www.electronicspk.com

Teach Yourself

PIC Microcontrollers

For Absolute Beginners

M. Amer Iqbal Qureshi

icrotronics Pakistan

About This Book

This book, is an entry level text for those who want to explore the wonderful world of microcontrollers. Electronics has always fascinated me, ever since I was a child, making small crystal radio was the best project I still remember. I still enjoy the feel when I first heard my radio. Over the period of years and decades electronics has progressed, analogs changed into digital and digital into programmable.

A few years back it was a haunting task to design a project, solely with gates and relays etc, today its extremely easy, just replace the components with your program, and that is it.

As an hobbyist I found it extremely difficult, to start microcontrollers, however thanks to internet, and excellent cataloging by Google which made my task easier.

A large number of material in this text has its origins in someone else's work, like I made extensive use of text available from Mikroelectronica and other sites.

This text is basically an accompanying tutorial for our PIC-Lab-II training board.

I wish my this attempt help someone, write another text.

Dr. Amer Iqbal

206 Sikandar Block

Allama Iqbal Town Lahore

Pakistan

ameriqbalqureshi@yahoo.com

Acknowledgment

I am extremely thankful to my family, specially Shamim, who spent hours making Tea and Cofee to make me working and for her love and support. My three kids, Osama, Taha and Muneeb, who have been a source of inspiration and energy, always playing with me and making the life a wonderful experience.

I would like to thank my friend, Mr. Haroon Rashid, who really made a nice effort to get me started, I am thankful to Mr. Shahid Afridi, who despite being a busy officer, spends a lot of time on his hobby, his thought provoking ideas are really appreciated.

To My Mother,
Prof. Razia Dr. Razia Iqbal
and my Father,
Late Prof. Dr. M. Iqbal Qureshi
who really did an excellent job, in my training and inspiration.
May Allah bless them.

Table of Contents

Introduction to Microcontrollers	6
Understanding Hardware	27
Setting up the Programmer	35
Setting Up Proton Basic Compiler	38
Basic Programming Language … A Primer	41
I/O Ports	48
Writing Your First program	52
Reading Switches	57
Using Graphic LCD	66
Asynchronous Serial Communication	70
Sound and Digital Signals	79
Analog Module	88
On-Chip EEPROM	94
On-Chip CCP Capture | Compare | PWM	98
Pulse	103
Interrupts	105
Timers and Interrupts	109
I2C Communication	117
Basic Electronics	120
Expanding Microcontroller I/O Lines	124
H-Bridge and DC Motors	126
Stepper Motors	128
Real Time Clock	130
Making a frequency Counter	133
Working with Matrix LED Displays	139
MPLAB® and ICD-2	144
Using Boot Loader	145

Chapter 1
Introduction to Microcontrollers

Welcome to the wonderful world of microcontrollers. I presume that you are reading this text because you are interested in learning and exploring microcontrollers. As you might be aware micro-processors in general and micro-controllers in particular have substantially changed the electronics today. Now electronic devices and circuits are not designed as electronic connections, but as software run within the microcontrollers. So electronic devices today are the blend of hardware and software.

This book will take you through all the steps necessary to learn and explore PIC-Microcontrollers. We shall remain confined to a particular class of microcontrollers, yet chances are that after mastering this you will find migration to other devices quite a bliss. This manual has been written specifically as a companion to our PIC-Lab-II a microcontroller development board.

These small devices have revolutionized the world of electronics. Today microcontrollers are everywhere, think of a device and you will find a microcontroller somewhere in it. May it be your remote control, air conditioner, microwave oven, DVD player, television or cell phone all have a microcontroller sitting inside. These small devices can do so much, that only imagination is the limit. Moreover they are very simple to use, you don't need to be an expert in electronics to use them in your next project. A basic understanding of electronics, and digital circuits is all that is required to get started. Once you are in the business, sky is the limit. Think of any logical application and you will find microcontroller handling the job nicely.

Industrial automation including automatic assembly lines, robots and quality control systems all are backed by some kind of microcontroller.

What is a Microcontroller?

