Embedded Systems: A hands-on Approach

N. Natarajan
Department of Electrical and Computer Engineering
University of Michigan-Dearborn
4901 Evergreen Road
Dearborn-48128
nnarasim@umich.edu

Printed on Sunday 17th February, 2013 at 16:42 GMT
Latest correction: Added material from my Google docs. It is at the end to facilitate printing

1This is an evolving document! Do not print but check the date every so often.
Contents

1 Preface 1

2 Getting Started 3
   2.1 What is an Embedded system 3
   2.2 How to get started 3
   2.3 The Hello World of Embedded Systems 4
      2.3.1 Requirements: 4
      2.3.2 Steps 4
   2.4 Downloading your code to PIC16F690 7
   2.5 What you should know about MikroC 8

3 Digital I/O 13
   3.1 Overview 13
   3.2 Tri-state 13
   3.3 Performing Digital I/O 14
   3.4 LCD panel 16
      3.4.1 Driver protocol 16
      3.4.2 Sample program 18

4 Clocks and timing 23
   4.1 The System Clock 23
   4.2 Timer Systems 23
      4.2.1 Event counting 25
      4.2.2 Pre scalers and Post Scalers 25
   4.3 Time Delays 26

5 Analog Input 29
   5.1 Analog to Digital Conversion 29
CHAPTER 5: Analog to Digital Conversion

5.1 Basics ................................................. 29
5.1.2 ADC in PIC16F690 and PIC16F1507 ........ 29
5.1.3 Procedure for reading analog input in PIC16F690 . 31
5.1.4 Example: Reading the temperature using MCP9701 . 32

CHAPTER 6: Character Input/Output 37

6.1 Asynchronous Serial Communication ............... 37
6.2 Printing .................................................. 38
   6.2.1 A sample program ................................ 40
6.3 Reading from keyboard .................................. 42
6.4 Numerical Input ......................................... 44
   6.4.1 Pushing back a character .......................... 44
   6.4.2 Decimal Input ...................................... 46
   6.4.3 Hexadecimal Input .................................. 47

CHAPTER 7: Pulse Width Modulation 49

7.1 Why Pulse Width Modulation? ......................... 49
7.2 Generating PWM signals ................................ 50
7.3 Hardware PWM generation in PIC16F690 ............ 51
   7.3.1 General Principles .................................. 51
   7.3.2 Implementation ..................................... 52
   7.3.3 Example Program ................................... 52
7.4 Hardware PWM generation in PIC16F1507 ............ 54
   7.4.1 Example program ................................... 55
7.5 Dual Motor Control .................................. 57
   7.5.1 Half bridge chip: SN754410 ......................... 57
   7.5.2 Controlling the motor .............................. 58
   7.5.3 Chip Independent programming .................. 58

CHAPTER 8: Serial Communications 65

8.1 SPI: Serial Peripheral Interface Bus ............... 65
   8.1.1 Communication Protocol ............................ 66
   8.1.2 An Example .......................................... 67
   8.1.3 Dual Digital Potentiometer ......................... 70
8.2 I2C: Inter-Integrated Circuit Bus ..................... 76
   8.2.1 Electrical Connection ................................ 77
   8.2.2 Communication Protocol ............................ 77
   8.2.3 An Example .......................................... 79
CONTENTS

8.2.4 Using 24LCXX ........................................... 82

9 Applications: Smart Serial Peripherals 91

9.1 Smart Serial Peripherals ................................. 91
  9.1.1 Hardware of Software UART? ......................... 92

9.2 Serial LCD display ...................................... 93
  9.2.1 Circuit Schmeatic .................................. 94
  9.2.2 Code ................................................. 95

9.3 Serial Motor Controller ............................... 99
  9.3.1 Command Structure ................................ 100
  9.3.2 Implementation details ............................ 101
  9.3.3 Schematic and Code ............................... 101

Appendices 109

A Elements of Embedded C 111
  A.1 Introduction .......................................... 111
  A.2 Number Representations .............................. 112
  A.3 Integers are NOT always integers! .................. 112
  A.4 Ports and Digital Input-Output (DIO) ............... 113
    A.4.1 Input pins ...................................... 114
    A.4.2 Output pins .................................... 115
  A.5 Bit Fields .......................................... 117
  A.6 Use of Macros ........................................ 118

B Using PicKit2 UART tool 121
  B.1 Using UART tool .................................... 121
List of Figures

2.1 Start the process by selecting New Project .......................... 5
2.2 Select the MCU and the processor speed ......................... 6
2.3 Select all the libraries ............................................ 6
2.4 Make sure you select the option to set configuration bits . ... 7
2.5 Pay attention to Oscillator Selection, Watchdog Timer, MCLR, Pin Function, the MCU name and Frequency .................... 8
2.6 Basic circuit to get started ....................................... 9
2.7 Opening screen when Pickit2 software starts ..................... 10
2.8 Writing the program to the processor and powering it on ....... 11

3.1 Experimental setup to perform Digital I/O ....................... 15
3.2 Connecting an LCD panel to a PIC16F690 ....................... 18

5.1 Schematics for testing MCP9701 using UART tool to view output 33
5.2 Typical output. Because of an accidental misconnection, the sensor got really hot! ........................................ 34

6.1 Output of the sample printing program ................................ 40

7.1 A typical PWM waveform ......................................... 49
7.2 Circuit to test PWM code for PIC16F690 ........................ 53
7.3 Circuit to test PWM code for PIC16F1507 ....................... 55
7.4 Dual Motor controller. Shown with PIC16F1507 but PIC16F690 can also be used ........................................ 57

8.1 A typical SPI bus consisting of 1 master and 3 slaves. Image source: Wikipedia .................................................. 67
8.2 Master slave connection between two PIC16F690s ............... 68
8.3 SPI connection to MCP4241. As the wipers are moved, we produce a voltage between 0 and 5 volts at the two wipers.

8.4 The voltage at the wiper shown on a dual oscilloscope. Note that as one voltage increases, the other decreases.

8.5 LCD panel controlled via an I2C port expander.

8.6 Schematic to use 24LCXX.

9.1 Serial LCD display device.

9.2 Serial LCD in action.

9.3 Basic architecture of the serial motor controller.

9.4 Schematic for a serial motor controller.

B.1 Typical connection for using PicKit2 Usart-tool.

B.2 Select UART tool from the main menu.

B.3 UART tool screen. Wiring is also shown at the bottom left.
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Important Project Settings</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Tri-state definitions</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>TRIS/PORT pairing</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Common commands to use with LCD</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Selecting System Frequency: PIC16F690</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Selecting System Frequency: PIC16F1507</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Configuring clock source for timers: PIC16F690</td>
<td>25</td>
</tr>
<tr>
<td>9.1</td>
<td>Commands for a serial LCD device</td>
<td>94</td>
</tr>
<tr>
<td>9.2</td>
<td>Command Structure for Serial Motor Controller</td>
<td>100</td>
</tr>
<tr>
<td>B.1</td>
<td>Wiring PicKit2 to PIC16F690 for use as UART tool</td>
<td>121</td>
</tr>
</tbody>
</table>
## Listings

<table>
<thead>
<tr>
<th>Chapter</th>
<th>File Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>hello.c</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>digital-io.c</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>lcd.c</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>tmr0delay.c</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>mcp9701.c</td>
<td>33</td>
</tr>
<tr>
<td>6.1</td>
<td>printing.c</td>
<td>40</td>
</tr>
<tr>
<td>6.2</td>
<td>reading.c</td>
<td>43</td>
</tr>
<tr>
<td>7.1</td>
<td>pwmtest.c</td>
<td>53</td>
</tr>
<tr>
<td>7.2</td>
<td>pwm1507.c</td>
<td>55</td>
</tr>
<tr>
<td>7.3</td>
<td>dual-motor.c</td>
<td>59</td>
</tr>
<tr>
<td>8.1</td>
<td>spimaster.c</td>
<td>68</td>
</tr>
<tr>
<td>8.2</td>
<td>spislave.c</td>
<td>69</td>
</tr>
<tr>
<td>8.3</td>
<td>mcp4241.c</td>
<td>70</td>
</tr>
<tr>
<td>8.4</td>
<td>port-expander-i2c-lcd.c</td>
<td>79</td>
</tr>
<tr>
<td>8.5</td>
<td>24LC04.c</td>
<td>85</td>
</tr>
<tr>
<td>9.1</td>
<td>serial-lcd.c</td>
<td>95</td>
</tr>
<tr>
<td>9.2</td>
<td>serial-motor-controller.c</td>
<td>101</td>
</tr>
</tbody>
</table>
Chapter 1

Preface

The idea for this book originated from a course I have been teaching since 2007. The course itself grew out of the realization that most electrical engineering students graduated with minimal experience with embedded systems. For most students, this usually was a course in microprocessors where the emphasis was on learning the inner workings of a microprocessor. To give the students a hands-on experience, the course ECE 4951 was developed. The central theme of the course was designing embedded systems and was meant to be a precursor to the senior design course. One of the challenges of teaching an open-ended course is the lack of a good textbook. This book is an attempt to address this need.

Although this book is meant to be used as a textbook, the presentation is in the form of a (one way) conversation which I find is more appealing. The central philosophy of the book is learning through doing. Do not expect to find theoretical arguments or generalization. Also, to make the book useful for a wide audience, I have kept programming complexity to a minimum. Throughout the book, I have marginal warning notes as shown here. This, I hope, would be helpful.

Target Audience: Who is the target audience for the book? If you have never written a computer program of any kind, you will initially find the material very difficult. The level of programming background expected of you is an introductory programming course in any language that exposes you to the basics. All the programs in this book are written in what is commonly referred to as embedded-C. Embedded-C is not really different from standard C. The term embedded-C is used to designate that part of C that is of most
use to embedded systems programmer. This is a small part of the C language and a quick introduction can be found in appendix A.

Secondly, you need to have some background in electronics. At a minimum, you need to know how to use a solder-less breadboard, wire circuits based on a schematic, read data sheets.

Minimum requirements:

1. PIC microprocessor: PIC16F690-I/P or PIC16F690-E/P. Retail cost: $2. These two chips are the PDIP version. Best to have 4 chips for a variety of projects.

2. PIC programmer: PicKit2. Retail cost: $35

3. Personal computer running Microsoft Windows with at least one free USB port.

4. Solder-less breadboard. Retail cost: $5. Best to have a few.

5. Wire cutters

6. Wires: Best to use AWG22 or AWG24 wires. AWG20 wires can damage the breadboard.

7. Electronic components

   (a) Resistors: You will be using 220 or 330 ohms, 2.2K ohms, 10K ohms.

   (b) LEDs: You will need quite a few of these.

   (c) LED bar graph: This is a conveniently packaged 10 LEDs in one unit.

   (d) Bussed Resistor array: This is a conveniently packaged set of 5 to 10 resistors that are internally connected to make circuits simpler to build.
Chapter 2

Getting Started

2.1 What is an Embedded system

TBD

2.2 How to get started

To get started in embedded systems, one needs a microcontroller\(^1\) or a micro. Once a micro has been selected, one needs to decide on the programming language to use. Here, the choice is often between assembly language and the C language. If the choice is assembly language, then one has to select an appropriate assembler and if it is any high level language, one has to select a suitable compiler\(^2\). The choice of compiler is largely determined by cost, ease of use, how extensive the library comes standard. One also needs some way to load the programs on to the micro. This is done with a device called a programmer. The programmer that is chosen must be compatible with the micro and also the system that will be used to develop programs.

This book deals with mid-level micro controller from Microchip, Inc, in particular the PIC16F690. The programs will all be written in the C language. As with all compilers targeting embedded system developers, C compilers tend to take liberties with the ANSI standards for the C language. Two

\(^1\)The terms \textit{microprocessor} and \textit{micro controller} are used interchangeably since the distinction is often irrelevant. In this book the neutral term micro is used.

\(^2\)Another valid choice is to use Java and run Java on the micro using a suitable Java virtual machine.
popular choice of C-compilers for the PIC16F690 are the MikroC and the HI-TECH C compilers. The choice of MikroC is dictated by the extensive library that is provided for free, while HI-TECH C is the standard compiler that Microchip supports. Because of differences in the two compilers, this book will contain two versions of most programs. There are a number of excellent programmers for the PIC16F690. This book describes the use of PicKit2. This is a low cost programmer that uses USB connection and there is support for both Windows operating system as well as Linux operating system. This book deals mostly with using the programmer in a Microsoft Windows machine.

2.3 The Hello World of Embedded Systems

2.3.1 Requirements:

Before proceeding further, make sure you have a PIC16F690, two light emitting diodes (LEDs), prototyping board, wires, two 220 ohm resistors, PicKit2 programmer.

2.3.2 Steps

1. Download and install drivers for PicKit2. The CD that accompanies PicKit2 is often out of date so it is best to get the latest version from Microchip.

2. Download and install the MikroC compiler. Sometimes, MikroC compiler has permission issues with Windows. So it is best to install the compiler in the public documents folder.

3. On the prototyping area, connect the LEDs and resistors to RC0 (pin #2) and RC1 (pin #1).

4. Run MikroC and go through the following steps to create a new project called hello:

   (a) From the project menu, select new project as shown in figure 2.1.
(b) Go through the series of dialog boxes. Of interest is the choice of the MCU in figure 2.2, selection of all the libraries 2.3, selecting the option to set configuration bits 2.4 and setting configuration bits 2.5. See table tbl:project for important settings.

5. Enter the program shown in listing 2.1 and select build from the main menu.

Listing 2.1: hello.c

```c
/*
 * This program toggles PORTC pins 0 and 1
 */
```
CHAPTER 2. GETTING STARTED

Figure 2.2: Select the MCU and the processor speed.

Figure 2.3: Select all the libraries.

Table 2.1: Important Project Settings

<table>
<thead>
<tr>
<th>Item</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oscillator Selection</td>
<td>INTIOSCIO</td>
</tr>
<tr>
<td>Watchdog Timer</td>
<td>Disabled</td>
</tr>
<tr>
<td>MCLR Pin Function</td>
<td>Disabled (preferable)</td>
</tr>
<tr>
<td>MCU Name</td>
<td>P16F690 (or P16F1507)</td>
</tr>
<tr>
<td>Osc. Frequency</td>
<td>4.000 for P16F690, 0.5 for P16F1507</td>
</tr>
</tbody>
</table>

Target compiler: MikroC

Compiler Specifics:
Uses delay function to introduce delay in msec.

```c
/*
 void main()
 {
   TRISC = 0xFF - 3; // Set data direction
   while (1) {
     PORTC = 1;
     delay_ms(500);
     PORTC = 2;
     delay_ms(500);
   }
 }
```
2.4. **DOWNLOADING YOUR CODE TO PIC16F690**

1. First wire the circuit shown in figure 2.6.

2. Start the PicKit2 program. It should find the programmer and the chip.

```c
/*
 * Notes
 * TRISC determines the direction of the port pins
 * 0xFF has the binary pattern 11111111
 * 3 has the binary pattern 00000011
 * 0xFF-3 has the binary pattern 11111100
 * PORTC is the data
 * 1 has the binary pattern 00000001
 * 2 has the binary pattern 00000010
 */
```
CHAPTER 2. GETTING STARTED

Figure 2.5: Pay attention to Oscillator Selection, Watchdog Timer, MCLR Pin Function, the MCU name and Frequency.

as shown in figure 2.7

3. Use the Auto import and Write button to download your hex file that the compiler creates as shown in 2.8

2.5 What you should know about MikroC

There are a number of ways (some minor and some not so minor) in which MikroC deviates from standard C and you should be aware if it. The following is a short list that I find useful to know.

1. You do not need to include any device specific header file. This is done automatically for you by based on the device you selected.

2. Identifiers are not case sensitive. Thus, UART1Write is same as uart1_write are equivalent.

3. Bit fields are integrated into the language. Thus, if you want bit #3
2.5. WHAT YOU SHOULD KNOW ABOUT MIKROC

of something, you can write something.F or something.B. This is part of the language.

4. MikroC introduces a data type called sbit to declare specific bit. This is mainly used when you want your program to configure the hardware to meet the requirements of a library function. In these cases, just follow what the help pages tell you!

One aspect of the C language that always throws students off is declaring variables at any place in the code. The language standard says that variables etc should be declared at the top of a block, i.e. immediately following the open brace. Almost all modern compilers let you break this rule but MikroC does not.
Figure 2.7: Opening screen when Pickit2 software starts.
Figure 2.8: Writing the program to the processor and powering it on.
Chapter 3

Digital I/O

3.1 Overview

The microprocessor receives its digital inputs from the i/o pins on the chip. For example, in the case of PIC16F690 or PIC16F1507 pins 2 to 19 are all digital i/o pins. Pin 0 is power and pin 20 is ground. The pins themselves are organized into ports which are alphabetically enumerated as PORTA, PORTB and PORTC. Each port has a total of 8 bits, though not all bits may be available at the i/o pins. For example, in both PIC16F690 and PIC16F1507 only 4 bits of PORTA and four bits of PORTB are available.

