
}

//
// Store tasks return addresses on stack
//
void initTask(unsigned char TaskNo, unsigned int t)
{
    TStack[TaskNo][0] = TOSL;           TStack[TaskNo][1] = TOSH;
    TStack[TaskNo][2] = TOSU;           TTime[TaskNo] = t;
    asm POP
}

//
// Start tasks
//
void StartTasks(void)
{
    SetUpTmrInt();
    Temp = TStack[0][0];              TOSL = Temp;
    Temp = TStack[0][1];              TOSH = Temp;
    Temp = TStack[0][2];              TOSU = Temp;
    asm RETURN
}

```

**Figure 7.158**  
cont'd

## Project Program

### mikroC Pro for PIC

The mikroC Pro for PIC program listing is shown in [Figure 7.159](#) (MIKROC-RTOS2.C). At the beginning of the program, the number of tasks is specified using symbol MaxTsk, and multitasking scheduler file RTOS.C is included in the program.

The main program configures PORTA, PORTB, and PORTD as digital. Timer variable TimerA is set to 4 so that the LED refreshing time can be set to be every 4 ms in Task0. The tasks are then called one after the other and function StartTasks is called to start the tasks and pass control to the multitasking scheduler.

TASK0 refreshes the seven-segment display every 4 ms. Global variable TimerA is loaded with four and is decremented by one every time a timer interrupt is generated. When variable TimerA is  $\leq 0$ , the display is refreshed. First, the MSD digit is refreshed, followed by the LSD digit by enabling the corresponding digit enable bits. Function call InitTask(0, 1) pushes the return address of Task0 on stack and allocates maximum processing time of 1 ms to TASK0.

TASK1 clears flag Count\_Flag so that counting stops, that is, no count is generated when an event occurs. Function call InitTask(1, 1) pushes the return address of Task1 on stack and allocates maximum processing time of 1 ms to TASK1. Note that the following code is used for this task:

```
while(1)
{
    while(STOP_COUNT == 1); // Wait until STOP_COUNT button is pressed
    Count_Flag = 0;        // Clear Count_Flag to stop counting
    SwapTask;              // Return to scheduler (Give up the CPU)
}
```

Note that since 1 ms is allocated to this task, the program will remain here for at least 1 ms if the button is not pressed. A quicker way of checking whether or not the button is pressed would be as in the following code. Here, the button is checked and if it is not pressed, the task releases the CPU by calling to function SwapTask:

```
while(1)
{
    if(STOP_COUNT == 0)Count_Flag = 0;
    SwapTask;              // Return to scheduler (Give up the CPU)
}
```

TASK2 clears the count by clearing variable Cnt.

TASK3 checks state of the EVENT button (RB0) and increments the event count Cnt when an event is detected and at the same time if the counting is enabled (Count\_Flag = 1).

```
*****
Dual 7-SEGMENT DISPLAY EVENT COUNTER WITH ROUND-ROBIN MULTITASKING
=====
```

In this project two common cathode 7-segment LED displays are connected to PORTD of a PIC18F45K22 microcontroller and the microcontroller is operated from an 8 MHz crystal. Digit 1 (right digit) enable pin is connected to port pin RA0 and digit 2 (left digit) enable pin is connected to port pin RA1 of the microcontroller.

The program is an event counter. 3 push-button switches are connected to PORTB as follows:

- RB0 Event push button switch
- RB1 Stop Count push button switch
- RB2 Clear Count push button switch

Events are assumed to occur when RB0 goes from 1 to 0. When an event occurs the variable Cnt is incremented by 1. The Display is refreshed every 5 ms and the count is displayed. Pressing Stop Count button disables the counting. Pressing the Clear Count button clears the count to 0.

This program uses round-robin multitasking algorithm with the modification that the allocated CPU time to each task can be set by the user.

There are 4 tasks in the program:

- TASK0: Refreshes the Display every 5ms
- TASK1: Disables counting
- TASK2: Clears counting to zero
- TASK3: Increments counting when an event is detected

Author: Dogan Ibrahim  
 Date: October 2013  
 File: MIKROC-RTOS2.C

```
*****
// Define number of tasks
// Include RTOS.C file

#define DIGIT1 PORTA.RA0          // Display DIGIT 1 enable
#define DIGIT2 PORTA.RA1          // Display DIGIT 2 enable
#define EVENT PORTB.RB0            // Event input (push button)
#define STOP_COUNT PORTB.RB1       // STOP_COUNT push button
#define CLEAR_COUNT PORTB.RB2      // CLEAR_COUNT push button

unsigned char refresh = 0;
unsigned char Count_Flag = 1;
unsigned char Cnt = 0;

//
// 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.
//
```

**Figure 7.159: mikroC Pro for PIC Program.**

```

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);
}

// This task refreshes the 7-segment LEDs every 5 ms
//
void TASK0()
{
    unsigned char Msd, Lsd;

    InitTask(0,1);                                     // Allocate max 1 ms to TASK0
    while(1)
    {
        if(TimerA <= 0)                                // Time to refresh ?
        {
            TimerA = 4;                                // Reload TimerA with 4 ms
            if(refresh == 0)                            // Time to refresh MSD digit ?
            {
                refresh = 1;
                DIGIT1 = 0;                             // MSD digit
                Msd = Cnt / 10;                         // Send to PORTD
                PORTD = Display(Msd);                  // Enable digit 2
                DIGIT2 = 1;
            }
            else if(refresh == 1)                      // Time to refresh LSD digit ?
            {
                refresh = 0;
                DIGIT2 = 0;                            // Disable digit 2
                Lsd = Cnt % 10;                         // LSD digit
                PORTD = Display(Lsd);                  // Send to PORTD
                DIGIT1 = 1;
            }
        }
        SwapTask;                                       // Return to scheduler
    }
}

// This task stops the count
//
void TASK1()                                         // Allocate maximun 1 ms to TASK1
{
    InitTask(1,1);
}

```

**Figure 7.159**  
cont'd

```

        while(1)
        {
            while(STOP_COUNT == 1);                                // Wait until button is pressed
            Count_Flag = 0;                                       // Clear count flag to stop counting
            SwapTask;                                            // Return to scheduler
        }
    }

    //
    // This task clears the count
    //
void TASK2()
{
    InitTask(2,1);                                         //Allocate maximum 1 ms to TASK2
    while(1)
    {
        while(CLEAR_COUNT == 1);                            // Wait until button is pressed
        Cnt = 0;                                           // Clear Cnt
        SwapTask;                                          // Return to scheduler
    }
}

//
// This task increments the event counter
//
void TASK3()
{
    InitTask(3,1);                                         // Allocate maximum 1 ms to TASK3
    while(1)
    {
        while(EVENT == 1);                                // Wait until button press -release
        while(EVENT == 0);
        if(Count_Flag == 1)Cnt++;
        SwapTask;                                          // Increment Cnt
                                                       // Return to scheduler
    }
}

//
// Start of MAIN Program
//
void main()
{
    ANSELA = 0;                                           // Configure PORTA as digital
    ANSELB = 0;                                           // Configure PORTB as digital
    ANSELD = 0;                                           // Configure PORTD as digital
    TRISA = 0;                                            // Configure PORTA as outputs
    TRISB = 7;                                             // Configure RB0, RB1, RB2 as input
    TRISD = 0;                                            // Configure PORTD as outputs
}

```

**Figure 7.159**  
cont'd

```
DIGIT1 = 0;                                // Disable digit 1
DIGIT2 = 0;                                // Disable digit 2
TimerA = 4;                                 // Set TimerA variable to 4ms

TASK0();                                    // Initialize TASK0
TASK1();                                    // Initialize TASK1
TASK2();                                    // Initialize TASK2
TASK3();                                    // Initialize TASK3
StartTasks();                               // Start Tasks