So exactly what is a microcontroller or a microprocessor? This is the question which needs to be clarified before putting the heads down. As a hobbyist or as a student of electronics you must have come across a number of integrated circuits. These are small devices, with lots of circuitry inside them, having few connections for external communication. However all these integrated circuits differ from each other, in terms of function. The circuit inside an integrated circuit, may it be digital or analog, is purpose designed. Like 555, a very popular timing IC, has all the necessary circuitry inside to make various types of oscillators. Similarly a 7447 is a binary to 7-segment decoder, and has input pins to accept binary coded decimal (BCD) number, the output pins will then turn on and off accordingly to display the number on a 7-segment display. So on an so forth, you come across hundreds and thousands of ICs with specific functions. In order to get an application work, you must know specifically the function, input and output requirements of the particular integrated circuit.

Microcontrollers and microprocessors are integrated circuits, but they differ fundamentally from other ICs. They are a class in themselves, that the designers have not made them to do a particular job. As such when you buy them from the market, you can not specify what function it will do. In order to get some useful function, these ICs have to be configured. Thus a microprocessor or microcontroller can be configured to check the status of a button, and then turn a motor ON or OFF. While the same IC can be configured later, to read the status of an infra-red sensor, decode the signal and turn another device ON or OFF. If these two types of circuitries were to be made using conventional digital ICs, it would have required a large number of components. Moreover any change in the specification, like change of Infra-Red codes would result in total change in design! Using a configurable IC, is a great idea. Not only the same IC, can be configured to

do different tasks, but a change in specifications can easily be implemented by just changing the device configuration. This greatly facilitated the engineers and hobbyists to rapidly develop new electronic devices, and continuously improve previous ones. Not only the hardware requirements decreased, but also design time, and time to market were decreased.

Microcontrollers and microprocessors therefore took over the market. Large hardware designs were reduced, and most of the circuitry was replaced by the configuration scripts. Today we call this ability to configure a microprocessor or microcontroller, programming.

A program is nothing but a series of instructions, in a correct and logical manner to instruct the microprocessor respond to various inputs. By changing the program, the behavior of microcontroller will change. Think of it as a music system. The manufacturer has not designed it to produce any particular sounds out of its speakers. Yet it has all the necessary circuitry to do that. What music it will produce would depend upon the tape, or CD inserted. Thus you change the CD, and the same hardware is playing different thing. So we can say that the music system, is a programmable device, and the information stored on tape, or CD is the program, or instructions to help the music system, make sounds.

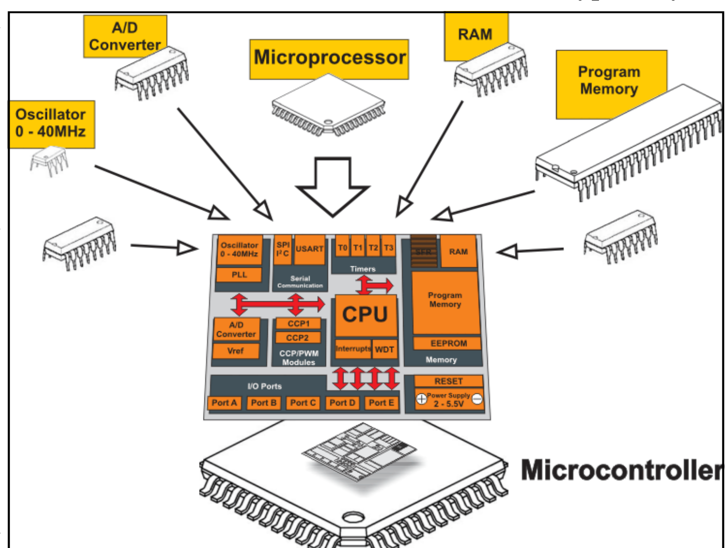
Similarly microprocessors and microcontrollers, are programmed to do a job. The job can be changing a TV channel to controlling complex movements of a robot. All these applications have a microcontroller doing its specific job. It can be astonishing to find the same microcontroller in the remote control, and the robot. In one place it is driving an infra-red LED and in other it is driving the motors.