Most i/o pins are multifunction pins. This means that depending on what your program does, the pin may get taken over by other modules and may not be available for digital i/o. This is especially true of ADC convertor described in chapter 5. Always make sure you turn off the ADC covertor using one of the following constructs:

\[
\text{ANSEL=ANSELH=0; //Turn off ADC in PIC16F690}
\]

or

\[
\text{ANSELA=ANSELB=ANSELC=0; //Turn off ADC in PIC16F1507}
\]

Failure to do this has caused many programs to fail for no apparent reason!

3.2 Tri-state

All digital i/o pin with one exception can be in one of three states as shown in table 3.1. The behavior of the pin is two registers, the TRIS and PORT.
Table 3.1: Tri-state definitions

<table>
<thead>
<tr>
<th>Input (High impedance or High-Z)</th>
<th>Input mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output 1 (Low-Z, output high)</td>
<td>Output mode. Pin is held high.</td>
</tr>
<tr>
<td>Output 0 (Low-Z, output low)</td>
<td>Output mode. Pin is held low.</td>
</tr>
</tbody>
</table>

Table 3.2: TRIS/PORT pairing

<table>
<thead>
<tr>
<th>TRIS Bit</th>
<th>PORT Bit</th>
<th>Pin State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Depends</td>
<td>This is High-Z state</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Output is held low (close to zero volts)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Output is held high (close to 5 volts)</td>
</tr>
</tbody>
</table>

Each I/O pin is associated with a bit in these two registers and in data sheet the pin is identified by the prefix R. For example, pin #2 on the chip is designated as RA5. Thus the pin is associated with bit #2 of TRISA and PORTA. The association is shown in table 3.2. When the TRIS bit is 1, the value of the PORT bit depends on the external circuitry connected to the pin. If the circuitry makes the voltage of the pin low (well below 2.5V) then the port bit will be 0 while of the voltage is well above 2.5 volts, the bit the port bit will be 1. When the voltage is close to 2.5 volts the value of the port bit is unpredictable.

3.3 Performing Digital I/O

This short experiment will illustrate the use of Digital I/O. Connect a push-button and an LED as shown in figure 3.1. If a push-button switch is not available, you can use a wire that you can connect to and disconnect from the ground. In this experiment, RC5 will be an output pin and the program will turn the LED on and off at some rate. If the switch is closed, then the rate would be speeded up. Note that when the switch is open, the 10K resistor will connect the pin RC4 to the 5V input. When the switch is closed, the
voltage at the pin will go to zero. The 10K resistor is there to prevent a dead short from power to ground. *Always make sure that your switches do not create a dead short between power and ground!* Execute the code shown in Listing 3.1. In the code, it is important that you turn off the ADC. *If you do not turn off the ADC then ADC will prevent you from reading the LED pin and it will always read as zero.* Try the program without turning off the ADC. Make sure you cycle power on the PIC16F690 before trying. What do you see? *Why?*

**Listing 3.1: digital-io.c**

```c
/*
 * This program toggles an LED connected to RC5
 * while monitoring the voltage at RC4 which
 * is controlled by a switch.
 *
 * If the voltage at RC4 is high, the LED is kept ON/OFF for 150 msec
 * If the voltage at RC4 is low, the LED is kept ON/OFF for 50 msec.
 */
void main()
{
```

Figure 3.1: Experimental setup to perform Digital I/O.
CHAPTER 3. DIGITAL I/O

3.4 LCD panel

LCD panels are an inexpensive way to display text. Almost all LCD panels come with a driver that allows you to interface with the panel. Almost all drivers use the same protocol and the data is sent to LCD driver using simple digital i/o. You can use either 4 or 8 i/o lines to send data to the LCD. In this section we shall use 8 line communication so that we can send a byte in one shot. With a 4 line communications you have to break up each byte into 2 halves (sometimes called nibbles\(^1\)). You then send the left nibble using the 4 data lines and then you send the right half using the same 4 data lines.

3.4.1 Driver protocol

**Software:** Initialization: After power up wait 250 milliseconds and then send the following textbf{command} bytes, one byte at a time.

\[0x30\ 0x30\ 0x30\ 0x38\ 0x10\ 0x0C\ 0x06\]

Most commands need about 100 microsecond delay. It is best to put some delays between sending command bytes.

---

\(^1\)half a byte is called either a *nibble* or *nybble* (to conform with the spelling of byte). However most spell checkers have a problem with the word nybble!
### 3.4 LCD PANEL

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Clear screen</td>
</tr>
<tr>
<td>0x02</td>
<td>Home the cursor. Does not clear</td>
</tr>
<tr>
<td>0x80+((row \times 0x40)+col)</td>
<td>Move cursor to any row, col. (See note)</td>
</tr>
</tbody>
</table>

**Note:** row can only be 0 or 1 and col between 0 and 0x3F. \((row \times 0x40)\) is best written as \((row \ll 6)\).

**Special commands:** Driver provides some special commands to control the display. The more important ones are given in table 3.3.

**Writing to screen:** Send the character using ASCII encoding.

**Hardware:** Communication between your micro and the LCD is controlled by three control lines: RS, R/W, E. The function of these control lines are as follows:

**RS** You use this line to LCD if you are sending it a command or you are sending it a data to be displayed on the screen. Set RS=0 if you are sending a command; set RS=0 if you are sending a data.

**R/W** This line determines who is sending and who is receiving data. Set R/W=0 if you are sending a byte to the LCD panel. Set R/W=1 if you want to receive a byte from the LCD panel. This assumes you are using an 8-bit interface. If you are using a 4 bit interface then you send or receive a nibble instead of a byte.

**E** This is the hand shake line. No communication takes place when E=0. To send a byte (or a nibble in the case of a 4 bit interface), you first set the data lines to what you want to send and then raise E i.e. E=1, put in a short delay and then bring it down, i.e. E=0. The exact amount of delay depends on the driver but they are all less than 1 milliseconds. So a delay of 1 millisecond should cover most manufacturers.
3.4.2 Sample program

The program shown in Listing 3.2 shows how to write to the LCD and also how to move the cursor to any place on the LCD screen. The program marches a string across the second row of LCD without affecting the first row. The schematic for connecting the PIC16F690 to LCD panel is shown in figure 3.2.

```c
/*
LCD pin out
GND
power
Contrast
RS => RA2
R/W => RA4
E => RA5
Data => PORTC (8 bit interface)
*/
#define RS PORTA.B2
#define RW PORTA.B4
#define E PORTA.B5
```

Figure 3.2: Connecting an LCD panel to a PIC16F690
3.4. LCD PANEL

void sendany(char c)
{
    PORTC = c;
    RW = 0;
    E = 1;
    delay_us(5);
    E = 0;
}

void send_command(char cmd)
{
    RS = 0;
    sendany(cmd);
    delay_ms(37);
}

void send_data(char value)
{
    RS = 1;
    sendany(value);
    delay_ms(37);
}

/* ASSUME E is always zero except as needed */
/*
 * sendany: The lowest level function to send a byte to lcd
 * Load PORTC with the byte you want to send
 * raise E
 * delay
 * lower E
 */

lcmdinit: Send the commands as required by the datasheet
```c
void lcdinit()
{
       delay_ms(40); // warmup
       send_command(0x30);
       delay_ms(100); // First command needs more
delay
       send_command(0x30);
       send_command(0x30);
       send_command(0x38);
       send_command(0x10);
       send_command(0x0c);
       send_command(0x06);
}

/*
 * Useful commands
 clear: clear the LCD screen
 move: Moves the cursor the the LCD
 */
void clear()
{
       send_command(0x01);
}
void move(char row, char col)
{
       char cmd;

       // Set DDRM address
       row = row & 1; // only 0 or 1 permitted
       col = col & 0x3F; // only 0 to 3F
       cmd = 0x80 + row * 0x40 + col;
       send_command(cmd);
}
init()
{
       PORTC = 0;
       E = 0;
       RW = 0;
       TRISA = TRISC = 0x00;
       ANSEL = ANSELH = 0;
lcdinit();
}

/*
 * send_string: Sends a string. This is a generic string
 printing routine
 */
```
/
void send_string(char *s)
{
    for (; *s != 0; s++)
        send_data(*s);
}

void main()
{
    char col = 0;
    init();
    clear();
    send_string("Hello world");

    // March the string " @ #" across the second row (row=1).
    while (1) {
        move(1, col);
        send_string(" @ #");
        delay_ms(100);
        col++;
        col = col & 0x0F;
    }
}
Chapter 4

Clocks and timing

4.1 The System Clock

The system clock is the main clock signal that drives the processor. Almost all micro controllers from Microchip have internal oscillators meaning you do not have to buy crystal oscillators and connect it the micro controller. The PIC16F690 has an 8 Mhz internal oscillator. By default, the micro controller runs at half the speed, i.e 4 Mhz. Bits 6-4 in OSCCON register controls the oscillator frequency as shown in the table 4.1. PIC16F1507 provides a wider choice as shown in table 4.2. The system clock’s frequency is referred to as FOSC in Microchip data sheets.

If you decide to change the clock rate in your program make sure that you do that at the very beginning of your program. Also, make sure you let the compiler know the clock rate. Compiler needs to know this! Various library functions must perform time critical functions and it has to know the clock rate.

4.2 Timer Systems

A timer in the simplest sense is a counter that counts clock ticks. The timers automatically rollover, i.e. once they reach the maximum value they rollover to zero\(^1\). Whenever the timer overflows, it will set a flag commonly referred to as an overflow flag. PIC16F690 and PIC16F1507 have very

\(^1\)On my desk is a digital clock and the seconds display goes from 00 to 59 and then rolls over back to zero. This is an example of a timer.
### Table 4.1: Selecting System Frequency: PIC16F690

<table>
<thead>
<tr>
<th>IRCF bits</th>
<th>System frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>8 MHz</td>
</tr>
<tr>
<td>110</td>
<td>4 MHz (default)</td>
</tr>
<tr>
<td>101</td>
<td>2 MHz</td>
</tr>
<tr>
<td>100</td>
<td>1 MHz</td>
</tr>
<tr>
<td>011</td>
<td>500 kHz</td>
</tr>
<tr>
<td>010</td>
<td>250 kHz</td>
</tr>
<tr>
<td>001</td>
<td>125 kHz</td>
</tr>
<tr>
<td>000</td>
<td>31 kHz (LFINTOSC)</td>
</tr>
</tbody>
</table>

### Table 4.2: Selecting System Frequency: PIC16F1507

<table>
<thead>
<tr>
<th>IRCF bits</th>
<th>System frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>16 MHz</td>
</tr>
<tr>
<td>1110</td>
<td>8 MHz</td>
</tr>
<tr>
<td>1101</td>
<td>4 MHz</td>
</tr>
<tr>
<td>1100</td>
<td>2 MHz</td>
</tr>
<tr>
<td>1011</td>
<td>1 MHz</td>
</tr>
<tr>
<td>1010</td>
<td>500 kHz</td>
</tr>
<tr>
<td>1001</td>
<td>250 kHz</td>
</tr>
<tr>
<td>1000</td>
<td>125 kHz</td>
</tr>
<tr>
<td>0111</td>
<td>500 kHz (default upon Reset)</td>
</tr>
<tr>
<td>0110</td>
<td>250 kHz</td>
</tr>
<tr>
<td>0101</td>
<td>125 kHz</td>
</tr>
<tr>
<td>0100</td>
<td>62.5 kHz</td>
</tr>
<tr>
<td>001x</td>
<td>31.25 kHz</td>
</tr>
<tr>
<td>000x</td>
<td>31 kHz (LFINTOSC)</td>
</tr>
</tbody>
</table>
4.2. TIMER SYSTEMS

<table>
<thead>
<tr>
<th>Timer</th>
<th>Control bit</th>
<th>Clock source when set</th>
<th>when cleared</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMR0</td>
<td>T0CS</td>
<td>T0CKI pin</td>
<td>FOSC/4</td>
</tr>
<tr>
<td>TMR1</td>
<td>TMR1CS</td>
<td>T0CKI pin</td>
<td>FOSC/4</td>
</tr>
</tbody>
</table>

Notes: T0CS is bit #5 in OPTION_REGISTER. TMR1CS is bit #1 in T1CON. T0CKI is RA2 configured as input.

flexible timers called TMR0, TMR1 and TMR2. TMR0 and TMR2 are 8 bit counters so their maximum value is 255 or 0xFF. TMR1 is a 16 bit counters, so its maximum value is 65535 or 0xFFF. TMR2 has the special feature so the value at which it rolls over can be set in software. This value is stored in the register PR2.

4.2.1 Event counting

The timers TMR0, TMR1 and TMR2 can be configured to count other sources of clock pulses. When a timer such as TMR0 counts the clock tics of an external clock, it is in an event counting mode. As a simple example, suppose that we want to count the number of vehicles using a highway. A simple solution is to use external electronics so that the external circuitry generates a pulse every time a car drives by\(^2\). By using the pulses generated by the circuitry, one can count the number of vehicles that uses the highway. Table 4.3 shows how the timers can be configured.

4.2.2 Pre scalers and Post Scalers

Pre scalers are a source of confusion for beginners. Suppose that you want to count some clock pulses but the clock rate is so fast that your counter may rollover several times for the time of interest. Rather than keeping track of how many times the timer rolls over etc. it is easier to slow down the

\(^2\)You may have seen a tube across the highway. The tube is just a mechanical switch that closes every time a vehicle goes over it
counting by counting every 16 clock ticks or every 32 clock ticks etc. This is what a pre scaler does. It sits between the clock source and timer and only lets in every 32 clock ticks. Then the timer will only count every 32 clock ticks. Thus the pre scaler is used to slow down the clock.

Post scalers do the same thing except the post scaler works after the count and not before. Without a post scaler, the overflow flag is set every time the timer overflows. However, if the post scaler is set to 1:2, then the flag will be set twice every full count. I.e. the flag will be set half way through the count and again at full count. Similarly a post scaler of 1:4 will have the flag set four times a full count. Thus the post scaler is used to speed up the clock rate of overflow. At home, I have clock that chimes every hour, i.e. every time the minutes counter overflows from 59 to zero. By flipping a switch, I can make it chime every 15 minutes. This equivalent to a post scaler of 1:4. An important point to remember: Post scaling does not change the speed of the clock. It only speeds up tasks that depend on the overflow flag.

4.3 Time Delays

One of the ways for PIC16F690 to waste time is to watch the timer. Your program waits for a timer overflow flag is set. Till the flag is set, it will be *nothing* loop. To make sure that it actually waits, the flag is initially cleared. The following program illustrates the use of timer based delays and also the effect of the pre scaler. The program waits for 50 overflows of TIMER0 and then toggles bit #0 of PORTC. Attach an LED circuit to RC0. Change the Pre scaler bits to see the effect on how fast the LED blinks.

Listing 4.1: tmr0delay.c

```c
/*
This program counts the TMR0 overflows and toggles PORTC every 50 counts

Clock source:
TOCS = 0  ->  clock = FOSC/4
PSA = 0   ->  Attach prescaler to TMR0
PS: prescaler value
  000  ->  1:2
  001  ->  1:4
  010  ->  1:8
  011  ->  1:16
*/
```
100 -> 1:32
101 -> 1:64
110 -> 1:128
111 -> 1:256

Note: If you do not want any prescaling, then do not attach
pre scaler
by setting PSA = 1

*/
void WaitForT0IF()
{
    INTCON.T0IF = 0; // Reset the flag
    while (INTCON.T0IF == 0) { //wait for flag to go high
    }
}

void main()
{
    int count;
    ANSEL = ANSELH = 0;
    TRISC = 0;
    OPTION_REG.T0CS = 0;
    OPTION_REG.PSA = 0;

    // Set Prescale to 000.
    // You can change the bits to see the effecgt of pre scaler
    OPTION_REG.PS0 = 0;
    OPTION_REG.PS1 = 0;
    OPTION_REG.PS2 = 0;
    TRISC = 0xFF;
    TRISC.F0 = 0;
    PORTC.F0 = 0;
    delay_ms(100);
    while (1) {
        for (count = 0; count < 50; count++)
            WaitForT0IF();
        PORTC.F0 = ~PORTC.F0;
    }
}
Chapter 5
Analog Input

5.1 Analog to Digital Conversion

5.1.1 Basics

Analog to Digital Conversion (ADC) is the most basic way for the micro-
processor to get information from the real world (analog world). An analog
to digital convertor, as its name suggests, converts an analog quantity to
da digital value. To sense the real world, a quantity in the real world such
as temperature, force, distance, etc. is first converted to an analog mea-
surabale quantity by a transducer. Common analog quantities produced by
a transducer include voltage, resistance, capacitance, inductance, time, dis-
placement\(^1\). An ADC can then convert this to a number that you can then
use in your programs.