}
```

**Figure 7.159**  
cont'd

### **Project 7.19—Stepper Motor Control Projects—Simple Unipolar Motor Drive**

This project is about using stepper motors in microcontroller-based systems. This is an introductory project where a stepper motor is driven from a microcontroller.

Before going into the details of the project, it is worthwhile to look at the theory and operation of stepper motors briefly.

Stepper motors are commonly used in printers, disk drives, position control systems, and many more systems where precision position control is required. Stepper motors come in a variety of sizes, shapes, strengths, and precision. There are two basic types of stepper motors: unipolar and bipolar.

#### ***Unipolar Stepper Motors***

Unipolar stepper motors have two identical and independent coils with center taps, and having five, six, or eight wires ([Figure 7.160](#)).

Unipolar stepper motors can be driven in three modes: One-phase full-step sequencing, two-phase full-step sequencing and two-phase half-step sequencing.

##### ***One-phase Full-step Sequencing***

[Table 7.20](#) shows the sequence of sending pulses to the motor. Each cycle consists of four pulses.

##### ***Two-phase Full-step Sequencing***

[Table 7.21](#) shows the sequence of sending pulses to the motor. The torque produced is higher in this mode of operation.

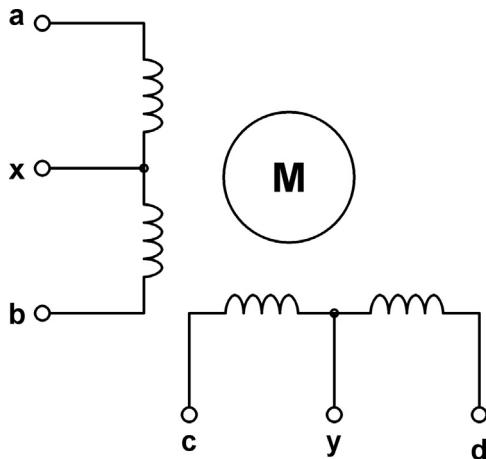


Figure 7.160: Unipolar Stepper Motor Windings.

Table 7.20: One-phase Full-step Sequencing.

Step	a	c	b	d
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

### Two-phase Half-step Sequencing

Table 7.22 shows the sequence of sending pulses to the motor. This mode of operation gives more accurate control of the motor rotation, but it requires twice as many pulses for each cycle.

As we shall see later in the project, the motor can be connected to a microcontroller using power transistors or power MOSFET transistors.

Table 7.21: Two-phase Full-step Sequencing.

Step	a	c	b	d
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

Table 7.22: Two-phase Half-step Sequencing.

Step	a	c	b	d
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

### Bipolar Stepper Motors

Bipolar stepper motors have two identical and independent coils and four wires, as shown in Figure 7.161.

The control of bipolar stepper motors is slightly more complex. Table 7.23 shows the driving sequence. The “+” and “-” signs denote the polarity of the voltage that should be given to the motor legs. Bipolar stepper motors are usually driven using H-bridge circuits.

### Project Description

In this project, a unipolar stepper motor is used and the motor is rotated for 100 turns before it is then stopped.

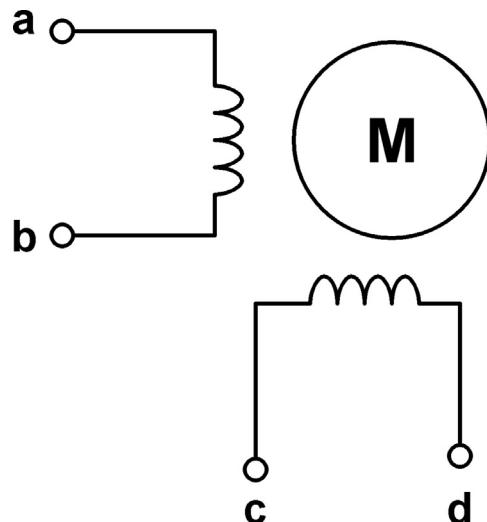


Figure 7.161: Bipolar Stepper Motor Windings.

Table 7.23: Bipolar Stepper Motor Driving Sequence.

Step	a	c	b	d
1	+	-	-	-
2	-	+	-	-
3	-	-	+	-
4	-	-	-	+

### Project Hardware

The circuit diagram of the project is shown in [Figure 7.162](#). In this project, an UAG2 type unipolar stepping motor is used. The motor is connected to RB0:RB3 pins of the microcontroller via IRLI520N-type power MOSFET transistors. UAG2 is a small stepper motor with an  $18^\circ$  stepping angle. Thus, a complete revolution requires 20 pulses. In this

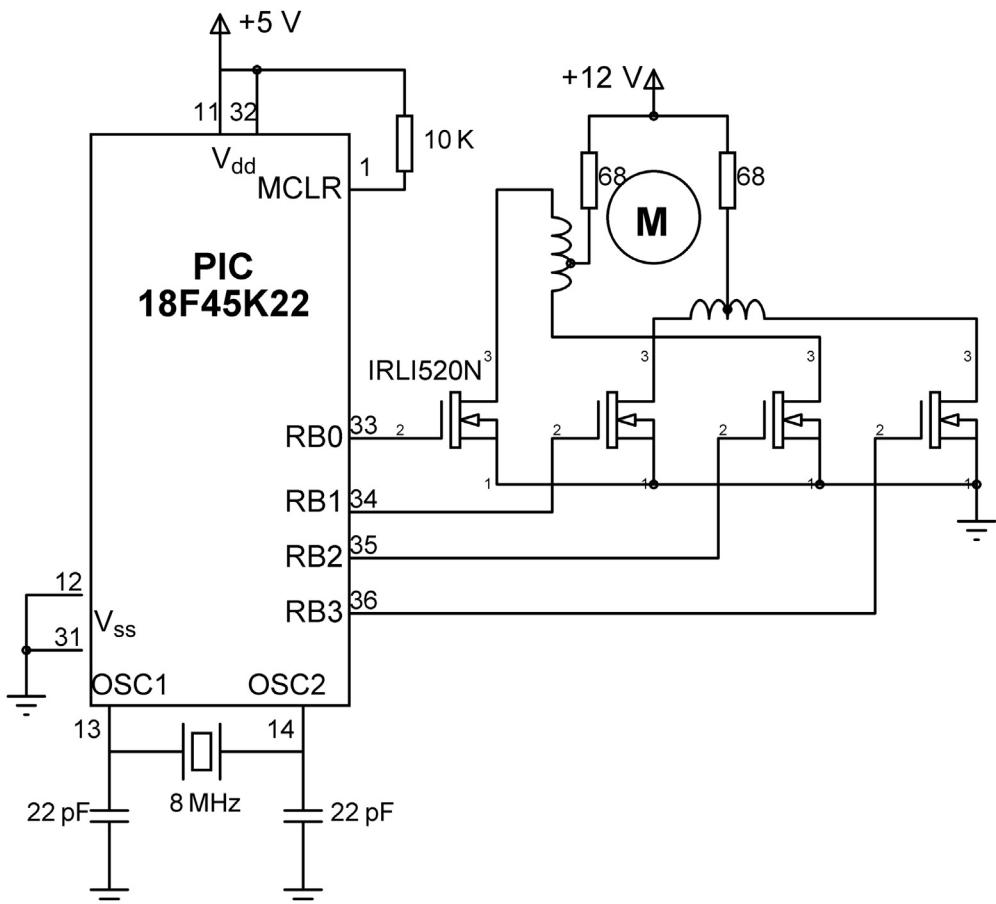


Figure 7.162: Circuit Diagram of the Project.

example, the motor rotates 100 turns (i.e. 2000 pulses are given), and then it stops. A 3-ms delay is inserted between each pulse to slow down the motor.

The pin connections of the UAG2 motor is as follows:

Pin	Description
1	Start of first coil
3	Middle connection of first coil
5	End of first coil
2	Start of second coil
4	Middle connection of second coil
6	End of second coil

### **Project PDL**

The project PDL is shown in [Figure 7.163](#).

### **Project Program**

#### *mikroC Pro for PIC*

The mikroC Pro for the PIC program listing is given in [Figure 7.164](#) (MIKROC-USTP1.C). The motor is operated in one-phase full-step sequencing mode. At the beginning of the program, the required number of revolutions and the motor step size are defined. Inside the main program array Step stores the sequence of pulses to be sent to the motor in each cycle. PORTB is configured as a digital output. Then, the cycle count is calculated and pulses are sent to the motor inside two *for* loops. The motor rotates 100 revolutions where 2000 pulses are sent to the motor. A 3-ms delay is inserted between each pulse. Therefore, the motor operates for 6 s. The speed of the motor can be calculated to be 1000 revolutions per minute (rpm).

```
BEGIN
    Configure PORTB as digital output
    Calculate number of cycles required
    DO for the number of cycles
        DO 4 times
            Send pulse sequence to the motor
            Wait 3ms
    ENDDO
    ENDDO
END
```

**Figure 7.163: Project PDL.**

```
*****
UNIPOLAR STEPPER MOTOR PROJECT
=====

In this project an UAG2 model unipolar stepper motor is connected to PORTB of a PIC18F45K22
microcontroller through IRLI520N type MOSFET transistor switches.

The program rotates the motor 100 times and then stops. In total 2000 pulses are sent to the
motor. 3ms delay is inserted between each pulse. Therefore, the motor rotates 100 revolutions
in 6 s (2000 x 3 ms = 6 s) and then stops.