Take another example. Consider plain paper and pencil. Now you have a choice of 26 alphabets, 0-9 numbers and few others like space, full stop etc. that is it. Not much hardware, only paper and pencil, and not much choice of letters, just 26 + few more. What you can do with it. You can do miracles. Write a complete thesis, a poem, a novel, an essay or what not. It all depends how you organize those letters. Using the pencil and paper. So the same hardware serving thousands of different jobs. The choice of letters are the instructions you can give, and paper is your microcontroller, whereas pencil is a device through which you transfer the idea in your mind, to the paper. Once transferred you do not need the pencil, to use the book, or notebook.

This example fits exactly on the scenario of microcontrollers and microprocessors. Thus you have to learn the instructions your particular microcontroller understands, and what those instructions order it to do. Then its your mind, and ideas how you play with these instructions to get your job done. Literally there are hundreds of methods to get the same job done. Just like in English, there many ways you can arrange the alphabets, to convey the same message.

Difference between a Microprocessor and Microcontroller

Essentially these two devices are similar, but with a little bit of difference. A CPU which is the heart of these devices needs a host of external devices to make it communicate with real-world. A typical system would need a system to read the inputs from keyboard, and write outputs to a terminal, store intermediate processing data into some memory, and to keep permanent information into some safe place. These devices which are independent circuits, work in harmony with the CPU, to make one system. In a typical Personal Computer these devices are attached to the CPU, using hard-wired connections. This makes the system more flexible, that means you can add more memory, change capacity of hard drives, add or remove CD-ROMs, sound cards etc.



A microcontroller on the other hand is made up of most of these devices built exactly within the same package. Your microcontroller will therefore contain, the CPU, RAM, ROM, Timers, I/O etc. all packed within one integrated circuit. This facilitates the development process, as well as reduce the requirements of external components, however this also means

you can not change, the number and type of integrated devices. The applications where a microcontroller will be used, vary. They are usually quite simple, and do not require as much processing power as a PC does, so the microcontrollers with varying amounts of RAM, ROM, I/O lines and timers etc have been made available. Essentially all are almost same, and they only vary in the number of resources available on them. So for a particular application you chose a microcontroller, not the one which has maximum resources, but the one which has just enough to do the job.

Thus a microcontroller is a complete, small scale computer with all the necessary devices on-board. All you need is the external hardware, which you want to drive, like sensors and motors etc.

Why there are to many different Microcontrollers?

Well after the idea of having a programmable device, many electronics manufacturers took the idea to develop their own chip. The internal architecture therefore differs among the manufacturers but from our point they are almost similar. Like there are so many different car manufacturers, Toyota, Suzuki, Honda, Mercedes and so on. Each one manufactures the cars with their own internal technologies, their engines, aerodynamics, peripherals all are different in specifications, yet if you can drive one car, chances are you will not find it difficult to drive another, is that not so. Despite being different in power, cylinders, valves, type of fuel etc, yet they have the same basic architecture and same basic theme.

So learning one microcontroller facilitates learning the other. Moreover the same company manufactures many different microcontrollers, which are all almost compatible. This is again like an automobile company. They make cars for many different types of users. Some bigger while others smaller. In addition to cars, they also manufacture other locomotives, like vans, truck and buses etc. All these have similar idea, but the nature of job they are required to do is different. Similarly in electronics the requirements of the project vary. For example to make a security device, you need little memory, whereas to make a data logger you need lots of memory. A remote control will not need to display data on LCD, so needs lesser number of I/O lines, whereas an industrial control unit will need to display its data, and therefore needs more I/O lines. A calculator needs only digital input, whereas a temperature controller needs to acquire analog data. These differences in requirements, makes the manufacturers produce different microcontrollers with different memory size, number of I/O lines and number of integrated peripheral devices. Otherwise they are all similar to use. Again, if you have mastered one, its easy to migrate to another. So the type of microcontroller to be used in a given project will be determined by the exact requirements.

How did Microcontrollers evolve?