5.1.2 ADC in \texttt{PIC16F690} and \texttt{PIC16F1507}

Both \texttt{PIC16F690} and \texttt{PIC16F1507} have built-in ADC to convert a \textit{voltage}
to a number. What this means is that you may have to use your back-
ground in electronics to convert the output of transducers to a voltage. For
our purposes we assume that the analog quantity we wish to measure is a
voltage. Key features of the built-in ADC convertors for \texttt{PIC16F690} and
\texttt{PIC16F1507} are as follows:

\(^1\)I will leave up to you to think up transducers that output the named quantities.
1. **There are hard limits on the input voltage levels.** If the supply voltage ($V_{DD}$) is 5V then the input voltage has to be between 0 and 5V. What happens if the input voltage is outside these limits? As long as the input voltage is not too much outside the limit, then any input above 5V is treated as 5V and any voltage below 0V is treated as 0V. It *is not a good idea to feed voltages outside to 0-5V range for any extended periods of time.* You also have the option of using some external voltage instead if $V_{DD}$. If you wish to do this, the external reference voltage must be connected to pin #18.

2. **Conversion is 10-bit linear.** As long as the input is between 0 and 5V, the converted value would be proportional to the input (within the resolution of the convertor). The converted value of 0V would be 0 and the converted value of 5V would be $2^{10} - 1$ or 1023. The converted value for any other voltage will be

$$\text{Converted value} = \begin{cases} 
0, & V < 0; \\
\left\lfloor 1023 \left( \frac{V}{5} \right) \right\rfloor, & 0 \leq V \leq 5; \\
1023, & V > 5.
\end{cases}$$

3. **Converted value is padded to 16 bits.** Although the converted value is a 10-bit quantity, it is padded with 6 zeros to produce a 16-bit quantity. The padding could be either on the left (right justification) or on the right (left justification). In terms of binary numbers, padding on the left does not change the value, while padding on the right has the effect of multiplying by 64, i.e multiplying by $2^6$. This padding is done in hardware.

4. **Conversion is clocked.** ADC requires a clock to perform its conversion. You can either supply an external clock or use the built-in system clock.

5. **Multiple ADC channels:** Both PIC16F690 and PIC16F1507 have multiple ADC channels, meaning you can measure up to 12 analog inputs. These channels are known as AN0, AN1, AN2.

6. **ADC shares pins with digital I/O:** ADC channels share pins with digital I/O. For example AN0 shares a pin with RA0, AN1 shares a pin
with RA1, AN2 shares a pin with RA2. PIC16F690 has two registers called ANSEL and ANSELH that determines if an analog input is selected or not. Bits in the register ANSEL determine if a the pin associated with AN channels 0-7 will be used as an analog pin or digital pin. For example suppose that bit #6 of ABSEL is one. This means that ADC channel #6 is active. This channel is connected to pin #14. RC2 is also connected to the same pin. Because ANSEL bit #6 is one, input to RC2 is disconnected from the pin and the pin is connected to AN6 instead. Similarly bits in ANSELH is used to connect analog channels 8 to 11, with bit #0 enabling AN8, bit #1 enabling AN9 etc. PIC16F1507 has three analog select registers called ANSELA, ANSELB, ANSELC that perform the same role.

7. **ADC is performed on demand:** Every time an analog value is to be read, you need to initiate conversion by specifying the channel you want to read. There is a hardware flag which will be high while the conversion take place and will be go low once the conversion is complete.

### 5.1.3 Procedure for reading analog input in **PIC16F690**

**Initialization:** You do this at the start of the program.

1. Turn on the appropriate ANSEL/ANSELH bits. If you plan on reading more than one channel then make sure you turn on all the bits.

2. Configure the conversion rate by setting appropriate bits in ADCON1. Table 9.1 in the data sheet recommends $F_{OSC}/8$ as the clock rate. This corresponds to setting just bit #4 in ADCON1.

**Conversion:** Every time you want to convert an analog input you have to do the following:

1. You start the conversion by writing to ADCON0. The key bits are ADON which turns on ADC module and GO/DONE_BAR bit that initiates the conversion. Other important bits are ADFM that tells the convertor if you want the result to be left or right justified and VCFG which determines if the convertor should use $V_{DD}$ as the reference or you will have external voltage input that will be
used as reference. In addition there are 4 more bits that are used to tell the convertor which ADC channel you want to measure.

2. After writing to ADCON0 your program should wait for conversion to complete. When the conversion is complete, the hardware will turn off the GO/DONE_BAR bit.

3. After the conversion is complete, the value can be read as a 16-bit number from the results register pair: [ADRESH:ADRESL].

The MikroC has all this neatly packaged in the ADC library. Note that you still need to do the initialization even if you plan to use the library.

5.1.4 Example: Reading the temperature using MCP9701

MCP9701 is an active thermistor chip. It is produces a voltage that is linearly related to the temperature in Celsius by the formula

\[ V = 400 + 19.5T, \quad T \text{ in degrees Celsius and } V \text{ in millivolts.} \]

Thus if the temperature to be measured is \( T \) then ADC will convert the voltage from MCP9701 as

\[ \text{ADC} = 1023, \left( \frac{400 + 19.5T}{5000} \right) = 4(T + 20.5) \]

We can solve for \( T \) as

\[ T = \frac{\text{ADC}}{4} - 20.5 \]

If we want to round off \( T \) then we need to add 0.5 to get

\[ T = \frac{\text{ADC}}{4} - 20 \]

Note that all the calculations can be done using integers! This is the rationale for the odd looking coefficient 19.5.

Program listing in Listing [5.1](#) implements a simple temperature measurement program. Although the output is sent to the PC using UART, you can easily send the output to an LCD or a serial terminal. The schematic is show in figure [5.1](#) and typical output is shown in figure [5.2](#)
5.1. ANALOG TO DIGITAL CONVERSION

Figure 5.1: Schematics for testing MCP9701 using UART tool to view output

Listing 5.1: mcp9701.c

/*
 * This program reads temperature sensor MCP9701
 * and prints temperature in C
 * 
 * Connect MPC9701 to RC2/AN6
 * 
 * This program uses RA0, RA1 for UART.
 * Since these pins are rarely used and these two pins are normally
 * connected to PicKIt2, setting RA0 as TX and RA1 as RX makes
 * it convenient to use UART tool.
 * Since we arbitrarily assign TX and RX and not use the one that is
 * built into the micro, we have to use software UART instead of hardware UART
 * /

/*
 * Support functions
 */
Figure 5.2: Typical output. Because of an accidental misconnection, the sensor got really hot!

```c
void print_string(char *s)
{
    for (; *s != 0; s++)
        Soft_Uart_Write(*s);
}

char buffer[10];
void print_value(int value)
{
    IntToStr(value, buffer);
    print_string(buffer);
}

init()
{
    TRISA = TRISB = TRISC = 0xFF;
    ANSEL = ANSELH = 0; // Turn off all ADC
}
5.1. ANALOG TO DIGITAL CONVERSION

// ADC initialization
ANSEL.F6 = 1; // configure AN6 to analog
ADCON1 = 0;
ADCON1.F4 = 1; // Set ADC clock to fosc/8
Soft_Uart_Init(&PORTA,1,0,9600,0); // Initialize UART to 9600 baud

main()
{
    int adcval, degc;
    init();
delay_ms(5000);
    print_string("+---------------------+");
    print_string("| Using MCP9701 |");
    print_string("+---------------------+");
    while (1) {
        adcval = Adc_Read(6);
        degc = adcval / 4 - 20; // Or (adcval >> 2) + 20
        print_string("Raw input: ");
        print_value(adcval);
        print_string(" Tempertaure in degrees C: ");
        print_value(degc);
        print_string("C\r\n");
delay_ms(1000);
    }
}
Chapter 6

Character Input/Output

6.1 Asynchronous Serial Communication

Asynchronous Serial Communication dates back to early 1930’s and evolved into RS232 in the early 1960. The letters RS stand for recommended standard meaning that manufacturers and vendors could adapt the standard to suit their needs. Since communication was central to early computing, standard chips that performed the actual communication started to evolve in the early 70’s and the chip that did the communication was called Universal Asynchronous Receiver/Transmitter (UART) and people started to refer to the communication as UART communication. Today the UART chip can perform both synchronous and asynchronous communication and are now called USART (universal synchronous/asynchronous receiver/transmitter).

What does all this mean? Simply put, the sender starts sending data one bit at a time. The bits are precisely timed, so the receiver can catch the bits as they come. The rate at which the bits are sent is called the baud rate. Both the sender and the receiver have to agree on the baud rate so the receiver can catch the bits as they are sent. The communication is bidirectional and the sender sends data out of the TX line and receives data on the RX line\(^1\). The communication is called asynchronous, because the receiver has no additional channels to know when the transmission starts and where one bit ends and the next bit starts. It is all done by timing which

\(^1\)Make sure that you connect TX of one device to the RX of the other device when you set up communication! A common mistake is to connect TX to TX and Rx to RX.
means both the sender and receiver agree on the baud rate\(^2\).

The voltage levels of RS232 are designed to meet the needs of electro-mechanical teletypes. Logic zero is sent as +12V and logic one is sent as -12V (commonly referred to as RS232 voltage levels). The micro controller world uses 0V for logic zero and 5V for logic 1 (commonly referred to as TTL levels). Since this is a common problem, there are specialized chips MAX232 and MAX233 that will change the voltage from one standard to the other.

Until recently, most computers came equipped with communication ports (COM ports) for serial communications. Today most computers come with USB ports and you need USB/Serial converters for serial communications. When using USB to Serial converter, make sure that the voltage levels are correct. Most of them use RS232 voltage levels (±12V). If you want to use one of these, then make sure you also use a MAX233 chip to convert between RS232 voltage levels and TTL voltage levels. A better option is to get a converter that is designed for TTL voltage levels. Two popular USB-Serial adapters are made by Silicon Labs (CP2102) and FTDI. They cost between $10 to $20\(^3\). If you do not mind a bit of inconvenience, then you can save the cost of a converter by using PicKit2. PicKit2 has a utility that will convert to PicKit2 to a bare bones terminal with USB/serial conversion.

### 6.2 Printing

If you have done any programming, you too printing for granted. All you had to do was call `printf` or `cout` and characters appeared magically on the computer screen. Believe me, there is a lot going on under the hood! When working with micro controller you have to do all the work. We will develop the core printing routines in this chapter. To avoid name conflicts with standard functions, I will use *nonstandard* names for functions. These names resemble similar functions available for MC68HC111.

**out1char:** This is the basic function that sends a character out. The equivalent ANSI function would be `putc` or `putchar`. If you have programmed

\(^2\)If you find that the communication is garbled, chances are the baud rates do not match.

\(^3\)My favorite is the FTDI cable costing around $20. You can get that from Mouser: Part number: 895-TTL-232R-5V
6.2. PRINTING

the MC68HC11 this would be equivalent to OUTA. MikroC provides a library function called Uart1_Write. The easiest and cleanest way to make the link is to use define directive

#define out1char Uart1_Write

If the library changes, all you have to do is change this one line.

**outstr:** This is the next level up where you print a string. A string is an array of characters with end of string denoted by ascii zero. So you just use a for loop to print a string as in

```c
// Print the string s
for(k=0; s[k]!=0; k++) out1char(s[k]);
```

**out1value:** To print an integer, we need to convert it to a series of characters, i.e. string. Since the largest value that fits in 16 bits is 65535 all we need to do is peel out the 10,000-th digit, the 1000-th digit etc. Fortunately, MikroC has done all the work for us and all we need to do is use it as shown below

```c
char buffer[10];
IntToStr(value, buffer);
outstr(buffer);
```

The above code snippet is to print integers. MikroC also provides additional functions in the conversion library.

**outrhlf:** out1value is useful when we want to output in decimal. However, printing in hexadecimal can be acceptable and it is considerably easy to print in hexadecimal format. Moreover, the functions that we will develop is completely portable in the sense it can be run on any computer (including your PC). The basic building block for printing in hexadecimal is to print one hex-digit. A hex-digit has a value from zero to fifteen and hence this value would be stored in the right half of a byte and hence this function will be called outrhlf
6.2.1  A sample program

To test this program you would need a way for your PC to receive serial data and display it on the screen. At the hardware level, you need a usb-serial convertor and for software you need a terminal emulation program such as *Putty* or *Hyperterm*. Alternatively, you can use the UART tool that comes with *PicKit2*. See Appendix B for details. If you use the UART tool, you should see something similar to figure 6.1.

![Figure 6.1: Output of the sample printing program.](image)

Listing 6.1: printing.c

```c
#define out1char Uart1_Write
void outstr(char s[]) {
    int k;
    for (k = 0; s[k] != 0; k++)
        out1char(s[k]);
}
```
// Print in decimal
void out1val(int value)
{
    char buffer[10];
    IntToStr(value, buffer);
    outstr(buffer);
}

// Print in hex...
void outrhlf(char x) // Print the right half of a byte (0-15) (nibble)
{
    x = x & 0x0F; // Knock out the left half
    if (x < 10) { // same as decimal digit. So
        out1char(x + '0');
    } else { //values 10-15 are printed as
        A-F
        out1char(x + ('A' - 10));
    }
}

// Print a byte
// First print the left nybble and then the right nibble
// We do this by shifting the left side 4 places so left
// nibble goes to right
// and us outrhlf
void out1byte(char x)
{
    outrhlf(x >> 4); // Print the left nibble by
                    // moving it right
    outrhlf(x);     // ... followed by right
                    // nibble
}

// print a word consisting of two bytes
// First print the left byte and then the right
// Same idea as before
void out1int(int x)
{
    out1byte(x >> 8); //Shift over the right byte to
                    // lefey
CHAPTER 6. CHARACTER INPUT/OUTPUT

```c
out1byte(x);
}

main()
{

delay_ms(5000); // Give time for the user to connect to UART etc.
Uart1_init(9600); // sets the baud rate
outstr("++++++++++++++++++++++++++++++\r\n");
outstr(" Testing print functions +\r\n");
outstr("++++++++++++++++++++++++++++++\r\n");
outstr("year is: ");
out1val(2012);
outstr(" or in hex: ");
out1int(2012);
outstr("\r\n");
}

/*
To test this program using PicKit2 UART tool
1. Make sure that you have the latest PicKit2 software.
   Older ones do not have a way to turn power on and off in the UART tool.
2. Download the hex file to PIC16F690
3. Connect RB7 to pin #4 of PiKkit2. If you want to be safe
   , disconnect the pin#4 from RA0.
4. select baud rate
5. Turn on power
6. Connect to Pic (you have less than 5 seconds to do this !)
*/

6.3 Reading from keyboard

Reading data from the user is a bit tricky. If your program tries to read from its input, it will block meaning it will wait for the input. This can be highly undesirable as it means nothing else can happen. Unless you write a multi-threaded program (using interrupts) some time critical tasks may not take place.

MikroC provides two functions for performing inputs. UARTx_Data_Rdy that returns a zero if there is no character to be read, or else it return one. If you do not want your program to block, make sure to call this function before you try and read a character. The other function is UARTx_Read
that actually reads the character. **Warning: If you call UARTx_Read and there is no character to read, the program will wait patiently till it receives a character.**

The following program in Listing 6.2 illustrates character input. It will print a period while waiting for an input to show that there is no blocking. Again I have defined two macros in1char to read a character and kbhit that checks to see if there is a character available. The program turns on or off an LED connected to RC1 depending on the user input. A '1' turns it on and a '0' turns it off.

**Listing 6.2: reading.c**

```c
#define out1char(c) Uart1_Write(c)
#define in1char() Uart1_Read()
#define kbhit() Uart1_Data_Ready()

void outstr(char s[])
{
    int k;
    for (k = 0; s[k] != 0; k++)
        out1char(s[k]);
    delay_ms(1);
}

/*
This program reads a character from the input and turns on or off
an LED connected RC5

The exact connection of the LED is defined using macros, so change the
macro accordingly

This program illustrates non-blocking user input. If there is no
input, the program prints a '.' and goes its merry way
*/
#define LED PORTC.F0
#define LED_DIR TRISC.F0

void main()
{
    char c;
    ANSEL = ANSELH = 0;
    TRISA = TRISB = TRISC = 0xFF; //All inputs by
```
CHAPTER 6. CHARACTER INPUT/OUTPUT

```c
default
LED_DIR = 0; // LED connected to RC5
LED = 1;
Uart1_Init(9600);
delay_ms(5000); // Give user time to connect terminal etc.
outstr("\r\nPress 1 to turn On, 0 to turn Off Led\r\n");
while (1) {
    if (kbhit()) {
        c = in1char();
        if (c == '1')
            LED = 1;

        else if (c == '0')
            LED = 0;

        else
            out1char('?');
    } else {
        out1char('.');
        delay_ms(500);
    }
}
```

6.4 Numerical Input

In many applications, we want to obtain numerical inputs from the user. The input comes one digit at a time. If the program expects a fixed number of characters, then there is no difficulty in deciding when to stop. However, most users will not be happy if you want them to type in 007 because your program requires three digit numbers as inputs. So, our program will now know when the input ends till it has read one character too many. One way to handle this is to have a mechanism to `unget` a character.