Author: Dogan Ibrahim
Date: October 2013
File: MIKROC-USTP1.C
*****/
const unsigned int Req_Rev_Count = 100;           // Required no of revs
const unsigned char Step_Size = 18;                // Motor Step Size (degrees)

void main()
{
    unsigned char Step[4] = {1, 2, 4, 8};
    unsigned int One_Rev_Step, Step_Count, Cycle_Count, j;
    unsigned char i;

    ANSELB = 0;                                     // Configure PORTB as digital
    TRISB = 0;                                     // Configure PORTB as digital

    One_Rev_Step = 360/Step_Size;                    // No of steps for 1 revolution
    Step_Count = Req_Rev_Count*One_Rev_Step;         // Total no of steps required
    Cycle_Count = Step_Count/4;                     // No of cycles

    for(j = 0; j < Cycle_Count; j++)              // Do for all cycles
    {
        for(i = 0; i < 4; i++)                    // Do for all steps
        {
            PORTB = Step[i];                      // Send pulses to the motor
            Delay_Ms(3);                         // 3 ms delay between each pulse
        }
    }
    while(1);                                      // End. Wait here forever
}
```

**Figure 7.164:** mikroC Pro for PIC Program.

## Project 7.20—Stepper Motor Control Projects—Complex Control Of A Unipolar Motor

In this project, a unipolar stepper motor is controlled in the following order:

- Turn 200 revolutions clockwise,
- Wait 5 s,
- Turn 50 revolutions anticlockwise,
- Wait 3 s,
- Turn 100 revolutions clockwise,
- Wait 1 s,
- Stop.

### Project Hardware

The circuit diagram of the project is shown in [Figure 7.165](#). In this project, a UCN5804B type stepper motor controller chip is used. This chip can drive small unipolar stepper motors up to +35 V and 1.25 A. The chip is connected to the microcontroller via its Step (pin 11) and Direction (pin 14) pins. The chip also has half-step (pin 10) and phase (pin 9) selection inputs. Depending upon the connection of pins 9 and 10, the chip can operate in one-phase full-step, two-phase full-step, or in two-phase half-step sequencing modes. In this project, both pins 9 and 10 are connected to ground to operate in two-phase full-step sequencing mode. The direction input controls direction of the motor. The motor rotates one step when a pulse is applied to the Step input. Notice that diodes are used at the output pins of the microcontroller to protect the pins from negative voltage.

### Project Program

#### mikroC Pro for PIC

The mikroC Pro for PIC program listing is shown in [Figure 7.166](#) (MIKROC-USTP2.C). The speed of a stepper motor depends upon the time delay between the step inputs. If the time delay between the steps is  $T$ , and the motor step constant is  $\beta$  degrees, then the motor rotates  $\beta/T$  steps in a second. Since a complete revolution is  $360^\circ$ , the number of revolutions in a second is  $\beta/360T$ . The number of revolutions per minute, that is, the revolutions per minute of the motor is then given by

$$\text{RPM} = 60\beta/360T$$

or

$$\text{RPM} = \beta/6T.$$

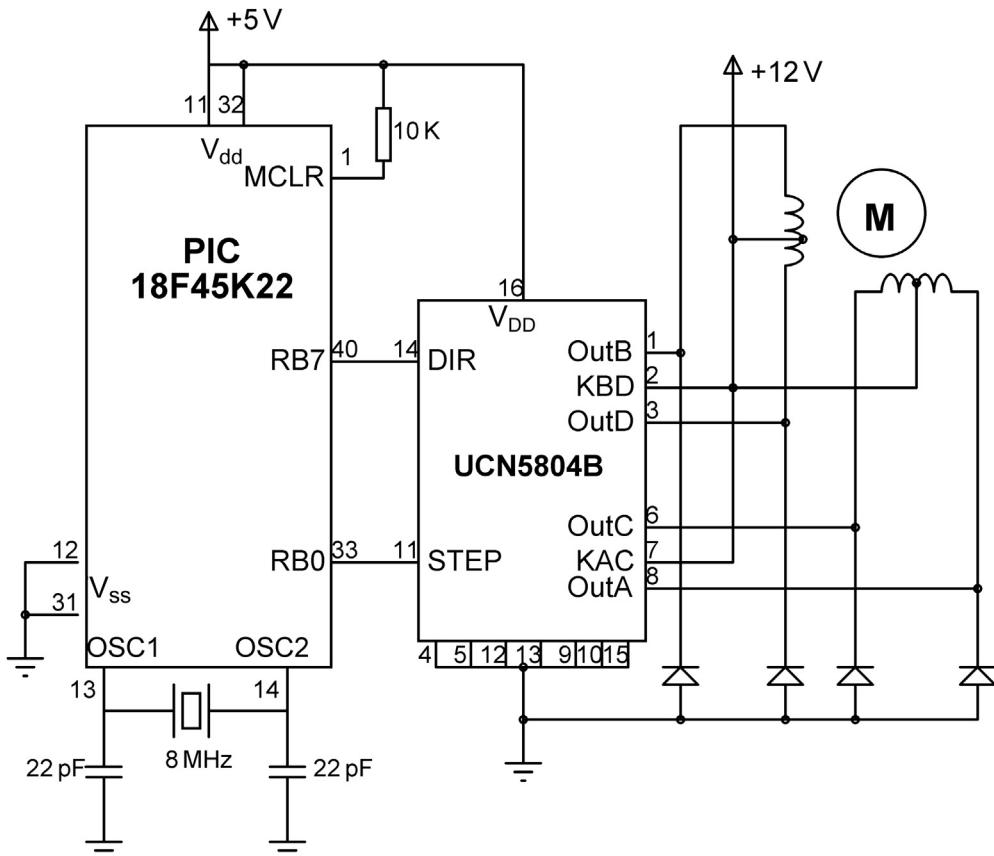


Figure 7.165: Circuit Diagram of the Project.

We can also write

$$T = \beta / 6 \times RPM.$$

In this project, the motor has  $\beta = 18^\circ$  and therefore

$$T = 3/RPM \text{ where } T \text{ is in seconds}$$

or

$$T = 3000/RPM \text{ where } T \text{ is in milliseconds.}$$

If the required revolutions per minute is 500 rpm (assuming this is below the maximum revolutions per minute that can be provided by the motor) then the delay between each step is

$$T = 3000/500 = 6 \text{ ms}$$

```
*****
        COMPLEX UNIPOLAR STEPPER MOTOR ROTATION
=====

In this project a UAG2 model unipolar stepper motor is connected to PORTB of a PIC18F45K22
microcontroller through a UCN5804B type motor controller chip

The program rotates the motor clockwise, or anticlockwise as requested with the required
amount of Delay between each rotation.

In this project the motor is rotated as follows:

Turn 200 revolutions clockwise
Wait 5 seconds
Turn 50 revolutions anticlockwise
Wait 3 seconds
Turn 100 revolutions clockwise
Wait 1 second
Stop

Author: Dogan Ibrahim
Date: October 2013
File: MIKROC-USTP2.C
*****/
#define STEP PORTB.RB0           // UCN5804B Step input
#define DIRECTION PORTB.RB7      // UCN5804B Direction input

const unsigned char Step_Size = 18;          // Motor Step Size (degrees)
const unsigned int RPM = 500;                // Required RPM

unsigned int Step_Count, Delay;

// This function operates the stepper motor as requested. "revcnt" is the rev count, "revdir"
// is the rev direction, and "gapdly" is the intergap Delay
// If the rev count is negative the motor rotates in the required direction continuously
// if rev is equal to 0 then the motor stops
//
void Stepper_Motor(int revcnt, unsigned char revdir, unsigned int gapdly)
{
    unsigned int k,p;

    if(revcnt < 0)
    {
        DIRECTION = revdir;
        for(;;)
        {
            STEP = 1;                                // Send pulse to the motor
            STEP = 0;
            VDelay_Ms(Delay);
        }
    }
}
```

Figure 7.166: mikroC Pro for PIC Program.

```

    }
else
{
    p = revcnt*Step_Count;
    DIRECTION = revdir;
    for(k = 0; k < p; k++)
    {
        STEP = 1;                                // Send pulse to the motor
        STEP = 0;
        VDelay_Ms(Delay);
    }
    for(k = 0; k < gapdly; k++)Delay_Ms(1);           // Inter command delay
}
}

// This is the main program. The required Rev_Count, Rev_Direction, and Inter_Delay
// must be specified here. In this example, the motor rotates 200 revs clockwise, then
// waits 5s, rotates 50 revs anticlockwise, waits 3s, rotates 100revs clockwise, waits
// 1s and then stops
//
void main()
{
    int Rev_Count[] = {200, 50, 100, 0};           // 0 is the terminator
    unsigned char Rev_Direction[] = {0, 1, 0};       // 0=clockwise, 1=anticlockwise
    unsigned char Inter_Delay[] = {5000, 3000, 1000}; // Delay in ms

    unsigned char j;

    ANSELB = 0;                                  // Configure PORTB as digital
    TRISB = 0;                                  // Configure PORTB as digital
    STEP = 0;                                   // Step = 0 to start with

    j = 0;
    Delay = 3000/RPM;                           // Delay after each command
    Step_Count = 360/Step_Size;

    while(Rev_Count[j] != 0)
    {
        Stepper_Motor(Rev_Count[j], Rev_Direction[j], Inter_Delay[j]);
        j++;
    }

    while(1);                                    // End. Wait here forever
}

```

**Figure 7.166**  
cont'd

Three arrays are used in the program to specify the required number of revolutions, the direction of rotation, and the interstep gap:

Rev\_Count: This array stores the required number of revolutions. A 0 entry specifies the end of the array. A negative value indicates that the motor is required to rotate continuously in the specified direction.

Rev\_Direction: This array stores the corresponding motor rotation direction. 0 specifies clockwise and 1 specified anticlockwise rotation.

Inter\_Delay: This array stores the required delay after each command.

For the example in this project, the arrays should have the following values:

```
Rev_Count[ ] = {200, 50, 100, 0};      // 0 is the terminator  
Rev_Direction[ ] = {0, 1, 0};           // 0 = clockwise, 1 = anticlockwise  
Inter_Delay[ ] = {5000, 3000, 1000};    // Delay is in ms
```

At the beginning of the program, symbols STEP and DIRECTION are assigned to port pins RB0 and RB7, respectively. Then, the above arrays are initialized, the step count is found, and a loop is formed. Inside this loop, the motor is activated by calling function Stepper\_Motor. This array has three arguments: revolution count, revolution direction, and the delay after each command. If the revolution count is negative, then the motor rotates continuously in the specified direction. The rotation stops when the revolution is specified as 0. The motor is rotated by setting the required direction and then sending pulses to the STEP input of the UCN5804N controller chip.

### ***Project 7.21—Stepper Motor Control Project—Simple Bipolar Motor Drive***

The Bipolar Stepper motor is similar to the unipolar motor discussed in previous projects. The bipolar consists of two coils, but there is no center tap. As a result of this, the bipolar motor requires a controller where the current flow through the coils can be reversed. A bipolar motor is capable of a higher torque since entire coils may be energized, not just half of the coils. Bipolar stepper motors are usually controlled using H-bridge circuits where the current flow through the coils can easily be reversed.

#### ***Project Description***

In this project, a bipolar stepper motor is used. The motor is rotated 10 revolutions in one direction, then stopped for 5 s, and then rotated 10 revolutions in the other direction, and is then stopped.

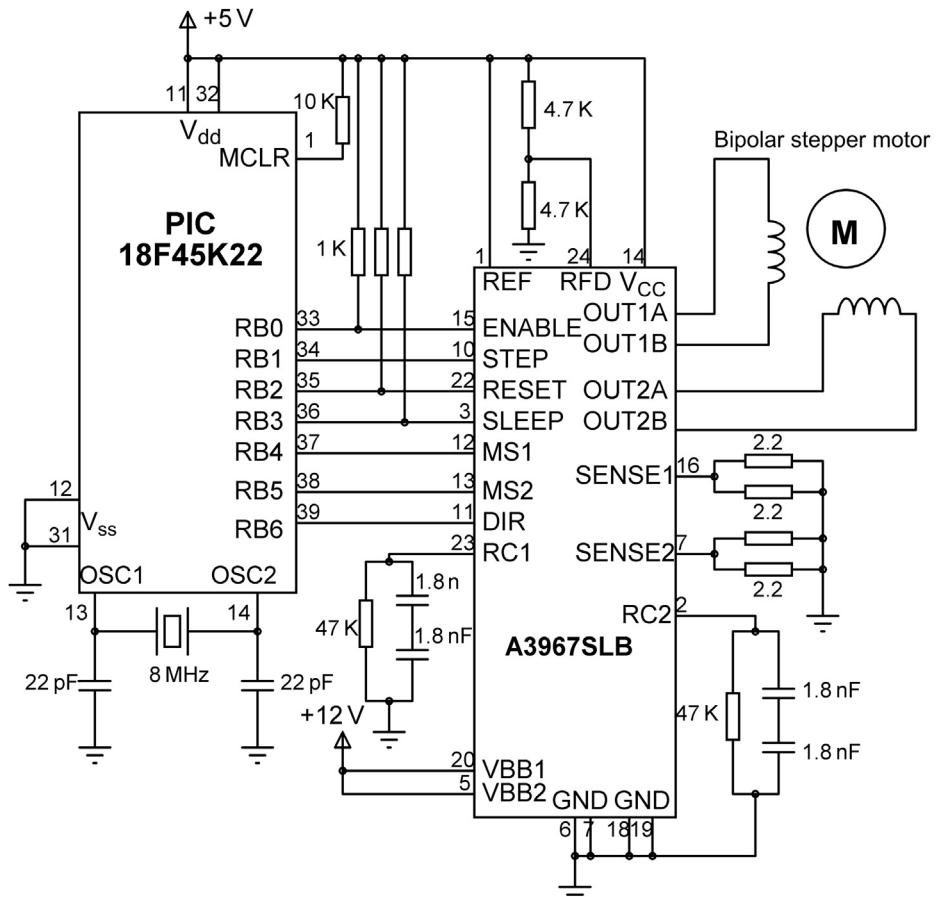


Figure 7.167: Circuit Diagram of the Project.

### Project Hardware

The circuit diagram of the project is shown in [Figure 7.167](#). In this project, A3967SLB type bipolar stepper motor controller chip is used. This chip can operate a bipolar stepper motor in the following modes, controlled by its MS1 and MS2 inputs:

- Full step,
- Half step,
- Quarter step,
- Eight microsteps.

In this project, the full step mode is used where  $MS1 = MS2 = 0$ . The other pins of interest are

**STEP:** A low-to-high transition on this pin rotates the motor by one step.

**ENABLE:** This input (active low) enables the chip outputs.

**RESET:** This pin (active low) resets the chip. During normal operation RESET = 1.

**SLEEP:** This pin (active low) puts the chip into the sleep mode for low-power consumption.

**DIR:** This pin controls the direction of rotation.

**OUT1A/B:** These are the connections for coil 1.

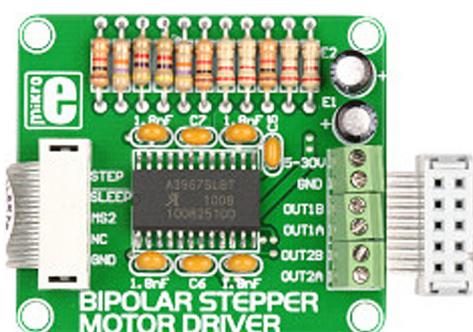
**OUT2A/B:** These are the connections for coil 2.

The mikroElektronika Bipolar Stepper Motor Driver board is used in this project. This board is plugged in to PORTB of the EasyPIC V7 development board. This board ([Figure 7.168](#)) has the following features:

- A 750 mA, 30-V output rating,
- Full-step and microstep modes,
- Direct interface to a bipolar stepper motor,
- Fully compatible with the EasyPIC V7 development board.

The following connections are made between the Bipolar Stepper Motor Driver board and PORTB:

PORTE	Driver Board
RC0	ENABLE
RC1	STEP
RC2	RESET
RC3	SLEEP
RC4	MS1
RC5	MS2
RC6	DIR



**Figure 7.168:** mikroElektronika Bipolar Stepper Motor Driver Board.



Figure 7.169: The 39HS02 Bipolar Stepper Motor.

The type of motor used in this project is the 39HS02 (Figure 7.169). This motor has the following features:

- A  $1.8^\circ$  step angle (200 steps for a complete revolution),
- A  $\pm 5\%$  step angle accuracy,
- A 0.6-A phase current,
- Four leads (Coil 1: brown + gray, Coil 2: orange + green).

### ***Project Program***

#### ***mikroC Pro for PIC***

The mikroC Pro for PIC program listing is shown in Figure 7.170 (MIKROC-BSTP.C). At the beginning of the program, the controller pins are defined by symbols. In the main program, the controller chip is enabled. Then, a loop is formed to send 2000 steps (10 revolutions) to the motor in one direction (DIR = 0). After a 5-s delay, the direction is changed (DIR = 1) and another 2000 steps are sent to the motor to rotate in the opposite direction. Note that a 1-ms delay is inserted between each step so that the speed of rotation is  $RPM = 1.8^\circ / 6 \times 1 \times 10^{-3} \text{ s} = 300$ .

#### ***MPLAB XC8***

The MPLAB XC8 program is shown in Figure 7.171 (XC8-BSTP.C). The program is similar to the mikroC version.

```
*****
BIPOLAR STEPPER MOTOR ROTATION
=====
```

In this project a 39HS model bipolar stepper motor is connected to PORTB of a PIC18F45K22 microcontroller through an A3967SLB type bipolar motor controller chip

The program rotates the motor clockwise 10 revolutions, stops for 5 s, and then rotates anticlockwise for 10 turns and stops.

39HS motor has step angle of 1.8 degrees. thus, a complete revolution requires 200 pulses (steps) to be sent to the motor. For 10 revolutions, we have to send 2000 pulses. The DIR input controls the direction of rotation.

Author: Dogan Ibrahim

Date: October 2013

File: MIKROC-BSTP.C

```
*****
#define ENABLE PORTB.RB0           // A3967SLB ENABLE input
#define STEP PORTB.RB1            // A3967SLB STEP input
#define RST PORTB.RB2             // A3967SLB RESET input
#define SLP PORTB.RB3             // A3967SLB SLEEP input
#define MS1 PORTB.RB4             // A3967SLB MS1 input
#define MS2 PORTB.RB5             // A3967SLB MS2 input
#define DIR PORTB.RB6             // A3967SLB DIR input
//
// This is the main program. The motor turns 10 revolutions clockwise with 1ms delay
// between each step. Then the motor stops for 5 seconds. The motor then rotates
// anticlockwise for 10 revolutions and then stops.
//
void main()
{
    unsigned int i;