The situation we find ourselves today in the field of microcontrollers had its beginnings in the development of technology of integrated circuits. This development has enabled to store hundreds of thousands of transistors into one chip. That was a precondition for manufacture of microprocessor and the first computers were made by adding external peripherals such as memory, input/output lines, timers and others to it. Further increasing of package density resulted in creating an integrated circuit which contained both processor and peripherals. That is how the first chip containing a microcomputer later known as a microcontroller was developed.

In the year 1969, a team of Japanese engineers from BUSICOM company came to the USA with a request that a few integrated circuits for calculators were to be designed according to their projects. The request was set to INTEL company and Marcian Hoff was in charge of the project there. Since having been experienced in working with a computer PDP8, he came to an idea to suggest fundamentally different solution instead of suggested design. That solution presumed that the operation of integrated circuit was to be determined by the program stored in the circuit itself. It meant that configuration would be simpler, but it would require far more memory than the project proposed by Japanese engineers. After a while, even though the Japanese engineers were trying to find an easier solution, Marcian's idea won and the first microprocessor was born. A major help with turning an idea into a ready-to-use product, Intel got from Federico Faggin. Nine months after his arrival to Intel he succeeded in developing such a product from its original concept. In 1971 Intel obtained the right to sell this integrated circuit. Before that Intel bought the license from BUSICOM company which had no idea what a treasure it had. During that year, a microprocessor called the 4004 appeared on the market. That was the first 4-bit microprocessor with the speed of 6000 operations per second. Not long after that, American company CTC requested from Intel and Texas Instruments to manufacture 8-bit microprocessor to be applied in terminals. Even though CTC gave up this project at last, Intel and Texas Instruments kept working on the microprocessor and in April 1972

the first 8-bit microprocessor called the 8008 appeared on the market. It was able to address 16Kb of memory, had 45 instructions and the speed of 300,000 operations per second. That microprocessor was the predecessor of all today's microprocessors. Intel kept on developing it and in April 1974 it launched 8-bit processor called the 8080. It was able to address 64Kb of memory, had 75 instructions and initial price was $360.

In another American company called Motorola, they quickly realized what was going on, so they launched 8-bit microprocessor 6800. Chief constructor was Chuck Peddle. Apart from the processor itself, Motorola was the first company that also manufactured other peripherals such as 6820 and 6850. At that time many companies recognized greater importance of microprocessors and began their own development. Chuck Peddle left Motorola to join MOS Technology and kept working intensively on developing microprocessors.

At the WESCON exhibition in the USA in 1975, a crucial event in the history of the microprocessors took place. MOS Technology announced that it was selling processors 6501 and 6502 at $25 each, which interested customers could purchase immediately. That was such sensation that many thought it was a kind of fraud, considering that competing companies were selling the 8080 and 6800 at $179 each. On the first day of exhibit, in response to the competitor, both Motorola and Intel cut the prices of their microprocessors to $69.95. Motorola accused MOS Technology and Chuck Peddle of plagiarizing the protected 6800. Because of that, MOS Technology gave up further manufacture of the 6501, but kept manufacturing the 6502. It was 8-bit microprocessor with 56 instructions and ability to directly address 64Kb of memory. Due to low price, 6502 became very popular so it was installed into computers such as KIM-1, Apple I, Apple II, Atari, Commodore, Acorn, Oric, Galeb, Orao, Ultra and many others. Soon appeared several companies manufacturing the 6502 (Rockwell, Sznertek, GTE, NCR, Ricoh, Commodore took over MOS Technology). In the year of its prosperity 1982, this processor was being sold at a rate of 15 million processors per year!

Other companies did not want to give up either. Frederico Faggin left Intel and started his own company Zilog Inc. In 1976 Zilog announced the Z80. When designing this microprocessor Faggin made the crucial decision. Having been familiar with the fact that for 8080 had already been developed he realized that many would remain loyal to that processor because of great expenditure which rewriting of all the programs would result in. Accordingly he decided that a new processor had to be compatible with the 8080, i.e. it had to be able to perform all the programs written for the 8080. Apart from that, many other features have been added so that the Z80 was the most powerful microprocessor at that time. It was able to directly address 64Kb of memory, had 176 instructions, a large number of registers, built in option for refreshing dynamic RAM memory, single power supply, greater operating speed etc. The Z80 was a great success and everybody replaced the 8080 by the Z80. Certainly the Z80 was commercially the most successful 8-bit microprocessor at that time. Besides Zilog, other new manufacturers such as Mostek, NEC, SHARP and SGS appeared soon. The Z80 was the heart of many computers such as: Spectrum, Partner, TRS703, Z-3 and Galaxy.