6.4.1 Pushing back a character

The portable way to perform this is to have a pushback character, and two cooperating functions `getc` and `ungetc` as shown below:

---

4 *digit* is used in a generic sense. It could be decimal digit, or hexadecimal digit, etc.
static char pushback; // Push back character.
                   // Can push back any nonzero (non ascii null)
void ungetc(char c) {
    pushback = c;
}
char rawinput() { // Read a char if available; else return 0
    if(kbhit()) return in1char();
    else return 0;
}
char getch() {
    char c;
    if (pushback) { // We have a character that was pushed back
        c = pushback;
        pushback=0;
    } else c = rawinput();
    return c;
}
// Patiently wait for a character and get it
char getchar() {
    char c;
    do {
        c = getch();
    } while (c==0); // Keep going till we get a non-null
    return c;
}

In the above code, rawinput reads the keyboard without regard to push
back mechanism. getch first checks to see if a character was pushed back.
If so, it returns that character. Or else it uses rawinput. Thus getch acts
as a gate keeper. Some times we want to wait for a character. The function
getchar takes care of it. A fortuitous by product is that we can clear any
pushed back character by pushing back a zero. Rather than remember this,
it is best to create a macro function clear_pushback as shown below:

#define clear_pushback() ungetc(0)

Get into the habit of creating macros for arcane and hard to remember con-
structs.
6.4.2 Decimal Input

To understand the algorithm, consider the following thought experiment. You are watching a scrolling display where digits appear one at a time. For simplicity assume that we are using decimal number representations. Suppose the display shows 7. In the absence of any further information, you can assume that the value is seven, but you will have to wait before you can decide. Suppose that the next digit on the display is 2. Based on what you have seen so far, you can assume that the value is seventy two. Suppose next you see a 3. Based on this, you can assume that the value is seven hundred and twenty three. You can proceed along this line until you get an invalid character. Hence the following algorithm:

Initialize value = 0

step 1: Get a digit d.

step 2: If the input is invalid, we are done else go to step 3.

step 3: Update value to 10 times value + d and go back to step 1.

The key question is how do we get a digit? Unless you are using some obscure protocol, chances are the characters are encoded using ASCII or a super set of ASCII, such as unicode. The advantage of ASCII is that numerals have consecutive ASCII codes. That is code for 1 is one more than the code for 0 and the code for 2 is one more than the code for 1 and so on. Second advantage of ASCII is that the C compiler knows ASCII codes so you don’t have to know that the code for 0 is 48. If you write '0' the compiler will treat it as 48. Hence the following code

```
int read_decimal() {
    int value;
    char c;
    while(1) {
        c = getchar();
        // Check and see if the input is valid
        if (c < '0' || c > '9') {
            //Invalid char
            ungetc(c); //Throw it back
            continue;
        }
        // Update value
        value = 10 * value + c - '0';
    }
    return value;
}
```

5We still have to clean up after ourselves by pushing back the input
6.4. NUMERICAL INPUT

return value;
} else {
    value = value * 10 + (c-'0');
}
}

Note that if the user tries to enter a value greater than $2^{16} - 1$ the function automatically casts off multiples of $2^{16}$.

6.4.3 Hexadecimal Input

Reading in hexadecimal is lot easier. The logic is same as before except we need to multiply by 16 instead of 10. Multiplying by 16 is best achieved by shifting left 4 bits. Also, the standard hexadecimal characters for 10 to 15 is A–F or a–f. The key is to realize that for letters A–F we need to subtract the ascii code of A and then add 10. Thus the code for reading hexadecimal input is shown below

```c
int read_hex() {
    int value;
    char c;
    char d;
    while(1) {
        c = getchar();
        // Check and see if the input is valid
        if (c >= '0' && c <= '9') // digit
            d = c - '0';
        else if (c >= 'A' && c <= 'F') // upper case Hex
            d = c - 'A'+10;
        else if (c >= 'a' && c <= 'f') // lower case Hex
            d = c - 'a'+10;
        else { // invalid char
            unget(c);
            return value;
        }
        // Accept the input digit
        value = (value << 4) + d; // We can also use the | instead of +
    }
}
```
It is best to package all these functions in a separate file called `io.c` and provide a header file with prototypes in `io.h`
Chapter 7

Pulse Width Modulation

7.1 Why Pulse Width Modulation?

Pulse Width Modulation (PWM) is the preferred way of generating analog signals from your microcontroller. Recall that the output of any pin on the microcontroller would be either 0V or 5V, i.e. it is a binary or digital output. Suppose you want 3.4V output out of the device? How can you do that? A precise answer to that is it can’t be done. However, you can generate 3.4V for all practical purposes by turning on a pin for some time and off for some time as shown in the figure 7.1. In the waveform, the output is kept ON for 68 msec and OFF for 32 msec. We say that the duty ratio or duty cycle is 68% with a cycle time of 100 msec. If this signal is sent to an analog device, the device will, on the average, see 5V.68, or an average voltage of

![Figure 7.1: A typical PWM waveform.](image)
3.4V. It is important that the value is only an average. However, of the waveform switches rapidly enough, most analog systems will only respond to the average value and we have in effect produced a 3.4V analog signal. For example, if the PWM waveform is fed to a light emitting diode and human eye will not respond to any switching above 30-50Hz. Similarly if the waveform is sent to speaker, human ear will not perceive any signals above 12-15KHz. And if the signal is sent to a motor, most motors will not respond to any frequencies above a few Hz. There are a two competing of considerations in the choice of the frequency

1. Any frequency below 4-5 KHz will produce an annoying audible hum.

2. Most analog systems will have what are called switching losses. These losses occur every time the signal goes from 0 to 5 or from 5 to 0. Higher the frequency, you have more switching per second and hence more losses per second. Loss per second is the power loss, so higher frequencies usually mean higher losses, more heat produced etc.

### 7.2 Generating PWM signals

The procedure for producing PWM signals is relatively simple. It consists of counter, a reset value and a compare value. The method is as follows:

1. Have a counter that starts at zero and counts up.

2. When the value in the counter matches the reset value, set the counter back to zero on the next count.

3. Every time the counter gets reset, turn on an output pin.

4. Every time the counter equals the compare value, turn off the output pin.

An example will clarify the concept. Suppose that the reset value is 107 and the compare value is 48. Now the counter starts at zero, and the output is high. The output will stay high until counter becomes 48. At this time the output will be zero. Thus the output will be on while counter is 0,1,2, · · · , 47 or total of 48 counts. After counter goes past 48, it will keep counting until it gets to 107. After this it will go back to zero and the whole cycle repeats. Thus the cycle time is a count of 108 and the duty ratio is $48/108 = 37.5\%$
7.3. HARDWARE PWM GENERATION IN PIC16F690

Some terminology is on order. The counter is also often called a free running clock and the speed with which it counts is called the clock rate usually expressed in ticks per second or Hz. This value affects the frequency. The following is a typical calculation that is performed

Sample Calculation A clock with frequency 1MHz is used to generate a PWM signal. Suppose that we want the PWM signal to have a frequency of 4KHz, and 45% duty ratio, determine the reset value and the compare value

Answer: Since we need a 4KHz signal, the cycle time should be $1/4000 = 0.00025$ seconds. Each clock tick is $1/10^6$ seconds. So each cycle should count to $0.00025 \times 10^6 = 250$. Thus we have

\[ \text{reset value} = 250 - 1 = 249 \]

To get 45% duty ratio, we need

\[ \text{compare value} = 250 \times 0.45 = 112.5 \]

Since counters are integers, we round this value to either 112 or 113 according to our personal preference.

7.3 Hardware PWM generation in PIC16F690

7.3.1 General Principles

PIC16F690 has compare module that does all the counting comparing etc. The counter is TMR1 and is clocked by the processor clock. We will get into minute details shortly. From a PWM generation point of view, the counts go up by $1/4$ (by the miracle of shifting by two bits) and the reset value is stored in a register called PR2. The compare value can have fractional values of 1/4, 1/2 or 3/4. The whole part of the compare value is stored in the register CCPR1L and the fractional part in fourths is stored in the bits 5 and 4 of CCP1CON. The fractional part is there to provide some extra accuracy but you can leave it zero for most practical purposes. Thus, the duty cycle ratio is given by

\[
\text{Duty Cycle Ratio} = \frac{\text{CCPR1L} + \text{CCCP1CON}<5:4>/4}{\text{PR2} + 1}
\]
and the cycle time for the waveform is given by

\[ \text{Period} = (PR2 + 1) \times (4T_{\text{clock}}) \]

The reason for 4 in the above formula is because the count is in fourths. See chapter 4 for determining the clock times.

### 7.3.2 Implementation

To generate the PWM signal we have to go through the following steps

1. Enable PWM generation by setting bits 3 and 2 of CCP1CON.

2. PIC16F690 can actually control four output lines when generating PWM signals. We usually need this for bidirectional control of DC motors. If we want just a simple PWM signal, then clear bits 6 and 7 of CCP1CON. In C we do this by

   \[
   \text{CCP1CON} = \text{BIT}(3) + \text{BIT}(2);
   \]

3. Next set PR2 to as high a value as you can. Note that this is the reset value. The compare value has to be less than this so higher this value, more resolution you have in the duty cycle ratio. Again it is best to chose the highest possible value of 255.

4. Finally enable the PWM generation by turning on the TMR2ON bit in T2CON register. Depending the compiler you have, this bit may be predefined either as TMR2ON or T2CON.TMR2ON. There seems to be no universal agreement on this.

This will generate a PWM signal on the CCP1/P1A pin which is RC5. Make sure that you set the TRIS bit to make this an output!

### 7.3.3 Example Program

Connect a LED circuit to RC5 as shown in figure 7.2 and compile and run the following program. The brightness will grow and dim alternately on the LED. The program gradually increases the duty ratio from 0 to maximum and then reduces it back to zero. There is a slight delay between changes so the eye can follow the change in intensity.
7.3. HARDWARE PWM GENERATION IN PIC16F690

Figure 7.2: Circuit to test PWM code for PIC16F690.

**Listing 7.1: pwmtest.c**

```c
/*
   This program gradually increases the duty cycle ratio from zero to maximum
   value of 255 and then gradually decrease it back to zero.
   See notes for details.
*/
#define BIT(n) ((1)<<(n))

void main()
{
    char duty;

    // Make all the io pins inputs (safe!) and make RC5 an output
    TRISC = 0xFF;
    TRISC.F5 = 0;
    CCP1CON = BIT(3) | BIT(2);  // Simple PWM signal
    PR2 = 255;                  // Maximum range for duty
```
7.4 Hardware PWM generation in \texttt{PIC16F1507}

This is one area where \texttt{PIC16F1507} is more powerful than \texttt{PIC16F690}. Unlike \texttt{PIC16F690} that can generate only one PWM signal, \texttt{PIC16F1507} can simultaneously generate up to four PWM signals. All the channels share the same reset value stored in \texttt{PR2} and the same counter. However, each channel has its own compare value stored in \texttt{PWM\textsubscript{x}DCH}, where \( x \) can be 1, 2, 3, or 4. Each channel has two Enabling bits. \texttt{PWM\textsubscript{x}EN} bit enables the operation of the PWM generator, and \texttt{PWM\textsubscript{x}OE} enables the output. There are situations when we want to keep the PWM generator going but want to temporarily disable the output. Hence the two enabling bits. The procedure for generating PWM signal is as follows:

1. Set \texttt{PR2} value (use 255 for the most range)
2. Enable the PWM channel and PWM output by setting bits 7 and 6 in \texttt{PWM\textsubscript{x}CON}.
3. Clear the corresponding TRIS bit to make the PWM pin and output pin.
4. Change the duty cycle ratio as needed up to a maximum of \texttt{PR2}. You do this by setting the whole part of the value in \texttt{PWM\textsubscript{x}DCH} and the
fractional part measured in fourths in bits 7 and 6 of PWMxDCL. If you do not need this extra accuracy, you can leave the fractional part zero.

As before the duty ratio for channel \(x\) is given by

\[
\text{Duty Cycle Ratio} = \frac{\text{PWMxCH} + \text{PWMxDCL}<7:6>/4}{\text{PR2} + 1}
\]

### 7.4.1 Example program

In this program we will generate two PWM signals PWM2 and PWM4. As the duty ratio of one increases the other will decrease. PWM2 appears is generated on RC3 while PWM4 is generated on RC1. Use the circuit shown in figure 7.3.

![Figure 7.3: Circuit to test PWM code for PIC16F1507.](image)

Listing 7.2: pwm1507.c

```c
/*

*/
This program gradually increases the duty cycle ratio from zero to maximum value of 255 and then gradually decreases it back to zero. Two LED's are controlled. As the duty cycle ratio of one of the LED's increases, the ratio of the other decreases.

See notes for details.

```c
#define BIT(n) ((1)<<(n))

void main()
{
    char duty;
    TRISA = TRISB = TRISC = 0xFF;  // All pins are inputs
    PR2 = 255;  // Maximum range for duty
    PWM2CON = BIT(7) | BIT(6);  // Bit7 is PWMxEN, Bit6 is PWMxOE
    PWM4CON = BIT(7) | BIT(6);
    TRISC.F3 = 0;  // Make PWM2 output pin
    TRISC.F1 = 0;  // ditto PWM4
    T2CON.TMR2ON = 1;
    while (1) {
        // Gradually increase the duty cycle ratio
        for (duty = 1; duty != 0; duty++) {
            PWM2DCH = duty;
            PWM4DCH = 255 - duty;
            delay_ms(25);
        }
        delay_ms(500);

        // ... gradually decrease it
        for (duty = 255; duty != 0; duty--) {
            PWM2DCH = duty;
            PWM4DCH = 255 - duty;
            delay_ms(25);
        }
        delay_ms(500);
    }
}
```
7.5 Dual Motor Control

This application brings in one place digital output, a practical application of pulse width modulation (using PIC16F1507) to control the speed of motors, peripheral I/O chips that can perform tasks that a micro can’t. The motor control circuit is shown in Figure 7.4.

![Figure 7.4: Dual Motor controller. Shown with PIC16F1507 but PIC16F690 can also be used.](image)

7.5.1 Half bridge chip: SN754410

At the heart of the application is the motor driver chip, SN754410. This chip is a motor driver that can drive up to two (brushed) DC motors. It has two independent power sources, a 5V power that powers the internal electronics and 5-30V power to drive the motors. The two power sources can be the same but this can create noise in the power line for the electronics. If you use two sources make sure they have a common ground, i.e. the two negatives are connected. Technically the chip is four buffer amplifier numbered 1-4. The input to the buffer is labeled A and the output is labeled Y as is customary in logic chips.
Principle of operation  In the figure 7.4 one of the motors is connected between 1Y and 2Y. If 1Y is driven high and 2Y is driven low, the motor will spin in one direction. If, on the other hand, 1Y is driven low and 2Y is driven high, the motor will spin in the opposite direction. If both 1Y and 2Y are driven the same, high or low, the motor is shorted and this causes braking. All of this is controlled by the enable line 1, 2EN. If the enable is low, then the chip does not control the motor an the motor can freewheel.

In the circuit, there are 6 LEDs for debugging purposes. Get into the habit of adding LEDs on output lines. To reduce the loading use 680 ohms to 1 K for a dull glow.

7.5.2 Controlling the motor

We will program a few basic modes of operation.

Controlling the speed

To control the speed of the motor, all one needs to do is send a pulse width modulated signal to the enable pins. Most inexpensive DC motors may not respond to signals above a few KHz so you should keep the PWM frequency to less than 4 KHz. If in the place of PIC16F1507 you use PIC16F690 all you can do is turn on or turn off the motor by enabling or disabling the motor drive.

7.5.3 Chip Independent programming

When one starts developing elaborate projects it gets to be both tedious and error prone to have one set of code for one processor and another for a different processor. All C compilers have what is known as conditional compilation. This feature allows one to include or exclude parts of your program based on conditions that the compiler can evaluate. A typical code will look something like this

```c
// uncomment the next line to target PIC16F690
// #define TARGET_P16F690
```

1 As contrasted with braking.
2 Technically the pre-processor component of the compiler.
The Listing 7.3 gives the complete code for controlling two motors. The code itself does not do much but can be used as the starting point for a number of other projects.

Listing 7.3: dual-motor.c

```c
//uncomment the next line to target PIC16F690
//#define TARGET_PIC16F690

// MAP IO pins to signals
// Note: Forward and reverse are notional directions and depend on how the
// motor is wired and how they are mounted. Change the definitions below
// to match reality!