    ANSELB = 0;                  // Configure PORTB as digital
    TRISB = 0;                  // Configure PORTB as digital
//
// A3967SLB chip configuration
//
    ENABLE = 0;                 // Enable the controller chip
    RST = 1;                    // Disable RESET
    SLP = 1;                    // Disable SLEEP
    MS1 = 0;                    // Full-step sequence
    MS2 = 0;
    STEP = 0;                   // Step = 0 to start with
//
// First rotate clockwise 10 revolutions
//
    DIR = 0;
    for(i = 0; i < 2000; i++)
{
```

**Figure 7.170: mikroC for the PIC Program.**

```

        STEP = 1;                                // Send STEP pulses
        STEP = 0;
        Delay_Ms(1);
    }

    Delay_Ms(5000);                           // Wait 5 s
//
// Now rotate anticlockwise 10 revolutions
//
    DIR = 1;
    for(i = 0; i < 2000; i++)
    {
        STEP = 1;                                // Send STEP pulses
        STEP = 0;
        Delay_Ms(1);
    }
    while(1);                               // End. Wait here forever
}

```

**Figure 7.170**  
cont'd

### Project 7.22—DC Motor Control Projects—Simple Motor Drive

DC motors are used in many industrial, commercial, and domestic applications. In this project, a DC motor is controlled by rotating it in either the clockwise or anticlockwise direction.

A simplified electrical circuit diagram of a DC motor is shown in [Figure 7.172](#).

The motor torque is proportional to the current through the motor:

$$T = K_T i. \quad (7.2)$$

The back emf is given by

$$V_e = K_e w, \quad (7.3)$$

Where  $w$  is the motor angular speed (radians per second). We can write the following formula for the motor circuit:

$$V - V_e = R i + L di/dt \quad (7.4)$$

or

$$V = R i + L di/dt + K_e w. \quad (7.5)$$

At the same time,

$$T = J dw/dt. \quad (7.6)$$

```
*****
BIPOLAR STEPPER MOTOR ROTATION
=====

In this project a 39HS model bipolar stepper motor is connected to PORTB of a PIC18F45K22
microcontroller through an A3967SLB type bipolar motor controller chip

The program rotates the motor clockwise 10 revolutions, stops for 5 s, and then
rotates anticlockwise for 10 turns and stops.

39HS motor has step angle of 1.8 degrees. thus, a complete revolution requires 200 pulses
(steps) to be sent to the motor. For 10 revolutions, we have to send 2000 pulses. The DIR
input controls the direction of rotation.

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

#define ENABLE PORTBbits.RB0 // A3967SLB ENABLE input
#define STEP PORTBbits.RB1 // A3967SLB STEP input
#define RST PORTBbits.RB2 // A3967SLB RESET input
#define SLP PORTBbits.RB3 // A3967SLB SLEEP input
#define MS1 PORTBbits.RB4 // A3967SLB MS1 input
#define MS2 PORTBbits.RB5 // A3967SLB MS2 input
#define DIR PORTBbits.RB6 // A3967SLB DIR input
// 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 is the main program. The motor turns 10 revolutions clockwise with 1ms delay
// between each step. Then the motor stops for 5 seconds. The motor then rotates
// anticlockwise for 10 revolutions and then stops.
// void main()
{
    unsigned int i;
```

**Figure 7.171: MPLAB XC8 Program.**

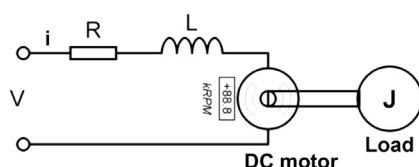
```

        ANSELB = 0;                                // Configure PORTB as digital
        TRISB = 0;                                // Configure PORTB as digital
        //
        // A3967SLB chip configuration
        //
        ENABLE = 0;                                // Enable the controller chip
        RST = 1;                                   // Disable RESET
        SLP = 1;                                   // Disable SLEEP
        MS1 = 0;                                   // Full-step sequence
        MS2 = 0;
        STEP = 0;                                  // Step = 0 to start with
        //
        // First rotate clockwise 10 revolutions
        //
        DIR = 0;
        for(i = 0; i < 2000; i++)
        {
            STEP = 1;                                // Send STEP pulses
            STEP = 0;
            __delay_ms(1);
        }

        Delay_Seconds(5);                          // Wait 5 s
        //
        // Now rotate anticlockwise 10 revolutions
        //
        DIR = 1;
        for(i = 0; i < 2000; i++)
        {
            STEP = 1;                                // Send STEP pulses
            STEP = 0;
            __delay_ms(1);
        }
        while(1);                                // End. Wait here forever
    }
}

```

**Figure 7.171**  
cont'd



**Figure 7.172: Simplified Electrical Circuit of a DC Motor, Where,  $R$  is the Motor Resistance;  $L$  is the Motor Inductance;  $J$  is the Motor Inertia;  $V_e$  is the Back Electromotive Force (emf).**

Thus, from Eqns (7.2) and (7.6),

$$I = J/K_T dw/dt. \quad (7.7)$$

Combining Eqns (7.5) and (7.7),

$$V = \frac{RJ}{K_T} \frac{dw}{dt} + \frac{LJ}{K_T} \frac{d^2w}{dt^2} + K_e w. \quad (7.8)$$

The inductance is negligible in small motors. If we remove the second-order inductance term from Eqn (7.8) we get

$$V = \frac{RJ}{K_T} \frac{dw}{dt} + K_e w. \quad (7.9)$$

Taking the Laplace transform of both sides, the relationship between the motor speed and applied voltage is simply given by

$$\frac{w(s)}{V(s)} = \frac{K_T}{K_T K_e + RJs}. \quad (7.10)$$

We can show the open-loop transfer function of the DC motor as in Figure 7.173.

The closed-loop motor transfer function is used in speed control applications. Figure 7.174 shows the closed-loop transfer function where the speed of the motor is sensed using either an optical encoder or a tachogenerator and is compared with the desired speed. The digital controller is usually a microcontroller or a PC that generates the control signals to drive the motor to obtain the required speed.

### **Project Description**

This project is about controlling the direction of rotation of a DC motor. Two push-button switches are used: one to control the direction and another one to start/stop the motor.

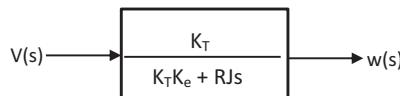


Figure 7.173: Open-loop Transfer Function of the DC Motor.

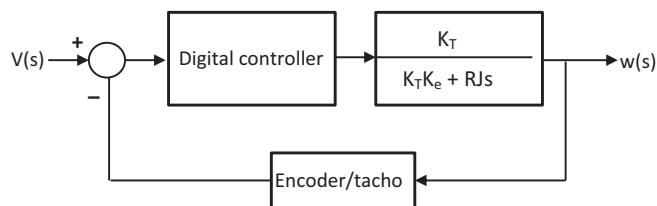


Figure 7.174: Closed-loop DC Motor Speed Control.

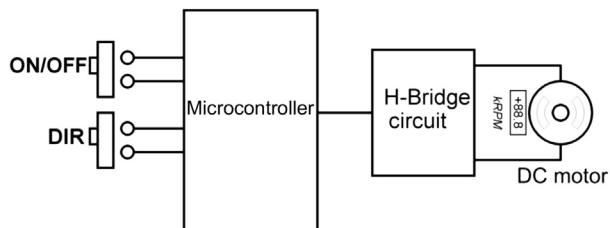


Figure 7.175: Block Diagram of the Project.

Figure 7.175 shows the block diagram of the project.

### Project Hardware

The circuit diagram of the project is shown in Figure 7.176. An H-bridge circuit is constructed from four MOSFET transistors, connected to PORTB of the microcontroller. The motor is controlled as shown in the following table:

ON/OFF Button	Direction Button	RB3	RB2	RB1	RB0	Motor State
0	X	0	0	0	0 (0x00)	OFF
1	0	0	1	0	1 (0x05)	Clockwise
1	1	1	0	1	0 (0x0A)	Anticlockwise

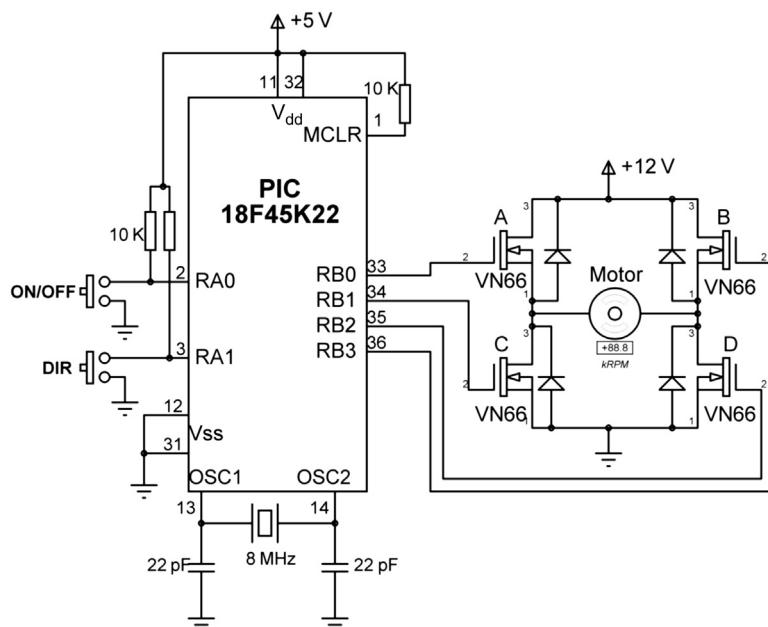


Figure 7.176: Circuit Diagram of the Project.

## Project Program

### mikro Pro for PIC

The mikroC Pro for the PIC program listing is shown in [Figure 7.177](#) (MIKROC-DCMTR1.C). Symbols ONOFF and DIR are assigned to I/O ports RA0 and RA1, respectively. The motor is controlled by the two buttons and rotates clockwise, anticlockwise, or stops.