In 1976 Intel came up with an upgraded version of 8-bit microprocessor called the 8085. However, the Z80 was so much better that Intel lost the battle. Even though a few more microprocessors appeared later on the market (6809, 2650, SC/MP etc.), everything was actually decided. There were no such great improvements which could make manufacturers to change their mind, so the 6502 and Z80 along with the 6800 remained chief representatives of the 8-bit microprocessors of that time.

The PIC Microcontroller

Although microcontrollers were being developed since early 1970's real boom came in mid 1990's. A company named Microchip® made its first simple microcontroller, which they called PIC. Originally this was developed as a supporting device for PDP computers to control its peripheral devices, and therefore named as PIC, Peripheral Interface Controller. Thus all the chips developed by Microchip® have been named as a class by themselves and called PIC. Microchip® itself does not use this term anymore to describe their microcontrollers, however use PIC as part of product name. they call their products MCU's.

A large number of microcontroller designs are available from microchip. Depending upon the architecture, memory layout and processing power. They have been classified as low range, mid range, high range and

now digital signal processing microcontrollers.

The beauty of these devices is their easy availability, low cost and easy programming and handling. This has made PIC microcontrollers as the apple of hobbyists and students eyes.

We shall be talking about mid-range PIC microcontrollers, and use PIC18F452 as a prototype in this manual to explore them. Knowledge gained by learning and exploring one microcontroller is almost 90% applicable on other microcontrollers of the same family. The only difference is in availability of resources on different chips.

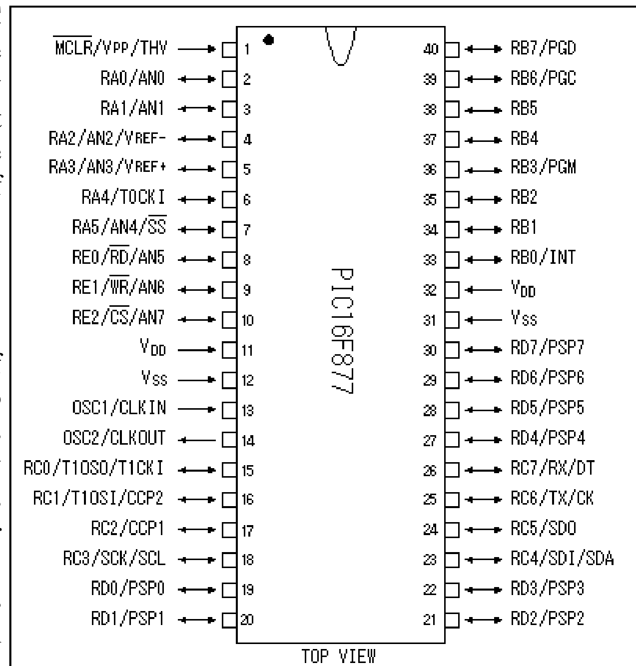


Fig-2 Showing Pin Outs of PIC-16F877 Microcontroller

General Organization of PIC Microcontrollers

Although we shall talk in detail on various aspects of these chips in relevant sections, here I would like to give a brief introduction on the overall business involved. Fig-2 shows the pin out details of a very popular 40-pin PIC microcontroller, PIC16F877. as you can see that each pin has been assigned a number of functions. Sometimes two and sometimes three. This situation is very common in microcontrollers, as there is always more which your microcontroller can offer, yet the number of pins on a given package is limited.

In a given circuit/application a pin is usually tied to a specific job, and all functionality of a pin is usually not required, however you make opt to use the specific pin your own way.