////// LEFT MOTOR
#define LMOT_FORWARD PORTC.F5
#define LMOT_REVERSE PORTC.F4
#define LMOT_ENABLE PORTC.F3

////// RIGHT MOTOR
#define RMOT_FORWARD PORTC.F0
#define RMOT_REVERSE PORTC.F2
#define RMOT_ENABLE PORTC.F1

// Useful macros
#define BIT(n) (1<<(n))

void adcOff()
{
    // Turn off ADC
    #if defined (TARGET_PIC16F690)
    ANSEL = ANSELH = 0;
    #else
    ANSELA = ANSELB = ANSELC = 0;
    #endif
}

void init()
{
    //Initialize the system
```
CHAPTER 7. PULSE WIDTH MODULATION

TRISA = TRISB = 0xFF; // All of PORTA & PORTB are inputs
PORTC = 0x00; // Make sure to turn off the port first!
TRISC = 0x00; // All of PORTC is output
adcOff();

#if !defined(TARGET_PIC16F690)
   // Initialize PWM generator. See text for explanation
   PR2 = 255; // Maximum range for duty
   PWM2CON = BIT(7) | BIT(6); // Bit7 is PWMxEN, Bit6 is PWMxOE
   PWM4CON = BIT(7) | BIT(6);
   T2CON.TMR2ON = 1;
   PWM2DCH = PWM4DCH = 0;
#endif /* */

//---------------- Motor Control functions
void setLeftVelocity(int velocity)
{
    int direction, speed;

    // Get direction and speed
    if (velocity >= 0) {
        direction = 0;
        speed = velocity;
    } else {
        direction = 1;
        speed = -velocity;
    }

    if (speed > 255)
        speed = 255;

    // Set the direction
    if (direction == 0) {
        LMOT_REVERSE = 0;
        LMOT_FORWARD = 1;
    } else {
        LMOT_REVERSE = 1;
        LMOT_FORWARD = 0;
    }

    // Set the speed
7.5. DUAL MOTOR CONTROL

// PIC16F690 does not have two PWM channels
// so it is either ON or OFF.
// So any speed >127 would be on, or else off
//
// PIC16F1507 has a smoother control
#if defined(TARGET_PIC16F690)
  if (speed > 127)
    LMOT_ENABLE = 1;
  else
    LMOT_ENABLE = 0;
#else /* */
  // Left Motor enable is connected to PWM channel 2
  PWM2DCH = speed;
#endif /* */

// Do the same for right motor
void setRightVelocity(int velocity)
{
  int direction, speed;

  // Get direction and speed
  if (velocity >= 0) {
    direction = 0;
    speed = velocity;
  } else {
    direction = 1;
    speed = -velocity;
  }
  if (speed > 255)
    speed = 255;

  // Set the direction
  if (direction == 0) {
    RMOT_REVERSE = 0;
    RMOT_FORWARD = 1;
  } else {
    RMOT_REVERSE = 1;
    RMOT_FORWARD = 0;
  }
}
CHAPTER 7. PULSE WIDTH MODULATION

// set the speed
// PIC16F690 does not have two PWM channels
// so it is either ON or OFF.
// So any speed >127 would be on, or else off
//
// PIC16F1507 has a smoother control
#endif defined(TARGET_PIC16F690)
if (speed > 127)
    RMOT_ENABLE = 1;
else
    RMOT_ENABLE = 0;
#endif /* */
// Right Motor enable is connected to PWM channel 4
PWM4DCH = speed;
#endif /* */

#define DELAY_TIME 10
// Test function
void test()
{
    int speed;

    // Gradually increase the left motor velocity from 0 255
    for (speed = 0; speed < 256; speed++) {
        setLeftVelocity(speed);
        delay_ms(DELAY_TIME);
    }

    // Go back to zero
    for (speed = 255; speed >= 0; speed--) {
        setLeftVelocity(speed);
        delay_ms(DELAY_TIME);
    }

    // repeat in opposite direction
    for (speed = 0; speed < 256; speed++) {
        setLeftVelocity(-speed);
        delay_ms(DELAY_TIME);
    }
}
7.5. DUAL MOTOR CONTROL

```c

    // Go back to zero
    for (speed = 255; speed >= 0; speed--) {
        setLeftVelocity(-speed);
        delay_ms(DELAY_TIME);
    }

    // Repeat for right motor
    for (speed = 0; speed < 256; speed++) {
        setRightVelocity(speed);
        delay_ms(DELAY_TIME);
    }

    // Go back to zero
    for (speed = 255; speed >= 0; speed--) {
        setRightVelocity(speed);
        delay_ms(DELAY_TIME);
    }

    // Repeat in opposite direction
    for (speed = 0; speed < 256; speed++) {
        setRightVelocity(-speed);
        delay_ms(DELAY_TIME);
    }

    // Go back to zero
    for (speed = 255; speed >= 0; speed--) {
        setRightVelocity(-speed);
        delay_ms(DELAY_TIME);
    }

    main()
    {
        init();
        while (1) {
            test();    // Keep running the test
        }
    }
```
Chapter 8

Serial Communications

8.1 SPI: Serial Peripheral Interface Bus

This is one of the most common serial communication busses. Characteristic features of the bus are

1. It is a bus and not a port. In a bus, you can connect multiple devices. Compare this to the serial port which connects one device to your microcontroller.

2. It is synchronous bus. What this means is there is a clock line as part of the bus. When the data is valid on the bus is signalled by the clock.

3. The connection on the bus is based on a Master-Slave\(^1\) configuration. There is a bus master (usually your micro controller) which controls all the data traffic.

4. Communication can take place between the master and a specific slave. Master decides which slave it wants to communicate with.

5. Each slave has a separate line to the master (this makes it hard to expand the bus easily to add more devices). Master uses this line to determine which slave it wants to communicate with. This line is often called SSBAR or SS. Here SS stands for \textit{slave select}. The suffix BAR is a reminder to the user that you select by setting the line to zero, i.e. negative logic.

\(^1\)I personally find the terminology Master-Slave odious but this is the established terminology.
6. Only the master can initiate communication with a slave. If a slave wants to send data to the master, it has to wait for master to establish communication (sort of having a phone line where outgoing calls are blocked!)

7. Communication between Master and Slave occurs over two dedicated data lines: MOSI and MISO.

**MOSI** Master Out Slave In. This is the line by which the master communicates with the slave.

**MISO** Master In Slave Out. This is the line by which the slave communicates with the master.

An alternate names for the data line that is viewpoint neutral are DI (data in), SDI (serial data in), DO (data out), SDO (serial data out). In this terminology, connect DI/SDI of one device to DO/SDO of the other device. A typical bus connection is shown in figure 8.1.

8. When Master and Slave communicate with each other, they exchange data. When Master sends data to the slave, the slave simultaneously sends data to the master. Thus the only way the slave can send data to the master is to keep the data ready (in a hardware output buffer) and the data will be sent out when the master sends the slave some data. From a software point of view, keeping all this working correctly requires intricate and precise choreographing.

### 8.1.1 Communication Protocol

The communication between the Master and the Slave follows the following steps:

1. Master selects the slave by bringing the corresponding slave select line low.

2. Master generates the 8 clock pulses to transfer 8 bits of data (byte-at-a-time communication). As 8 bits of data go from Master to slave, 8 bits go from slave to master.

3. Previous step is repeated as many times as needed.
8.1. SPI: SERIAL PERIPHERAL INTERFACE BUS

Figure 8.1: A typical SPI bus consisting of 1 master and 3 slaves. Image source: Wikipedia.

4. Master deselects the slave by bringing the corresponding slave select line high.

Most micro controllers have built-in hardware to generate the clock pulses and to ship data in and out. However, you as programmers have to select the correct slave in software.

8.1.2 An Example

This example illustrates establishing communication between two micro controllers. **PIC16F690** has built in hardware to make it operate either as a master or as a slave. One of the **PIC16F690s** acts as the master and the other as the slave. The program itself does not do anything fancy. The master sends a character to the slave. The slave prints it using UART. To keep
wiring to a minimum I have used software UART so the PIC16F690 can communicate using the two PORTA pins that are convenient for using the UART tool that is part of PicKit2. Connect the PicKit2 to the slave and start the UART Tool and you should see the characters on the screen. You can also do the same with the master to see what is sent. If you have two PicKit2 then you can monitor both!

Note some key differences. The master calls standard SPI initialization. The slave has to call specialized initialization Spi1_Init_Advanced. Also, when slave reads data it has to send something back to the master. Hence the use of the variable dontcare. The connection between the two PIC16F690s is shown in figure 8.2.

Listing

Listing 8.1: spimaster.c

```c
// Master uses RA2 as the slave select line
#define SSLINE PORTA.F2
#define TRIS_SSLINE TRISA.F2
#define SSASSERT (0) // Assert value SS
```

Figure 8.2: Master slave connection between two PIC16F690s.
#define SSUNASSERT (!(SSASSERT))
// Use softUART so we can use RA0 and RA1 for UART communications
#define UART_PORT PORTA
#define RXPIN 1
#define TXPIN 0

void main()
{
    TRISA = TRISB = TRISC = 0xFF;
    SSSLINE = SSUNASSERT;
    TRIS_SSSLINE = 0;       // output
    ANSEL = ANSELH = 0x00;
    Soft_UART_Init(&UART_PORT, RXPIN, TXPIN, 4800, 0);
    SPI1_init();            // Initialize SPI as master.
    delay_ms(5000);         // Give time for turning on power connecting etc.
    while (1) {
        Soft_UART_Write('+');
        SSSLINE = SSASSERT;
        SPI1_Write('+');
        SSSLINE = SSUNASSERT;
        delay_ms(100);        // Long delay to debug data
    }
}

Listing 8.2: spislave.c

// Use softUART so we can use RA0 and RA1
#define UART_PORT PORTA
#define RXPIN 1
#define TXPIN 0

void main()
{
    short buffer;
    char temp;
    TRISA = TRISB = TRISC = 0xFF;
    ANSEL = ANSELH = 0;
    C1ON_bit = C2ON_bit = 0;
    Spi1_Init_Advanced(_SPI_SLAVE_SS_ENABLE,
                        _SPI_DATA_SAMPLE_MIDDLE,
                        _SPI_CLK_IDLE_LOW, _SPI_LOW_2_HIGH);
    Soft_UART_Init(&UART_PORT, RXPIN, TXPIN, 4800, 0);
    delay_ms(5000);       // Give time for turning on power connecting etc.
for (temp = 0; temp < 26; temp++)
    Soft_UART_Write(temp + 'a');
while (1) {
    temp = SPI1_Read(buffer);
    Soft_UART_write(temp);
    //light_led(temp);
}

8.1.3 Dual Digital Potentiometer

We next consider a specific IC chip that uses SPI communication, MCP4241. This is a dual potentiometer, meaning it has two potentiometers whose wipers can be moved by sending commands to the chip. This particular chip is essentially a 7 bit potentiometer meaning that the wiper can be moved from one end to the other by sending a number between 0 and 127 inclusive. In addition, it has two special values of 128 and 129 that is not used in this demonstration. We connect the two ends of the potentiometer to ground and 5 volts. As the wiper is moved from one end to the other, we get an analog voltage between 0 and 5V. In this sample application, we move one wiper from 0 to 127 and at the same time move the other wiper in the opposite direction from 127 to 0. Thus as one voltage increases the other decreases. The result is shown in figure 8.4. If a scope is not readily available then increase the delay between wiper movements to 250 milliseconds so we can monitor the voltage changes using a voltmeter.

The circuit for this demonstration is in figure 8.3.

Listing

```c
#define DELAY 1 // 1 if using scope, 250 for voltmeter

/*
 Sample code for using MCP4251

4251 is a dual digital potentiometer

It has two wipers that can be "moved" by sending a command to the chip using SPI. To move the wiper from one end to the
```
other send a command between 0 and 255. The two ends of the potentiometer are labeled A and B. When the command is 255, the wiper is connected to A and when the command is 0 it is connected to B. Note that, in terms of the command A is high and B is low, but in pinout drawings the B is on top, and A is at the bottom. Argh....

To set a wiper value, we send two bytes.

First byte:
The first byte is the control byte. Left most bits determine which wiper we are setting.
- If the left most bit is 0000 then we are setting wiper 0
- If the left most bit is 0001 then we are setting wiper 1

The next two bits are used to specify what we want to do

- If the next two bits are 00, then we are writing (setting)
- If the next two bits are 11, then we are reading. We can use this if we want to see where the wiper is.

The Last two bits are special. Not used in this demo

Second byte:
The wiper value we want to send.
DEMO CODE

In the demo code, we move wiper 0 from 0 to 255 and back to 0 in a cycle. Simultaneously, we move wiper 1 from 255 down to 0.

At the end of each cycle, program pauses with a debug LED on for debugging.

IO

RC6->CSBAR
RC7-> SDO
RB4-> SDI
RB6-> SCK

TEST CONNECTION

5V -> P0B and P1B
GND -> P0A and P1A

Use dual scope to display the voltage of the wipers
If a scope is not available, use two voltmeters.

Use DELAY=1 if using a scope, DELAY=250 if using voltmetersss

*/

/*

PicKit2 header:

RA5
RA4
RA3
RC5
RC4
RC3
RA0
RA1
RA2
RC0
CHAPTER 8. SERIAL COMMUNICATIONS

RC1
RC2
+5V
GND

+=============+
  |     |
-/-CS---1 14----VDD--
  |     |
-SCK---2 13----SDO--
  |     |
-SDI---3 12----/SHDN-
  |     |
-VSS---4 11----/WP--
  |     |
-P1B---5 10----POB--
  |     |
-P1W---6 9----POW--
  |     |
-P1A---7 8----POA--
  |     |
+=============+
MCP4251

*/

// ============= START OF PROGRAM
===========================
#define BIT(n) (1 << (n) )
// Wiring info
#define CSBAR PORTC.F6
#define CSTRIS TRISC.F6

// Specific to MCP4241
#define WIPER0 0
#define WIPER1 BIT(4)
#define WRITE_COMMAND 0
#define READ_COMMAND (BIT(3) |BIT(2)) // 00001100
#define MAXVALUE 127
// Use 127 for mcp423x
// Use 255 for mcp425x
void init_mcp4241()
{
    TRISA = TRISB = TRISC = 0xFF;
    ANSEL = ANSELH = 0;
8.1. **SPI: SERIAL PERIPHERAL INTERFACE BUS**

```c
CSBAR = 1;
CSTRIS = 0;
SPI1_Init();
}
void set_pot(int value, char wiper)
{
    // Set the value of the potentiometer 0
    // Value is a 10 bit number
    // Caller sets the wiper value as needed
    char lowbyte, highbyte;

    // split the lower and upper half
    lowbyte = value & 0xFF; // lower 8 bits
    highbyte = (value >> 8) & 0x03; // Keep just two bits
    highbyte = highbyte | WRITE_COMMAND | wiper; // mix in the commands