```
*****
DC MOTOR CONTROL
=====

In this project a DC motor is connected to PORTB of a microcontroller via an H-bridge circuit,
constructed from 4 MOSFET transistors.

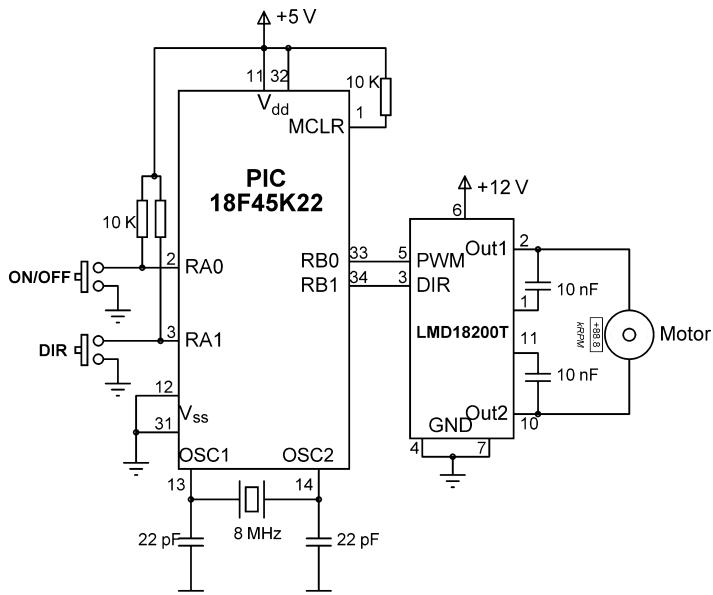
Two push-button switches are used to control the motor. Normally the motor rotates
anticlockwise. Pressing the DIR button changes the direction of rotation. Pressing the ON/OFF
button stops the motor.

Author: Dogan Ibrahim
Date: October 2013
File: MIKROC-DCMTR1.C
*****/

#define ONOFF PORTA.RA0
#define DIR PORTA.RA1
#define ON 1
#define ClockWise 0x05
#define AntiClockWise 0x0A
//
// Two buttons are used to control the motor movements: The motor turns clockwise,
// anticlockwise, or stops
//
void main()
{
    ANSELA = 0;                                // Configure PORTA as digital
    ANSELB = 0;                                // Configure PORTB as digital
    TRISB = 0;                                 // Configure PORTB as output
    TRISA = 3;                                 // Configure RA0 and RA1 as inputs

    for(;;)                                     // Do forever
    {
        if(ONOFF == ON)                         // The motor is normally ON
        {
            if(DIR == 0)
                PORTB = ClockWise;
            else
                PORTB = AntiClockWise;
        }
        else
            PORTB = 0;                          // If the ON/OFF button is pressed, stop the motor
    }
}
```

**Figure 7.177: mikroC Pro for PIC Program.**



**Figure 7.178:** Using the LMD18200T for DC Motor Control.

Instead of using four transistors, we could have used a motor controller chip, for example, LMD18200T (Figure 7.178). This chip is connected directly to the motor. For fixed speed applications, we can apply logic 1 to the PWM input. The DIR input controls the motor direction.

### **Project 7.23—A Homemade Optical Encoder For Motor Speed Measurement**

Optical encoders are used as sensors in motor speed and position control applications. There are several types of optical sensors available, such as incremental sensors, absolute sensors, and linear sensors. The encoder technology is not covered in this book and interested readers can search the Internet for further information and manufacturers' datasheets.

In this project, we will look at the design of an optical sensor to measure and display the speed of a motor. In the design, a round-shaped plate is attached to the motor shaft whose speed is to be measured. Holes are made on this plate at equal distances around the corner of the plate (Figure 7.179). A light source (e.g. infrared) emits light that passes through the holes as the plate rotates. At the other side of the plate, a light detector senses when a hole passes in front of it. This information is then passed to a microcontroller that counts the number of holes passing in front of the detector at a given time interval. This reading is then converted into motor speed.

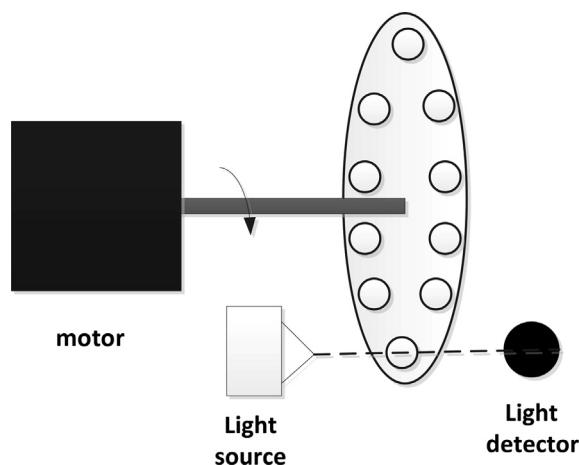


Figure 7.179: Simple Optical Encoder.

### Project Hardware

In this project, a small plate with eight holes is attached to the motor shaft (Figure 7.180). The accuracy is increased when the number of holes is increased. Most commercially available encoders provide  $\geq 200$  holes. An infrared light source and light detector pair are

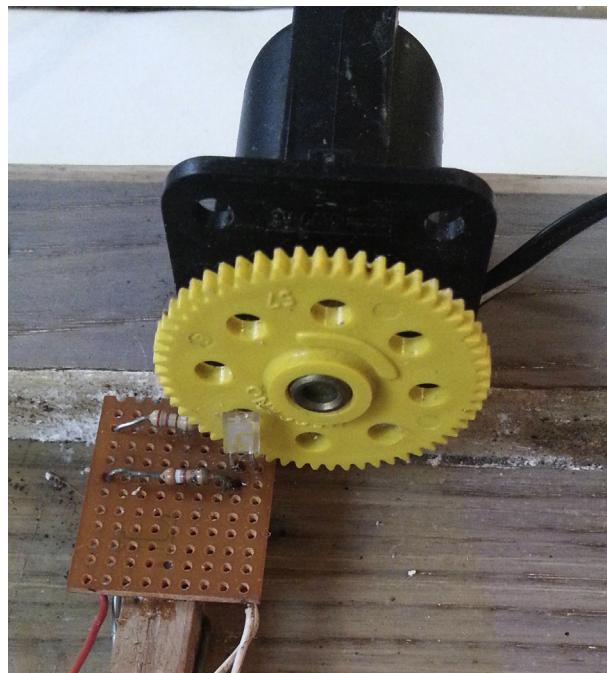
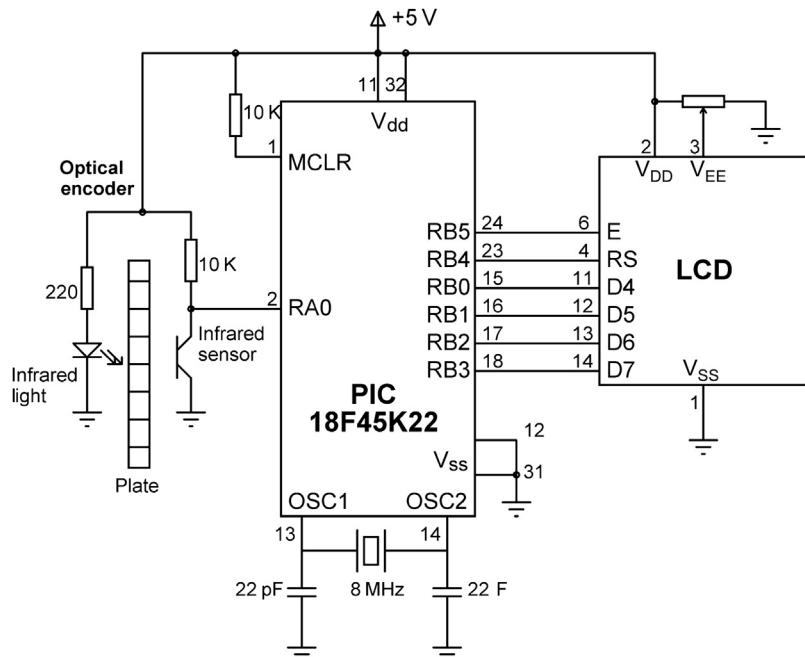


Figure 7.180: Homemade Optical Encoder Attached to a DC Motor.



**Figure 7.181: Circuit Diagram of the Project.**

used in this project as shown in [Figure 7.181](#). The output of the light detector is connected to pin RA0 of the microcontroller. The motor speed is displayed on the LCD.

### **Project Program**

#### *mikroC Pro for PIC*

In this project, Timer0 of the microcontroller is configured to generate interrupts at every 100 ms and the number of holes passing in-front of the detector are counted within this time interval. If  $n$  is the number of holes passing in 100 ms, then we have  $10n$  holes passing every second. If  $N$  is the number of holes on the encoder plate, then the plate makes  $10n/N$  revolutions every second, that is,

$$\text{Motor speed} = 10n/N \text{ revolutions per second}$$

In this project,  $N = 8$  and therefore

$$\text{Motor speed} = 10n/8 = 1.25n \text{ revolutions per second}$$

The speed is usually measured in revolutions per minute. Thus, multiplying the right-hand side by 60 we get

$$\text{Motor Speed (revolutions per minute)} = 60 \times 1.25n = 75n.$$