    // Now we are ready to write....
    CSBAR = 0; // Bring the CS line down
    Spi1_Write(highbyte);
    Spi1_Write(lowbyte);
    delay_ms(DELAY / 5 + 1); // to see it on LED
    CSBAR = 1;
}
#define DEBUG PORTC.F0
#define TRISBUG TRISC.F0
main()
{
    int value;
    ANSEL = ANSELH = 0;
    init_mcp4241();
    DEBUG = 0;
    TRISBUG = 0;
    value = 0;
    while (1) {
        set_pot(value, WIPER0);
        set_pot(MAXVALUE - value, WIPER1);
        value++;
        if (value > MAXVALUE) {
            // Turn on debug for a second
            // so we can see the change over
            DEBUG = 1;
            delay_ms(1);
            DEBUG = 0;
        }
    }
}
```
8.2 I2C: Inter-Integrated Circuit Bus

This is one of the another common serial communication busses. Characteristic features are almost identical to SPI communication

1. It is a bus and not a port. In a bus, you can connect multiple devices. Compare this to the serial port which connects one device to your micro controller.

2. It is synchronous bus. What this means is there is a clock line as part of the bus. When the data is valid on the bus is signalled by the clock.

3. The connection on the bus is based on a Master-Slave configuration. There is a bus master (usually your micro controller) which controls all the data traffic.

4. Communication can take place between the master and a specific slave. Master decides which slave it wants to communicate with.

5. Each slave must have a unique 7-bit address on the I2C bus. As a general rule, the first four bits of the address is factory set and can not be changed. Depending on the device, you may be able to change the next three bits. These three bits are generally identified as A2, A1, and A0. On most chips will have pins designated with A2, A1, and A0 and you can select desired address by either connecting the corresponding pin to logic 1 or logic 0. Some chips such as 24LC04 do not use these three bits to identify the chip, even though the chip has pins labeled A2, A1, A0.

6. Only the master can initiate communication with a slave. If a slave wants to send data to the master, it has to wait for master to establish communication (sort of having a phone line where outgoing calls are blocked!)
8.2. I2C: INTER-INTEGRATED CIRCUIT BUS

7. Communication between Master and Slave occurs over two wires.

   SDA  This is the serial data line (bi-directional)
   SCL  This is the serial clock line (also bi-directional)

8.2.1 Electrical Connection

Since both the master and the slave can communicate on the same line, how do we prevent electrical shorts when one device drives the line high while the other drives it low? This is done by the miracle of pull-up resistors. These are resistors that connect the line to the 5V supply. Typically the resistance is between 1.8K and 10K. By the specification of the protocol any device can drive this line low. The current will be limited by the pull up resistors. However, no device is allowed to drive this line high. Instead, the device can electrically isolate the line from its internal drivers, i.e. let the line float. If all devices let go of the line, then the pull up resistor will automatically pull the line high.\(^2\)

As a programmer, you achieve this by

To drive low  \texttt{PORTX.FY = 0; TRISX.FY = 0;} This sequence will bring down the line connected to \texttt{PORTX} and bit \texttt{Y}.

To float the line  \texttt{TRISX.FY = 1;} This will float the line

Note that in the above sequence, we should never write a one to the port!

8.2.2 Communication Protocol

The communication between the Master and the Slave follows the following steps:

1. Initially both SDA and SCL would be high by default

2. Master initiates the communication by bringing SDA low and then SCL low in that order.

3. Master then sends the slave’s address, an 8 bit quantity by clocking the data. This is done by first setting SDA high or low depending on the bit to be sent and then signaling by setting SCL high and then setting SCL low.

\(^2\)My mental image is a group of people pulling down a spring loaded screen. When all of them let go of the screen, the spring will pull the screen up.
4. After the 8 bits have been clocked out, the slave must acknowledge by sending back a \textit{zero}. \footnote{Note: If there is no slave to reply or slave does not respond, the pull up resistor will float the line high. So to acknowledge the receipt of data, the slave must actively participate, i.e. drive the data line down. This is true of all acknowledgement. Since the acknowledgment bit is zero, we usually say the acknowledgement uses \textit{NOT ONE}. The jargon for this \textit{NAK} for not-acknowledge. I.e. the slave acknowledges by not acknowledging! English can be a very confusing language!!}

5. Once the master has established communication with the slave, master can send information to the slave. The information is usually a pair of bytes, the first byte identifies the nature of the information and the second the data. Almost all I\textbf{2C} devices save the data in some internal register. Hence the first byte identifies the register (location) and the second the value.

6. The master can give the slave permission to talk. Once this permission is granted, the slave can send data back to the master on the same line. The master will acknowledge each byte. Eventually, the master will not acknowledge the receipt of the byte. When this happens, the master has revoked the permission to talk and the slave will now go back to listening the the master.

7. The end of communication is signified by the master sending a stop sequence. This is done by raising the SCL high with SDA low and then raising SDA line. Note that at the end of the communication, both SDA and SCL are high as at the start of the communication.

\textbf{Slave’s address:} On the I\textbf{2C} bus, you can connect any number of slaves as long as they all have different addresses. The address of the slave is an 8=bit quantity of which the bit #0 (LSB) is special. The first four bits are generally set by the manufacturer and is usually fixed for a given chip. The next three bits are generally labeled A2, A1 and A0. Some chips will have pins so designated and you can set these three bits of the address by connecting the corresponding pin to 5V or ground. \footnote{Some chips may have pins labeled A2, A1, A0 that may not be internally connected. This means these bits are not used to address the chips but are used as part of communication. In this case you can have only one such chip on the I\textbf{2C} bus} Thus in general, you can have up to 8 chips of the same kind from the same manufacturer on the
same bus by using different addresses for each chip. If you need more, you
may need to look for the same kind of chip made by a different manufacturer
who may use a different leading four bits. Given these limitations, you can
have up to 128 chips on the same I2C bus.

The last bit is special. This is not used to identify the slave. Instead
when the master sends the address with this bit cleared, it means the master
wants to talk and the slave should listen. If the master sends the address
with this bit set, then the master wants the slave to talk and in effect gives
slave the permission to talk. As mentioned before, before master sends the
address on the I2C bus, master has to send the start sequence and may have
to make several attempts before the slave acknowledges.

8.2.3 An Example

This example illustrates the use of a I2C port expander to write to an LCD
screen using just two wires. Note that every character that is sent to the
LCD display is also sent out on UART so you can debug the program in case
things don’t quite work out as you planned. The schematic for connecting
the components is shown in figure 8.5.

Listing

```c
#define BIT(n) (1<<(n))
void start_expander(char id)
{
    int attempts;
    char ack;
    for (attempts = 0; attempts < 25; attempts++) {
        Uart1_Write('!');
        Soft_I2C_start();
        Uart1_Write('@');
        ack = Soft_I2C_Write(0x40 | (id << 1));
        Uart1_Write('.');
        if (ack == 0) {
            Uart1_Write('+');
            return;
        }
    }
    Uart1_Write('X');
}
```

Figure 8.5: LCD panel controlled via a I2C port expander.

```c
void write_to_expander(char id, char location, char value)
{
    start_expander(id);
    uart1_write('#');
    Soft_I2C_write(location);
    Soft_I2C_write(value);
    Soft_I2C_stop();
}
```

```c
sbit Soft_I2C_Scl at RB6_bit;
sbit Soft_I2C_Scl_Direction at TRISb6_bit;
sbit Soft_I2C_Sda at RB4_bit;
sbit Soft_I2C_Sda_Direction at TRISb4_bit;
void init()
```
8.2. **I2C: INTER-INTEGRATED CIRCUIT BUS**

```c
{
    TRISA = TRISB = TRISC = 0xFF;
    ANSEL = ANSELH = 0;
    Uart1_init(9600);
    Soft_I2C_Init();
}

// Locations on the PORT Expander
// Page 5 of data sheet
// Default is BANK=0
// In the expander, TRIS is called DIR
// PORTS are called LAT (Latches)
#define XTRISA 0x00        // (IODIRA in the data sheet)
#define XPORTA 0x14
#define XTRISB 0x01
#define XPORTB 0x15
#define RS BIT(0)
#define RW BIT(1)
#define EN BIT(2)
#define DEBUG BIT(3)

void send_to_LCD(char rsval, char value)
{
    write_to_expander(0, XPORTB, value);
    delay_ms(1);
    write_to_expander(0, XPORTA, rsval + EN + DEBUG);
    delay_ms(1);
    write_to_expander(0, XPORTA, 0); // Bring down EN and RS
}

#define data_to_LCD(value) send_to_LCD(RS, value)
#define command_to_LCD(value) send_to_LCD(0, value)

// Generic function to send command or data to LCD
// RS=1 data. RS=0 for commands
void NH_lcd_init()
{
    // From sample code on LCD data sheet
    // See SPLC780D.pdf for complete set of commands
    command_to_LCD(0x30);
    command_to_LCD(0x30);
    command_to_LCD(0x30);  // Wake up call
    command_to_LCD(0x38);  // 8 bit interface, 2 lines
    command_to_LCD(0x10);  // Cursor Home
    command_to_LCD(0x0C);  // Turn on Display
    command_to_LCD(0x06);  // Entry Left to right
}
```
} void lcd_cls()
{
    command_to_LCD(0x02);  // clear screen
}
} void lcd_outs(char *s)
{
    for (; *s != 0; ++s) {
        data_to_LCD(*s);
        uart1_write(*s);
    }
}

void main()
{
    init();
    delay_ms(5000);
    write_to_expander(0, XTRISA, 0);  // All outputs
    write_to_expander(0, XTRISB, 0);  // All outputs
    NH_lcd_init();
    while (1) {
        lcd_cls();
        lcd_outs("Hello!");
        delay_ms(500);
        lcd_cls();
        lcd_outs("XXXXXX");
        delay_ms(150);
    }
}

8.2.4 Using 24LCXX

24LCXX are inexpensive EEPROM memory chips that use I2C protocol. Unlike other I2C devices, the device address specified by A2, A1 and A0 are not used. However, the chip has three pins marked A2, A1 and A0 for compatibility with similar products. Instead of A2, A1 and A0 that are used to identify a device on the I2C bus, these fields are used to identify a block inside the chip. Important points to note are:

1. Memory inside the chip is organized into blocks. Each block represents 256 bytes of memory. 24LC04 chip has a memory capacity of 4096 bits or 512 bytes of memory. Thus it has two blocks. 24LC08 has a capacity of 8192 bytes or a total of 4 blocks.

2. Each device is addressed by the 8-bit quantity 1010XYZW. Here XYZ is used to identify a block inside the chip. As before the W bit is used
by the master to determine if the master talks \((\bar{w}=0)\) or master listens \((\bar{w}=1)\). Thus to work with the chip and select block \(B\) master will have to create the address, where the first 4 bits are 1010, the next three are the block value and the last bit can be zero or one as appropriate. This is best done by starting with 10100000 and \textit{or-ing} it with the block value shifted left one place (so it occupies bits 3, 2, and 1) and finally or-ing the last bit. Since \(C\) does not natively support binary, the \(C\) statement to achieve it is

\[
0xA0 \mid (\text{block} \ll 1) \mid \text{lastbit}
\]

3. 24LCXX allows sequential access to memory. But the procedure to perform sequential access is same as accessing just one byte of memory. Procedure to store and retrieve data from memory are as follows:

**Storing Data:**

(a) Initiate conversation with the chip by sending a start sequence followed by the 8-bit address with \(\bar{w}=0\). As explained earlier, only the first 4 bits of the address identifies the chip. The next three bits identify the block inside the chip. \textit{Master may have to make several attempts at establishing communication before the chip responds.} For debugging purposes, it may be advantageous for your program to print something to the screen during this operation.

(b) Once the communication is established, the master sends the address of the location where master wants to store data. Thus this address together with the block selected when establishing the communication determines the location inside the chip.

(c) After fixing the location in the previous step, the master sends the data that is to be stored. If master wants to save more than one byte of data sequentially (in consecutive memory location inside the block that was specified at the start of communication), then master can send up to 16 bytes of data.

(d) After sending the data to be stored, master has to send a \textbf{I2C} stop sequence.
Retrieving Data:

(a) Initiate conversation with the chip by sending a start sequence followed by the 8-bit address with W=0. As explained earlier, only the first 4 bits of the address identifies the chip. The next three bits identify the block inside the chip. Master may have to make several attempts at establishing communication before the chip responds. For debugging purposes, it may be advantageous for your program to print something to the screen during this operation.

(b) Once the communication is established, the master sends the address of the location where master wants to retrieve data from. Thus this address together with the block selected when establishing the communication determines the location inside the chip.

(c) After fixing the location in the previous step, the master sends another start sequence except this time master sets W=1.

(d) Slave sends the data that is stored in the location that was established. If master wants to terminate data retrieval then master should not acknowledge the receipt of the data. The chip will note the lack of acknowledgment and stop talking on the I2C bus.
   However, if master wants to retrieve more data, it should acknowledge the receipt of data except for the last one. Master can retrieve up to 16 bytes of data from consecutive locations.

(e) After retrieving data from the chip, master has to send a I2C stop sequence.

An Example

This example illustrates the use of external memory, 24LCXX, to store and retrieve data. The schematic for connecting the components is shown in figure 8.5 and the program can be found in Listing 8.5

Listing
8.2. **I2C**: INTER-INTEGRATED CIRCUIT BUS

**Figure 8.6**: Schematic to use 24LCXX.

Connect RB7-RA0 if you want to use Uart tool in PicKit2 and connect PicKit2 as usual.

---

**Listing 8.5: 24LC04.c**

```c
/* Map generic names to ones specific to MikroC */
#define i2c_start      Soft_i2c_start
#define i2c_stop       Soft_i2c_stop
#define i2c_tx          Soft_i2c_write
#define i2c_rx          Soft_i2c_read

// MikroC requires us to specify the I2C connections using sbit data types

// SCL RC4
// SDA RC3
sbit Soft_I2C_Scl at RC4_bit;
sbit Soft_I2C_Sda at RC3_bit;
sbit Soft_I2C_Scl_Direction at TRISC4_bit;
sbit Soft_I2C_Sda_Direction at TRISC3_bit;

/*
About 24LC04/24LC08
24LC08 is an inexpensive EEPROM memory chip
```

About 24LC04/24LC08
24LC08 is an inexpensive EEPROM memory chip
Memory is organized into blocks. Blocks are identified by an 3 bit value.
Within each block are 256 addresses. Thus of the 7 bits of the slave address, the first four are set at the factory to 1010. The next 3 bits are the block address within the chip. Thus on any I2C bus there can be only one 24LC04/24L08 memory chip.

Capacity of 24LC04 is 4K bits (yes bits!) and has 2 blocks of 256 bytes each
Capacity of 24LC08 is 8K bits and has 4 blocks of 256 bytes each

/*
 Test program to use 24LC04/08
 Write data to memory
 Read data back from memory
 if they are equal turn on RC0
 We will write to location 55 in block 0
 functions:
 write to memory
 I2C start
 Send 24LC + block shifted one place
 Send location value
 Send data value
 I2C stop
 Read from memory
 Set address if needed
 - I2C start
   - Send 24LC + (block<<1)
   - send address
 I2C start
 send 24LC + block <<1 + 1 (+1 to ask chip to talk
 read as many data as you want (address increments automatically)
 - acknowledge after each read
 - DO NOT acknowledge after the last read (tells slave we are done)
 I2C end

Note:
Memory is slower than the processor. So if memory is busy, it will not acknowledge the initial transmission. The program should keep trying till it gets an acknowledgement. REMEMBER: Devices acknowledge by sending a ZERO not ONE!
To avoid I2C bus from hanging up the system, it is best if the program gives up after a fixed number of attempts at communication.

```c
// Start sequence
void start_24LC(char block, char lastbit)
{
    // lastbit can be either 1 or 0
    char ack, attempts;

    for (attempts = 0; attempts < 100; attempts++) {
        // Make 100 attempts
        Uart1_write('A');
        i2c_start();
        ack = i2c_tx(0xA0 | (block << 1) | lastbit);
        if (ack == 0)
            break; // break out of the loop if we get acknowledgement */
    }
    Uart1_write('B');
}

/*
 * write_24LC: Stores data to memory at a given block and address
 */
void write_24LC(char block, char address, char value)
{
    start_24LC(block, 0); // send 1010 [block]
    lastbit=0
    i2c_tx(address);
    i2c_tx(value);
    i2c_stop();
}
```
/ * read_24LC: Retrieves data from memory at a given block and address */
char read_24LC(char block, char address)
{
    char memval;

    /* Set the address */
    start_24LC(block, 0);
    i2c_tx(address);

    /* Read the data */
    start_24LC(block, 1);
    memval = i2c_rx(0);        /* Read but no ack */
    i2c_stop();
    return memval;
}

char buffer[15];
void PrByte(char x)
{
    int k;
    ByteToStr(x, buffer);
    for (k = 0; k < 3; k++)
        Uart1_write(buffer[k]);
    Uart1_write(' ');
}

void main()
{
    char address;
    char block;
    char memval;
    char value;
    TRISC = 0;
    ANSEL = 0;
    ANSELH = 0;
    PORTC = 0;
    block = 0;
    address = 3;

    Uart1_Init(9600);        // Serial at 9600 Baud
    Soft_i2c_init();
    value = 123;            // Some value
8.2. **I2C: INTER-INTEGRATED CIRCUIT BUS**

```c
delay_ms(5000);
while (1) {
    Uart1_write('>');
    Uart1_write(' ');
    write_24LC(block, address, value);   /* write
to memory */
delay_ms(50);  /* give some time for 24
  LC to write the data */
memval = read_24LC(block, address);    /* Read
  it back */
PrByte(block);
PrByte(address);
PrByte(value);
PrByte(memval);
Uart1_write(0x0D);
Uart1_write(0x0A);
delay_ms(500);
value++;
address++;
}
}

//////////////////////////////////////////////////////////////////////////
// Modifications for using 24FC256
/*
About 24LC256 /24LFC256
24LC256 is a 32K EEPROM
Addressing is a bit different.
The slave address is a full 7 bits
The first 4 bits are set at the factory to 1010. The next
four bits A2, A1, A0 can be set by the user by connecting
the corresponding pins on the chip to 5V (logic 1) or ground (logic 0)
Thus one can have upto 8 of these devices on a single
I2C bus.
To address any location, one has to send the high byte followed
by the low byte of the address.
*/
/*
Changes to use 24LC256
```
You would need two macros to extract high and low byte of a 16 bit value

```c
#define HIGHBYTE(value) (((value) >> 8) & 0xFF)
#define LOWBYTE(value) ((value) & 0xFF)
```

Assume that A0-A2 are all connected to ground. Then the slave address is 0xA0

To store the value V in location L we need to do the following