```
*****
OPTICAL ENCODER MOTOR SPEED MEASUREMENT
=====

In this project a homemade optical encoder is used. The encoder has 8 holes and is connected
to the shaft of DC motor. Infrared light source and detectors are used and connected to a
microcontroller to count the number of holes passing in-front of the detector. The speed is then
calculated in RPM and is displayed on the LCD.

Timer 0 is configured in 16-bit mode to generate interrupts at every 100 ms. The number of holes
passing in-front of the detector in 100 ms is counted and then the RPM is calculated as described in
the text.

Author: Dogan Ibrahim
Date: October 2013
File: MIKROC-ENCODER.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 ENCODER PORTA.RA0
unsigned int RPM, Cnt = 0;
unsigned char Txt[] = "RPM=    ";
unsigned char DspltCnt = 0;

void interrupt(void)
{
    TMROH = 0x3C;                                // Reload TMRO for 100ms interrupts
    TMROL = 0xB0;
    RPM = 75*Cnt;                                 // Speed in RPM
    IntToStr(RPM, Txt+4);                         // Convert to string
    Cnt = 0;
    DspltCnt++;
    if(DspltCnt == 10)                            // 1 s ?
    {
        DspltCnt = 0;
        Lcd_Cmd(_LCD_CLEAR);                      // Clear LCD
        Ltrim(Txt+4);                           // Remove leading spaces
    }
}
```

Figure 7.182: mikroC Pro for PIC Program.

```

        Lcd_Out(1,1,Txt);                                // Display speed every second
    }
    INTCON.T0IF = 0;                                  // Re-enable TMRO interrupts
}

// Two buttons are used to control the motor movements: The motor turns clockwise,
// anticlockwise, or stops
//
void main()
{
    ANSELA = 0;                                     // Configure PORTA as digital
    ANSELB = 0;                                     // Configure PORTB as digital
    TRISB = 0;                                      // Configure PORTB as output
    TRISA = 1;                                       // Configure RA0 as input

    Lcd_Init();                                     // Initialize LCD
//
// Configure TMRO is 16 bit mode to generate interrupts every 100 ms
//
    TOCON = 0x81;                                    // 16-bit mode, prescaler = 4
    TMROH = 0x3C;                                    // Load 0x3C0 for 100 ms interrupts
    TMROL = 0xB0;
    INTCON = 0xA0;                                    // Enable TMRO and global interrupts

    while(1)
    {
        while(ENCODER == 0);                         // Wait if 0
        Cnt++;                                       // Increment encoder count
        while(ENCODER == 1);                         // Wait if 1
    }
}

```

**Figure 7.182**  
cont'd

Therefore, by counting the number of holes passing in-front of the detector in 100 ms we can easily find the motor speed in revolutions per minute.

In some applications, the motor speed is measured in radians per second. Since each rotation is  $2\pi$  radians, the motor rotation in 1 s will be  $10n \times 2\pi/N$ . Thus, the motor speed in radians per second is given by

$$\text{Motor speed (radians/second)} = 10n \times 2\pi/N.$$

In this project,  $N = 8$  and therefore,

$$\text{Motor speed (radians per second)} = 7.85n.$$

The mikroC Pro for PIC program listing is shown in [Figure 7.182](#). The main program initializes the LCD, configures the timer, and then enters an endless loop. Inside this loop, the number of holes passing in front of the detector are counted and stored in variable Cnt.

Timer 0 is configured in the 16-bit mode to generate interrupts at every 100 ms. The value to be loaded into the timer registers for 100 ms (100,000 µs) interrupt is found as follows:

$$\text{TMR0H:TMR0L} = 65536 - 100,000/(4 \times \text{TxPre-scaler}).$$

With a clock frequency of 8 MHz ( $T = 0.125 \mu\text{s}$ ) and the Prescaler value of 4,

$$\text{TMR0H:TMR0L} = 65536 - 100,000/(4 \times 0.125 \mu\text{s}) = 15,536$$

that is,  $\text{TMR0H} = 0x3C$  and  $\text{TMR0L} = 0xB0$ .

Inside the ISR, the motor speed is calculated in revolutions per minute. The speed is displayed every second (every 10 times we enter the ISR) on the LCD. The motor used in this project had a speed of 5800 RPM, which is displayed as follows:

$$\text{RPM} = 5800$$

### **Project 7.24—Closed-Loop DC Motor Speed Control—On/Off Control**

This project is about closed-loop speed control of a DC motor using a microcontroller. In the project, an optical encoder is used to detect the motor speed and feedback is applied to achieve the desired speed. The motor control signal is in the form of a PWM waveform.

PWM waveform is frequently used in power control applications, such as motor control, pump control, and heating control. By changing the duty cycle (ON time), we can control the average voltage (or power) applied to the load.

If  $V_i$  is the amplitude of the PWM signal, the average voltage supplied to the load is given by

$$\text{Average voltage} = \frac{t_{\text{ON}}}{t_{\text{ON}} + t_{\text{OFF}}} V_i = \frac{t_{\text{ON}}}{T} V_i.$$

The relationship between  $t_{\text{ON}}$  and the average load voltage is linear, and as  $t_{\text{ON}}$  changes from 0 to T (0–100% duty cycle), the average voltage delivered to the load changes from 0 to  $+V_i$ . Thus, we can control  $t_{\text{ON}}$  time to control the motor voltage and hence the motor speed.

In this project, the built-in PWM module of the microcontroller is used to generate the PWM signal. We can choose the PWM frequency from a few kilohertz to  $\geq 10$  kHz. Here, we chose 10 kHz ( $T = 0.1$  ms). We can find the register values to generate the required PWM waveform. With an 8-MHz clock:

$$\text{PR2} = \frac{\text{PWM period}}{\text{TMR2 prescale value} * 4 * \text{T}_{\text{OSC}}} - 1$$

or

$$PR2 = \frac{0.1 \times 10^{-3}}{4 * 4 * 0.125 \times 10^{-6}} - 1 = 49 \text{ or } 0x31 \text{ in hexadecimal.}$$

Also,

$$CCPR1L : CCP1CON\langle 5 : 4 \rangle = \frac{\text{PWM pulse width}}{\text{TMR2 prescale value} * T_{osc}}$$

or

$$CCPR1L : CCP1CON\langle 5 : 4 \rangle = \frac{\text{Duty cycle} \times 10^{-3}}{4 * 0.125 \times 10^{-6}} = 2000 \times \text{Duty cycle},$$

where the Duty cycle is in milliseconds.

Thus, for a Duty cycle of 0.001 ms, CCPR1L:CCP1CON<5:4> = 2, which gives the average voltage as

$$\text{Average voltage} = \frac{0.001}{0.1} V_i = 0.01 V_i$$

For 100% Duty cycle, Duty cycle = 0.1 ms and

$$CCPR1L:CCP1CON\langle 5:4 \rangle = 200 \text{ which gives the average voltage equal to } V_i.$$

As a summary, as we change the register value from 2 to 200, the average voltage applied to the motor will change linearly from 0.01V<sub>i</sub> to V<sub>i</sub>. Combining the above equations, we can write an expression for the average load voltage as follows:

$$\text{Average voltage} = \frac{CCPR1L : CCP1CON \langle 5 : 4 \rangle}{2000 \times 0.1} V_i$$

or

$$\text{Average voltage} = \frac{CCPR1L : CCP1CON\langle 5 : 4 \rangle}{200} V_i.$$

The value loaded into CCPR1L:CCP1CON<5:4> changes from 2 to 200.

The CCPR1L:CCP1CON<5:4> register combination is 10 bits wide. Assuming that the number to be loaded into the register pair is integer N, the value to be loaded into CCPR1L can be found by right shifting N by 2 bits. The number to be loaded into bits 5 and 4 of CCP1CON can be found by taking the two least significant bits of N and left shifting by 4 bits. Bits 2 and 3 of CCP1CON are then set to 1 for

PWM operation. The following program code shows how to load the PWM register pair:

```
CCPR1L = N >> 2;  
Temp = N & 0x03;  
Temp = Temp << 4;  
CCP1CON = Temp | 0x0C;
```

In this project, the homemade optical encoder (with eight holes) described in the previous project is used. The encoder data are read every 10 ms. If  $n$  is the number of holes passing in front of the detector in 0.01 s, then  $100n$  holes pass in 1 s. If  $N$  is the number of holes on the encoder, the motor speed is given by

$$\text{Motor speed} = 100n/N \text{ revolutions per second}$$

With  $N = 8$ ,

$$\text{Motor speed} = 100n/8 = 12.5 \text{ revolutions per second}$$

or

$$\text{Motor speed (revolutions per minute)} = 750n$$

The timer interrupt TMR0 is set to generate interrupts every 10 ms, and the motor speed is calculated inside the ISR.

Assuming a prescaler value of 128- and 8-MHz clock ( $T = 0.125 \mu\text{s}$ ), the TMR0 register value for 10-ms interrupts can be found from

$$\text{TMR0L} = 256 - 10000/(4 \times 0.125 \times 128) = 100.$$

### ***Project Hardware***

The circuit diagram of the project is shown in [Figure 7.183](#). The speed of the motor is fed back via the optical encoder to RA0 input of the microcontroller. The PWM output of the microcontroller drives the motor through a MOSFET transistor. An LCD is connected to PORTB to display the actual motor speed in revolutions per minute.

### ***Project Program***

#### *mikroC Pro for PIC*

The mikroC Pro for the PIC program listing is shown in [Figure 7.184](#) (MIKROC-MTRONOFF.C). At the beginning of the main program, I/O ports are configured, and the LCD is initialized. Then, the PWM module is configured to generate pulses with a period of 9.1 ms. Timer 2 is used to provide clock to the PWM module, and Timer 0 is configured to generate interrupts at every 10 ms. The desired speed is set to

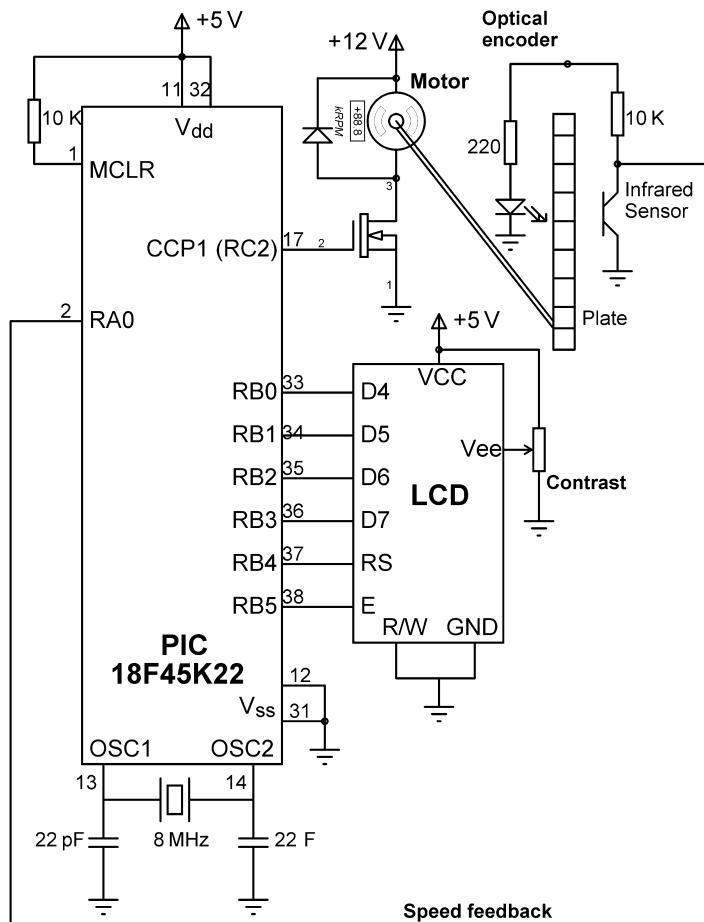


Figure 7.183: Circuit Diagram of the Project.

2000 RPM with a dead band of 100, that is, it is required to keep the speed between  $2000 \pm 100$  RPM. The remainder of the main program counts the encoder pulses.

Inside the ISR the timer register TMR0L is reloaded, the motor speed is calculated and the ON/OFF control action is applied using the following program code:

```

/* IF SLOW */
if(RPM < (DesiredRPM - Deadband))
{
    SetSpeed(200);      // Rotate motor FAST
}
/* IF FAST */
if(RPM > (DesiredRPM + Deadband))
{
    SetSpeed(2);        // Rotate motor SLOW
}

```

```
*****
CLOSED-LOOP ON/OFF MOTOR SPEED CONTROL
=====
```

In this project a DC motor is controlled using feedback and ON/OFF type simple control. The motor speed is measured using a homemade optical encoder.

If the motor speed is higher than the desired speed than the motor supply is cut. If on the other hand the motor speed is lower than the desired speed than full power is applied to the motor.

The motor is connected to the microcontroller through a VN66AFD type MOSFET transistor. The PWM module of the microcontroller is used to provide control pulses to the motor. An LCD is connected to PORTB and the LCD shows the actual motor speed every second.

The optical encoder is connected to RA0 input of the microcontroller. The motor is driven from the CCP1 (RC2) output.

Homemade optical encoder is used. The encoder has 8 holes and is connected to the shaft of DC motor. Infrared light source and detectors are used and connected to a microcontroller to count the number of holes passing in-front of the detector. The speed is then calculated in RPM and is displayed on the LCD.

Author: Dogan Ibrahim

Date: October 2013

File: MIKROC-MTRONOFF.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 ENCODER PORTA.RA0
```

unsigned int Cnt = 0;

unsigned int RPM, DesiredRPM, DeadBand;

unsigned char Txt[] = "RPM= ";

unsigned char DsplyCnt = 0;

```
//
```

// This function sets the motor speed

```
//
```

**Figure 7.184: mikroC Pro for the PIC Program.**

```

void SetSpeed (unsigned char N)
{
    unsigned char Temp;

    CCPR1L = N >> 2;
    Temp = N & 0x03;
    Temp = Temp << 4;
    CCP1CON = Temp | 0x0C;
}

// This is the Timer0 interrupt service routine. The program jumps here every 10ms.
// The actual motor speed is compared with the desired speed and either the motor is
// operated with full voltage (maximum speed), or with low voltage (minimum speed). The
// motor speed is displayed on the LCD.
//
void interrupt(void)
{
    TMROL = 100;                                // Reload Timer register
    RPM = 750*Cnt;                             // Actual motor speed (in RPM)
    Cnt = 0;
    /* IF SLOW */
    if(RPM < (DesiredRPM - Deadband))
    {
        SetSpeed(200);                         // Rotate motor FAST
    }

    /* IF FAST */
    if(RPM > (DesiredRPM + Deadband))
    {
        SetSpeed(2);                           // Rotate motor SLOW
    }
    DsplyCnt++;
    if(DsplyCnt == 100)                         // 1 second (10msx100=1s) ?
    {
        DsplyCnt = 0;
        Lcd_Cmd(_LCD_CLEAR);                  // Clear LCD
        IntToStr(RPM,Txt+4);                 // Convert into string
        Ltrim(Txt+4);                       // Remove leading spaces
        Lcd_Out(1,1,Txt);                   // Display speed every second
    }

    INTCON.T0IF = 0;                            // Re-enable TMRO interrupts
}

// The desired RPM is set to 2000 and the DeadBand is set to 100 RPM
//
void main()
{
    ANSELA = 0;                                // Configure PORTA as digital
}

```

**Figure 7.184**

cont'd

```
ANSELB = 0;                                // Configure PORTB as digital
ANSELC = 0;                                // Configure PORTC as digital
TRISA = 1;                                 // Configure RA0 as input
TRISB = 0;                                 // Configure RB0 as input

LCD_Init();                                  // Initialize LCD
//
// Configure the PWM module
//
T2CON = 0x05;                               // Set Timer 2 ON and prescaler to 4
PR2 = 0x31;                                 // Load Timer 2 PR2 register with 49
CCPR1L = 0;                                // Motor OFF to start with
TRISC = 0;                                 // Configure CCP1 (RC2) as output
CCP1CON = 0x0C;                            // Enable PWM module
CCPTMRS0 = 0;                             // Use Timer 2 for the PWM module (CCP1)
//
// Configure TMRO in 8 bit mode to generate interrupts every 10ms
//
TOCON = 0xC6;                               // 8-bit mode, prescaler = 128
TMROL = 100;                                // Timer value for 10ms timer overflow
INTCON = 0xA0;                            // Enable TMRO interrupts

DesiredRPM = 2000;                           // Desired RPM
DeadBand = 100;                            // Deadband range

while(1)
{
    while(ENCODER == 0);                     // Wait if 0
    Cnt++;                                 // Increment encoder count
    while(ENCODER == 1);                     // Wait if 1
}
}
```

**Figure 7.184**  
cont'd

Function SetSpeed loads the CCPR1L and CCP1CON register of the PWM module accordingly. The LCD displays the actual motor speed every second.