```c
i2c_start();
i2c_tx(0xA0); // May need several attempts!
i2c_tx(HIGHBYTE(L));
i2c_tx(LOWBYTE(L));
i2c_tx(V);
i2c_stop();
```

To read the value stored on location L we need to do the following

```c
i2c_start();
i2c_tx(0xA0); // May need several attempts!
i2c_tx(HIGHBYTE(L));
i2c_tx(LOWBYTE(L));
i2c_start();
i2c_tx(0xA0 + 1); // May need several attempts!
memval=i2c_rx(0);
i2c_stop();
```
Chapter 9

Applications: Smart Serial Peripherals

9.1 Smart Serial Peripherals

Smart Peripherals are stand alone units that are designed to perform a specific task. They are used in situations where a primary processor does not have the ability to perform such a task, or to relieve the primary processor the task that has to be performed. For example, typical personal computers have incredible computational power but if I want to run a DC motor using my Windows based processor, I have no way of doing so. However, if I have a peripheral device that can control all the nitty gritty details of controlling motors, then all that PC has to do is command the peripheral to perform the task. When designing the peripheral, we need to make the following design decisions:

- **What is the communication mechanism to use?** The major choices are (i) UART serial, (ii) SPI serial, (iii) I2C serial, (iv) Ethernet/WiFi using internet protocol. While the world is moving towards internet protocol, very few micro controllers have the communication hardware built in. While SPI and I2C are popular in the embedded systems world (for example port expanders and digital potentiometers use these protocols) they are not useful if we want the peripheral to be used with a personal computer or a tablet. UART serial is by far the most common.

- **What is the command structure?** Again, as designers we need
to decide the command structure and the command language that the peripheral understands. Here are some design choices:

- **Use only printable characters?** If we use only printable characters, then it is easy to use a terminal program on the PC and type the commands on the keyboard.

- **Should all commands be of the same length?** If this is the case, then it is easy to determine the end of the command.

- **Should there be a special sentinel that marks the end of the command?** Having such a sentinel, say a space character or carriage return or new line character makes it easier for humans to use the peripheral. However, it also increases the length of commands.

- **Will we have to send both commands and data?** If this is the case, then we would need a way to tell the peripheral if we are sending data or commands. One way to tackle this issue is to designate a special character as the escape character. When the peripheral receives this character, then the next character will be treated as being special (usually a command). It is best to use an infrequently used character as the escape character. Two most popular are the backslash or the escape key (ascii code 27). There are some issues you should be aware of: The backslash character is special to the C language, so it makes it cumbersome if you use C to send strings. Many windows programs treat the Esc key as a command to quit the program and many programs treat it as being special. If at all possible, avoid these two characters!

### 9.1.1 Hardware of Software UART?

This question gets asked a lot. *What should I use and what is the difference?* If you decide to use hardware UART, you do not have any choice in which pins you can use to send and receive data. However, the advantage is that all that your program has to do send a byte is load some hardware register and away it goes. Also, if you want to just check to see if there is a character to read, but not wait for input, then there are hardware flags that you can check. The primary disadvantage of hardware UART is that you cannot choose which I/O pins you can use, and sometimes these pins are used for
some other purpose. Also, being forced to use a specific pin may require you to rewire your circuit every time you decide to use hardware UART. Finally, some chips such as PIC16F1507 do not have hardware UART. In this case you are forced to use software UART. The primary disadvantage of software UART is that inputs are usually *blocking*. What this means if your program tries to read a character and no character is available, then your program comes to a standstill till a character is available to read. This may have some catastrophic consequences!

To make it convenient to use the UART tool that is included with PicKit2 (see section 13 for more details), then it is best to use RA0 to transmit and RA1. I have found that 4800 baud rate to be more reliable than 9600 when using PIC16F1507 in its default clock speed. This is best done with the following defines

```c
#define UART_PORT PORTA
#define RXPIN 1
#define TXPIN 0
```

and the following initialization code.

```c
void initserial() {
    Soft_UART_Init(&UART_PORT, RXPIN, TXPIN, 4800, 0);
}
```

One final caution when using software UART. *You need to have some pause* between characters when sending data to a serial peripheral. I found that a delay of 5-10 msec between characters is adequate.

### 9.2 Serial LCD display

This is a very useful peripheral, especially for debugging and display. This peripheral accepts characters via UART and displays it on the screen. To be useful the peripheral must also accept commands. So we need to decide on designing the command structure. Again, as mentioned before, we will avoid the backslash and escape key. I find that in most cases, I rarely use the @ key, so I will use it as the escape character. The command structure is shown in table 9.1. If @ is followed by any character other than 0, 1, G then the @ is ignored. What this means is if you want to print the @ character you need to send @@. When moving the cursor to any row and column @G is
Table 9.1: Commands for a serial LCD device

<table>
<thead>
<tr>
<th>Function</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear LCD</td>
<td>@0</td>
</tr>
<tr>
<td>Home cursor</td>
<td>@1</td>
</tr>
<tr>
<td>Move cursor to row R, col C</td>
<td>@GRC</td>
</tr>
</tbody>
</table>

followed by two characters. These characters encode the value with A being 1 and Z being 26. The top left corner of the LCD is at row=1, and column = 1.

9.2.1 Circuit Schematic

The schematic for this peripheral is shown in figure 9.1 and the software is shown in Listing 9.1. Figure 9.2 shows the display when the following string

Figure 9.1: Serial LCD display device.
9.2. SERIAL LCD DISPLAY

is sent to the device: Embedded World@GBGRocks

Figure 9.2: Serial LCD in action.

9.2.2 Code

Listing 9.1: serial-lcd.c

/*
   LCD pin out
   GND
   power
   Contrast
   RS   =>RA2
   R/W  =>RA4
   E    => RA5
   Data => PORTC (8 bit interface)
*/
#define RS PORTA.B2
#define RW PORTA.B4
#define E PORTA.B5

/* ASSUME E is always zero except as needed */

/* sendany: The lowest level function to send a byte to lcd
   Load PORTC with the byte you want to send
   raise E
   delay
   lower E
*/
void sendany(char c)
{
    PORTC = c;
    RW = 0;
    E = 1;
    delay_us(5);
    E = 0;
}

/*
  send_command: Sends a command
  RS=0
  use sendany to actually send the command
  delay
*/
void send_command(char cmd)
{
    RS = 0;
    sendany(cmd);
    delay_ms(37);
}

/*
  send_data: Sends a data to be displayed
  RS=1
  use sendany to actually send the value
  delay
*/
void send_data(char value)
{
    RS = 1;
    sendany(value);
    delay_ms(37);
}
/ * lcdinit: Send the commands as required by the datasheet */
void lcdinit()
{
    delay_ms(400); // warmup
    send_command(0x30);
    delay_ms(100); // First command needs more delay
    send_command(0x30);
    send_command(0x30);
    send_command(0x38);
    send_command(0x10);
    send_command(0x0c);
    send_command(0x06);
}

/*
 * Useful commands
 * clear: clear the LCD screen
 * move: Moves the cursor the the LCD
 */
void clear()
{
    send_command(0x01);
}

void move(char row, char col)
{
    char cmd;

    // Set DDRM address
    row = row & 1; // only 0 or 1 permitted
    col = col & 0x3F; // only 0 to 3F
    cmd = 0x80 + row * 0x40 + col;
    send_command(cmd);
}

init()
{
    PORTC = 0;
    E = 0;
    RW = 0;
    TRISA = TRISC = 0x00;
    ANSEL = ANSELH = 0;
    lcdinit();
/* send_string: Sends a string. This is a generic string printing routine */
void send_string(char *s)
{
    for (; *s != 0; s++)
        send_data(*s);
}

#define UART_PORT PORTA
#define RXPIN 1
#define TXPIN 0
#define BAUD 9600
void initserial()
{
    Soft_UART_Init(&UART_PORT, RXPIN, TXPIN, BAUD, 0);
}

#define ESCAPE_CHAR '@'
void main()
{
    char col, row;
    char c;
    char error;

    init();
    initserial();
    clear();
    send_string(" Serial Display");
    move(1, 0);
    send_string(" Ready!");
    delay_ms(2000);
    clear();
    move(0, 0);
    while (1) {
        c = Soft_Uart_Read(&error);
        if (c >= ' ' && c <= '~') { // Only accept valid ascii
            if (c != ESCAPE_CHAR)
send_data(c);
else {  // Got the ESCAPE_CHAR... get
    the next char
    c = Soft_Uart_Read(&error);
    switch (c) {
    case '0':
        clear();
        move(0, 0);
        break;
    case '1':
        move(0, 0);
        break;
    case 'G':  // Get row and column
        row = Soft_Uart_Read(&error) - 'A';
        col = Soft_Uart_Read(&error) - 'A';
        move(row, col);
        break;
    default:  // None of the above
        send_data(c);
        break;
    }
    }
}

9.3 Serial Motor Controller

Serial Motor controller are very popular. As the laptops become extremely
powerful and relatively inexpensive, one can perform control tasks that were
unimaginable even a few years ago. However, in spite of their computational
power, the output capabilities of a personal computer is limited. A Serial
Motor Controller expands the power of a PC. The basic idea is that PC will
send commands to the micro controller and the micro controller will perform
the required task. The basic architecture is shown in figure 9.3. In most
applications we want to drive two motors so we would like to have two inde-
pendent speeds. This requirement poses a challenge. PIC16F690 can only
generate one PWM signal but has built in UART. PIC16F1507 on the other
hand has four PWM channels but does not have UART. MikroC provides a
software UART library to handle those situations where the hardware UART
does not exist or cannot be used. The problem with software UART is while
the micro controller is waiting for input, it cannot do anything else without
elaborate multitasking system in place. Fortunately, in our application, there is nothing else to do.

### 9.3.1 Command Structure

We first have to design the structure of the commands. It is best to mimic established controllers. The one we use is based on the commands for a commercial controller called RobotEQ. The commands can be seen in Table 9.2. In addition to controlling motors, most applications require digital output (to engage the clutch, turn on peripherals, etc.). The design we have will have two digital output lines. Each command must be terminated by carriage
9.3. SERIAL MOTOR CONTROLLER

return and new line (to be compatible with RobotEQ). In the commands $HH$ represents a two hexadigit value that determines the speed. $00$ is stopped and $FF$ is full speed.

9.3.2 Implementation details

Not written yet

9.3.3 Schematic and Code

The schematic for serial motor controller is shown in figure 9.4.

![Figure 9.4: Schematic for a serial motor controller.](image)

Listing 9.2: serial-motor-controller.c

```c
//uncomment the next line to target PIC16F690
//#define TARGET_PIC16F690

#define UART_PORT PORTA
#define RXPIN 1
#define TXPIN 0
```
// MAP IO pins to signals
// Note: Forward and reverse are notional directions and depend on how the motor is wired and how they are mounted. Change the definitions below to match reality!

///// LEFT MOTOR
#define LMOT_FORWARD PORTC.F5
#define LMOT_REVERSE PORTC.F4
#define LMOT_ENABLE PORTC.F3

/// // RIGHT MOTOR
#define RMOT_FORWARD PORTC.F0
#define RMOT_REVERSE PORTC.F2
#define RMOT_ENABLE PORTC.F1
#define DIGITAL_OUT_1 PORTC.F6
#define DIGITAL_OUT_2 PORTC.F7

// Useful macros
#define BIT(n) (1 << (n))

// Some Debugging routines
void outs(char *s)
{
    while (*s)
    {
        Soft_UART_Write(*s);
        s++;
    }
}

void outnum(int val)
{
    char buffer[10];
    IntToStr(val, buffer);
    outs(buffer);
}

void showvelocity(char *motor, int velocity)
{
    outs(motor);
    outnum(velocity);
    outs("\r\n");
}

// Turn off ADC
void adcOff()
{
    #if defined (TARGET_PIC16F690)
    ANSEL = ANSELH = 0;
    
}
# 9.3. SERIAL MOTOR CONTROLLER

```c
#else /* */
   ANSELA = ANSELB = ANSELC = 0;
#endif /* */

} void init() /* */
{
   //Initialize the system
   TRISA = TRISB = 0xFF; // All of PORTA & PORTB are inputs
   PORTC = 0x00; //Make sure to turn off the port first!
   TRISC = 0x00; // All of PORTC is output
   adcOff();

#if !defined(TARGET_PIC16F690)
   // Initialize PWM generator. See text for explanation
   PR2 = 255; // Maximum range for duty
   PWM2CON = BIT(7) | BIT(6); // Bit7 is PWMxEN, Bit6 is PWMxOE
   PWM4CON = BIT(7) | BIT(6);
   T2CON.TMR2ON = 1;
   PWM2DCH = PWM4DCH = 0;
#endif /* */

} void initserial() /* */
{
   Soft_UART_Init(&UART_PORT, RXPIN, TXPIN, 4800, 0);
}

//---------------- Motor Control functions
void setLeftVelocity(int velocity)
{
   int direction, speed;
   showvelocity("L: ", velocity);

   //Get direction and speed
   if (velocity >= 0) {
      direction = 0;
      speed = velocity;
   } else {
      direction = 1;
      speed = -velocity;
   }
   if (speed > 255)
      speed = 255;
```
// Set the direction
if (direction == 0) {
    LMOT_REVERSE = 0;
    LMOT_FORWARD = 1;
} else {
    LMOT_REVERSE = 1;
    LMOT_FORWARD = 0;
}

// Set the speed
// PIC16F690 does not have two PWM channels
// so it is either ON or OFF.
// So any speed >127 would be on, or else off
//
// PIC16F1507 has a smoother control
#if defined(TARGET_PIC16F690)
    if (speed > 127)
        LMOT_ENABLE = 1;
    else
        LMOT_ENABLE = 0;
#else
    // Left Motor enable is connected to PWM channel 2
    PWM2DCH = speed;
#endif

// Do the same for right motor
void setRightVelocity(int velocity)
{
    int direction, speed;
    showvelocity("R: ", velocity);

    // Get direction and speed
    if (velocity >= 0) {
        direction = 0;
        speed = velocity;
    } else {
        direction = 1;
        speed = -velocity;
    }
}
if (speed > 255)
    speed = 255;

    // Set the direction
if (direction == 0) {
    RMOT_REVERSE = 0;
    RMOT_FORWARD = 1;
} else {
    RMOT_REVERSE = 1;
    RMOT_FORWARD = 0;
}

    // set the speed
    // PIC16F690 does not have two PWM channels
    // so it is either ON or OFF.
    // So any speed >127 would be on, or else off
    //
    // PIC16F1507 has a smoother control
#if defined(TARGET_PIC16F690)
    if (speed > 127)
        RMOT_ENABLE = 1;
    else
        RMOT_ENABLE = 0;
#else /* */
    //Right Motor enable is connected to PWM channel 4
    PWM4DCH = speed;
#endif /* */

char hex2nyb(char c)
{
    char val;
    val = c - '0';
    if (val > 9)
        val -= 7;
    return val & 0x0F;
}

    // States of finite state machine
#define START 0
#define SAW_BANG 1
```c
#define SAW_CMD 2
#define SAW_H1 3
#define SAW_H2 4
#define SAW_CRLF 5

// Commands
#define LFWD 0
#define LRVS 1
#define RFWD 2
#define RRVS 3
#define O1HIGH 4
#define O1LOW 5
#define O2HIGH 6
#define O2LOW 7
#define ILLEGAL -1

main()
{
    int state, cmd;
    char c;
    char h1, h2;
    int speed;
    char error;

#if !defined(TARGET_PIC16F690)
    // Set the clock speed of 1507 to 16 MHz
    OSCCON.F6 = OSCCON.F5 = OSCCON.F4 = OSCCON.F3 = 1;
#endif

    init();  /* */
    initserial();
    delay_ms(1000);  // Delay for user to connect etc.
    PORTC = 0;

#if defined(TARGET_PIC16F690)
    outs("Target: PIC16F690\r\n");
#else  /* */
    outs("Target: PIC16F1507\r\n");
#endif  /* */

/* */
    state = START;
    while (1) {
        c = soft_Uart_Read(&error);
        if (c == '\r' || c == '\n' || c == ' ' || c == ',' || c == '\t') {
```
switch (cmd) {
    case O1HIGH:
        DIGITAL_OUT_1 = 1;
        break;
    case O1LOW:
        DIGITAL_OUT_1 = 0;
        break;
    case O2HIGH:
        DIGITAL_OUT_2 = 1;
        break;
    case O2LOW:
        DIGITAL_OUT_2 = 0;
        break;
    case LFWD:
        setLeftVelocity(speed);
        break;
    case LRVS:
        setLeftVelocity(-speed);
        break;
    case RFWD:
        setRightVelocity(speed);
        break;
    case RRVS:
        setRightVelocity(-speed);
        break;
}
    state = START;
}
switch (state) {
    case START:
        if (c == '!')
            state = SAW_BANG;
        else
            state = START;
        break;
    case SAW_BANG:
        switch (c) {
            case 'A':
                cmd = LFWD;
                break;
            case 'a':
                cmd = LRVS;
                break;
            case 'B':
                break;
            case 'B':
                break;
        }
```c
  cmd = RFWD;
  break;
  case 'b':
    cmd = RRVS;
    break;
  case 'X':
    cmd = O1HIGH;
    break;
  case 'x':
    cmd = O1LOW;
    break;
  case 'Y':
    cmd = O2HIGH;
    break;
  case 'y':
    cmd = O2LOW;
    break;
  default:
    cmd = ILLEGAL;
    break;
  }
  state = SAW_CMD;
  break;
  case SAW_CMD:
    h1 = hex2nyb(c);
    state = SAW_H1;
    break;
  case SAW_H1:
    h2 = hex2nyb(c);
    state = SAW_H2;
    speed = (h1 << 4) | h2;
    break;
  }
```

Appendices
Appendix A

Elements of Embedded C

A.1 Introduction

What is Embedded C? Is it different from DevC or VisualC? The answer is "no!". First off, DevC, VisualC etc. are brand names of C compilers. Technically, they all compile standard C programs, though each compiler may have a different user interface and a different way of doing things. Embedded C on the other hand is not a brand name. Instead, it covers those aspects of the "standard" C language that is of interest to programmers of embedded systems. I use the word "standard" between quotes because often vendors of C compilers may take liberties with the language. These are often called "extensions" to the language. The true power and the longevity of the C language rests on the fact that with a handful of extensions, one can write hardware programs without resorting to assembly language.

Okay, then what exactly is Embedded C? It generally refers to those aspects of the C language that are not used by non-hardware programmers. The following are some of the commonly found features:

1. Programmers prefer to write numbers in Hex instead of decimal. For example, one would write 0x80 instead of 128, or 0xFF instead of 65535. There is absolutely no reason why you should not use decimal. It is just hex is more "natural".

2. Integers, etc. though they have all the properties of numbers, are not treated as numbers for use in arithmetic, i.e. integers are used to represent quantities that are not numbers in the mathematical sense. To
take an example from everyday life, we use numbers to represent non-
mathematical quantities. For example, we have telephone numbers. It
makes no sense to add two telephone numbers!

3. Heavy use of macros to manipulate hardware.

4. Bit level operations.

5. Minimal use of ”standard” libraries.

We will look at each of these in detail

A.2 Number Representations

Depending on the context, one uses representations other than decimal when
writing numbers. For example, instead of writing 65, one may write 0x41.
To indicate that the representation is in hex, one uses the prefix 0x (zero-ex).
Note that you can use either lowercase or uppercase x. Though not as often
as hex, one can write the same value in octal using the prefix 0 (zero) as 0101.
Octal has become somewhat obsolete in C programs. C does not provide a
mechanism for binary representations. However, many compilers provide an
extension where the prefix 0b (zero-b). In compilers with this extension, one
can write 0b01000001 for the value 65. Many compilers will allow you to
put one or more periods, so you can write 0b0100.0001. If the value 65 is
used to encode characters, then one can also write ’A’. The following table
summarizes the various representations.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Octal</th>
<th>Ascii</th>
<th>Binary (not standard)</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>0x41</td>
<td>0101</td>
<td>’A’</td>
<td>0b01000001 or 0b0100.0001</td>
</tr>
<tr>
<td>48</td>
<td>0x30</td>
<td>060</td>
<td>’0’</td>
<td>0b00110000 or 0b0011.0000</td>
</tr>
</tbody>
</table>

A.3 Integers are NOT always integers!

In embedded systems integers are used to represent bit patterns. For example
the integer 64 is used to represent the bit pattern 01000000. Note this use
of integers is valid even if the underlying hardware is not a binary computer,
though, today I am not aware of any non-binary computers. Also, integers
represent bit patterns in 2-s complement representations. Thus an 8-bit
integer value of -2 represents the bit pattern 11111110. However, if the
integer is a 16-bit integer, then -2 represents 1111111111111110. Leftmost bit is special! Note that when an 8-bit value 11111110 is promoted to a 16-bit value it becomes 1111111111111110. However, an 8-bit value 01111110 gets promoted to 0000000011111110. The left most bit is called the sign bit and the act of promotion results in the sign bit being copied. This is called *sign bit propagation*. Sometimes you don’t want the left most bit to be treated as being special. i.e. you don’t want a sign bit. C/C++ lets you specify if you want to treat the left bit special or not by the two modifiers signed and unsigned when you declare a variable. Thus if we want an 8 bit variable, x, with the left most bit special we should declare

```c
signed char x;  // x an 8 bit variable, bit #7 special
```

However, if we want to declare an 8 bit variable, y, with the left most bit NOT special, we should declare

```c
unsigned char y;  // bit #7 not special.
```

The modifier signed is optional but get into the habit of adding it. Here is a simple rule. If a variable is to be used for arithmetic it should be declared signed; otherwise it should be declared unsigned. In embedded systems, you are not likely to used signed variables for controlling hardware, so, when in doubt, declare the variable unsigned. *Sign extension can cause unexplained software failures, especially when performing right shifts.*

### A.4 Ports and Digital Input-Output (DIO)

All digital input-outputs are via I/O ports. A port is typically a collection of 8 bits. Ports are named A, B, C etc. The bits are numbered right to left starting at zero. Thus the right most bit is numbered zero, the left most bit is numbered 7. Individual pins are designated either by prefix P or R. For example, using the prefix R, the bits of PORTB will be designated as

```c
| RB7 | RB6 | RB5 | RB4 | RB3 | RB2 | RB1 | RB0 |
```

Each of these port bits will be associated with a particular pin. Note the port pin is a physical connection to which you connect your external circuitry. Generally speaking there are two types of computer hardware. Some require specialized instructions to manipulate the port bits. However, most modern microprocessors and microcontrollers map the port to some memory address. We will cover only the memory mapped ports in these notes.
A.4.1 Input pins

Some of the pins in a port may be input pins. These pins are used to read the state of external input devices. The bit associated with a port pin will reflect the state the pin. If the voltage of the pin is above a certain threshold value (typically approximately half the system voltage), this bit will be a one. If the voltage of the pin is below a threshold value (approximately half the system voltage), the bit will be zero. You typically connect a sensor (either store bought or designed in-house) to an input pin. For example, you could connect a thermostat sensor which will produce a high voltage when the temperature is high and a low voltage when the temperature is low.

How does one read the state of the input pin in C/C++? First off, you need to map the value in the port to a C variable. Most compilers will do this mapping for you in some header file so you do not have to do it (more on this later). As a rule, the name of the variable will be the same as the name of the port, often written in all caps. Next, we need some way to isolate the individual bit from the other bits in the port using the & operation. As far as the operation is concerned, each bit is treated as a logical variable, with one representing true and zero representing false. In logic, when you use the operation & of two bits, the result is true only if both the bits are true. For example if we are interested in checking bit 5 of a port, we & value in the port with 0b0010.0000 and check the result. If the result is zero, then the bit we are checking is zero; otherwise the bit we are checking is one. (shown below)

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>V7</td>
<td>V6</td>
<td>V5</td>
<td>V4</td>
<td>V3</td>
<td>V2</td>
<td>V1</td>
<td>V0</td>
</tr>
<tr>
<td>Mask</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Result of &amp;</td>
<td>0</td>
<td>0</td>
<td>V5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Conveniently, C/C++ treats zero as false and anything else as true. Thus in C we will write

```c
if (PORTA & 0x20) {
    //Code to perform when the bit #5 is set
}
else {
```
// Code to perform when the bit is not set
}

Note that if your compiler allows writing numbers in binary then you may write

if (PORTA & 0b00100000){ // Use this if possible
    //Code to perform when the bit #5 is set
} else {
    // Code to perform when the bit is not set
}

A.4.2 Output pins

You use output pins to control external devices. Your program controls the voltage of an output pin by changing the bit associated with the pin. If your program sets the bit to one, then this will cause the voltage of the pin to go high (equal to system voltage); if your program sets the bit to zero, then the voltage of the pin will go to zero. You can then use the pin to drive any external electronics such as light emitting diodes, dc motors, relays, buzzer etc.

How do you turn on a specific bit in a port (make it a one)? When turning on a specific bit, it is extremely important that the other bits are not modified in anyway. This is especially true if some other bit is associated with another output pin that is connected to a different device. Clearly, you don't want to design a system that will turn on a motor, every time you ring a buzzer! How do we make sure that only one bit in a port is changed without affecting other bits in the port? This depends on the nature of the port. In some cases it is possible to read back the value of a bit associated with an output pin. We call such a port a registered port. In some cases, it may not be possible to read back the value of a bit associated with an output
pin (unregistered port). If a port is not necessarily registered, then one keeps a copy of the port in a variable. Then, instead of manipulating the port, one modifies the copy first and then transfer the copy to the port. To turn on a particular bit, we use the or operator denoted by |. This operator is similar to the & operator, except that result of performing an | of two logical values would be true if either one of them is true. Thus, if we want to turn on bit #2 of some value, we perform an | operation with 0b0000.0100 as shown below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>V7</td>
<td>V6</td>
<td>V5</td>
<td>V4</td>
<td>V3</td>
<td>V2</td>
<td>V1</td>
<td>V0</td>
</tr>
<tr>
<td>Mask</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Result of</td>
<td>V7</td>
<td>V6</td>
<td>V5</td>
<td>V4</td>
<td>V3</td>
<td>1</td>
<td>V1</td>
<td>V0</td>
</tr>
</tbody>
</table>

Note that in the result none of the other bits are affected but bit #2 becomes a one.

**How do you clear a specific bit in a port (make it a zero)?** As before, if the port is registered, we can directly work with the value in the port. If not, we will have to work with a copy of port. In either case, we use the & operation. For example if we want to make bit #3 zero without affecting any other bit, we use the mask 0b1111.0111 when performing & as shown below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>V7</td>
<td>V6</td>
<td>V5</td>
<td>V4</td>
<td>V3</td>
<td>V2</td>
<td>V1</td>
<td>V0</td>
</tr>
<tr>
<td>Mask</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Result of &amp;</td>
<td>V7</td>
<td>V6</td>
<td>V5</td>
<td>V4</td>
<td>V3</td>
<td>0</td>
<td>V2</td>
<td>V1</td>
</tr>
</tbody>
</table>

Note that all the bits except bit #3 are unchanged and bit #3 is cleared. Note that in defining the mask, the bit complement operator is useful.

**Bit level operators**

When operating at the bit level, C/C++ provides a number of useful operators that are described below.
### A.5 Bit Fields

*Bit fields* is a way to access individual bits. Although bit fields are defined in the C language, the actual construction is left to the programmer. A characteristic feature of Embedded C is the extensive use of bit fields. Most compiler vendors will provide names for each bit and depending on the compiler, these names could be used as part of any entity or only as part of hardware specific quantities. For example, suppose that we have a variable (or a hardware quantity) called `ALPHA` and suppose we want to work with bit #1 and the fields are named by the letter `B` followed by the bit number. The following table shows how the bit fields are used.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-wise and</td>
<td>&amp;</td>
<td>x &amp; y</td>
<td>Each bit of x is logically and-ed with the corresponding bit of y. If x and y do not have the same number of bits, then the one with fewer bits is extended with or without sign extension.</td>
</tr>
<tr>
<td>Bit-wise inclusive or</td>
<td></td>
<td>x ∣ y</td>
<td>Similar to &amp; except bitwise inclusive-or operation is performed.</td>
</tr>
<tr>
<td>Bit-wise exclusive or</td>
<td>^</td>
<td>x ^ y</td>
<td>Similar to &amp; except bitwise exclusive-or is performed.</td>
</tr>
<tr>
<td>Bit-wise left shift</td>
<td>&lt;&lt;</td>
<td>x &lt;&lt; 3</td>
<td>Bit pattern in x is shifted left 3 places. The vacated bits are zeroed.</td>
</tr>
<tr>
<td>Bit-wise right shift</td>
<td>&gt;&gt;</td>
<td>x &gt;&gt; 5</td>
<td>Bit pattern in x is shifted right 5 places with or without sign extension depending on whether x is declared signed or unsigned.</td>
</tr>
<tr>
<td>Bit-wise complement</td>
<td>~</td>
<td>~ x</td>
<td>The result is bitwise complement of the bits of x</td>
</tr>
</tbody>
</table>
APPENDIX A. ELEMENTS OF EMBEDDED C

### Operation Without bit fields with bit fields

<table>
<thead>
<tr>
<th>Operation</th>
<th>Without bit fields</th>
<th>with bit fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set bit #1</td>
<td>ALPHA</td>
<td>= 0b00000010;</td>
</tr>
<tr>
<td>Clear bit #1</td>
<td>ALPHA &amp;= ~0b00000010;</td>
<td>ALPHA.B1=0;</td>
</tr>
<tr>
<td>Test bit #1</td>
<td>if (ALPHA &amp; 0b00000010)</td>
<td>if(ALPHA.B1)</td>
</tr>
</tbody>
</table>

Note that the use of bit fields makes the code easier to read and also less error prone. Check the documentation of your C compiler to see if bit fields have standard names. In MikroC the bit fields are named with prefix B or F.

### A.6 Use of Macros

Embedded C programmers make extensive use of macros to hide hardware details. Suppose that in a project, one has a warning light connected to bit #5 of PORTC, an alarm input on bit #6 in PORTB and in your code you want to turn the warning light on when alarm input is high, or else you want to turn the warning off. Using bit fields you may write

```c
// some code
if (PORTB.B6) {
    PORTC.B5 = 1;
    // Code when alarm is high
} else {
    PORTC.B5 = 0;
    // Code when alarm is low
}
```

The above code leaves a lot to be desired. First off, you need extensive comments to know what exactly PORTB.B6 is. And for some very good reason, the circuit changes and the alarm gets connected to bit #3 in PORTA. You have to go and make sure that you make changed everywhere. One oversight, your program will mysteriously fail. Macros cleanly solve this problem as shown below

```c
// somewhere at the top of the file
// or in an included file (preferable)
```

Note: Check the documentation of your C compiler to see if bit fields have standard names. In MikroC the bit fields are named with prefix B or F.
A.6. USE OF MACROS

// define the following
#define WARNING PORTC.B5
#define ALARM PORTB.B6

// Now your code will look like this
// some code
if (ALARM) {
    WARNING = 1;
    // Code when alarm is high
} else {
    WARNING = 0;
    // Code when alarm is low
}

Often times macros are used to hide complexity. Suppose that in the above code we want the alarm action to be activated when PORTB.B6 is high, or if PORTA.B5 is low and PORTA.B2 is high a straightforward code will read

if ((PORTB.B6) || (!PORTA.B5 && PORTA.B2)) {
    // Alarm action
}

A cleaner way to achieve the same is to use macros as follows

#define ALARM ((PORTB.B6) || (!PORTA.B5 && PORTA.B2))
// more code
if(ALARM) {
    // Alarm action
}
Appendix B

Using PicKit2 UART tool

B.1 Using UART tool

UART tool that is part of PicKit2 is a convenient terminal emulation program that allows you to send and receive data to your PIC16F690 from your PC. You wire your PicKit2 as shown in the table B.1. See also figure B.1.

**Note:** If your PicKit2 is permanently wired to RA0 and RA1 then make sure in your code to set RA0 and RA1 as input pins. Making them output pins can lead to unexplained failures and burnt chips! One way to protect the chip is to connect RB7 to RA0 with a 1K resistor and RB5 to RA1 with another 1K resistor. If you use programming headers, it is best to have two headers, one for programming and the other for Usart tool.

<table>
<thead>
<tr>
<th>PicKit2</th>
<th>PIC16F690</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NC</td>
<td>Okay leave this connected to RA3</td>
</tr>
<tr>
<td>2</td>
<td>Vdd</td>
<td>This has to be connected!</td>
</tr>
<tr>
<td>3</td>
<td>Vss</td>
<td>Ground</td>
</tr>
<tr>
<td>4</td>
<td>RB7</td>
<td>Okay to be connected to RA0 (see below)</td>
</tr>
<tr>
<td>5</td>
<td>RB5</td>
<td>Okay to be connected to RA1 (see below)</td>
</tr>
</tbody>
</table>
APPENDIX B. USING PICKIT2 UART TOOL

Figure B.1: Typical connection for using PicKit2 Usart-tool

- Start the UART tool from the main menu as shown in figure B.2
- You should see the tool as in figure B.3. Select the baud rate, turn on VDD and then connect using the connect button.
- What ever you type will be sent to PIC16F690 and the output of the PIC16F690 will show up on the terminal.
- You can view the output in ascii (default) or on Hex.
- Echoing what you type can be confusing. If that is the case, turn off the local echo (Echo On)
Figure B.2: Select UART tool from the main menu.
Figure B.3: UART tool screen. Wiring is also shown at the bottom left.