The specific function of a pin is selected by configuring various bits of internal registers. The number and names of these special function registers (SFRs) vary from device to device as some devices have limited functionality while others have more. Nevertheless if we are talking about a function which is present in both devices, its SFR will be same. The selection and settings of these SFR's is the key to successful programming. It is therefore mandatory to go through the data sheets of the device before starting a project.

Second important thing to know is that the devices with same number of pins (from microchip®), are all pin-compatible. Which means if you design a project for 40 pin PIC microcontroller, and later want to replace the chip with another 40 pin PIC the pins are all compatible. It is also good to know that a pin labeled as lets say RB0 is located on pin 33 of PIC 16F877, but the same pin is available on pin 6 in 18 pin PIC16F628. the pins are functionally same, as long as their names are same. So if you develop a project while experimenting on 18F452 using pin RB0, after successful testing you want to transport the project to an 18 pin device, which also has RB0 on it, apart from pin number on package, and recompiling the program, you don have to bother much about anything else.

Power Supply

PIC microcontrollers use TTL logic, and therefore expect a well regulated 5V power supply. The supply may however range from 3.5V to 5.5V. These microcontrollers require very small amount of current. Indeed these devices have been labeled as nano-watt technology devices. The logical levels are also same, a signal from 0 to about 2V is considered as logical '0' and a signal from 3.5V to 4.5V is considered as logical '1'. In order to communicate with devices using higher logical voltages, consider level conversion.

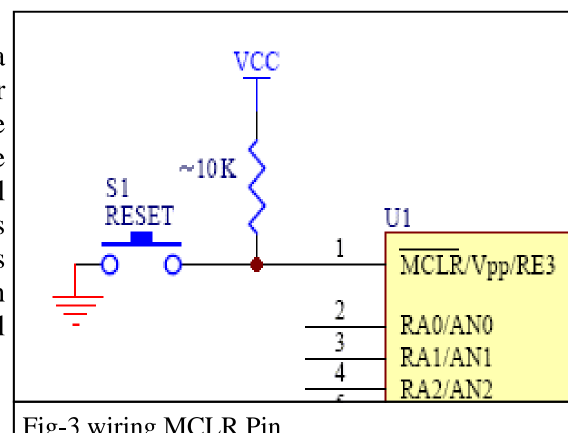


Fig-3 wiring MCLR Pin

MCLR , Master Clear

On every PIC microcontroller you will find a pin labeled as MCLR. This pin has two basic functions. It is

used to reset the microcontroller, like soft-boot. As well as to put the microcontroller into programming mode. The MCLR pin when connected to ground, will reset the microcontroller, and keep it in reset state, till the ground connection is released. After that the microcontroller will have all its RAM reset, and program execution will begin, just like the system has been just powered on. A 10K pull up resistor is usually connected with the pin, to keep it high when reset switch is released.

The same pin will also work as program mode pin. When a new software is to be downloaded into the microchip, about 12V are applied to the MCLR pin, by your programming device. This can be done right in your circuit, or by taking the IC out of circuit and putting it into the IC socket on your programmer. We shall talk more about this in section on programming. The 10K resistor is then useful to avoid 12V reaching VCC and therefore to other devices.

Analog and Digital Data

Our microprocessors use digital data to represent everything. Even music, videos and images all are represented as digital data, which is a series of logical '0' and '1'. However our real world data is not digital. It is rather analog. It is rightly said, "We live in an analog world, but process the data in digital world". Real world data like light, temperature, pressure, heat, height, distance, speed, force etc. all are analog data. In order to utilize these data we have to acquire them with specific sensors or transducers and then convert into digital format for use within microprocessor's digital world. Many other microcontrollers require an external ADC chip to implement this, however this feature has been nicely built into PIC microcontrollers. The number of Analog channels will vary among devices and some devices will not have this feature on-board. Pins labeled as AN0, AN1 etc are for analog data if required, however they can also function as normal digital pins to work with digital data. As previously said this selection is made by configuring specific registers in microcontroller.

BASIC CONCEPTS

Did you know that all people can be classified into one of 10 groups- those who are familiar with binary number system and those who are not familiar with it. You don't understand? That means that you still belong to the later group. If you want to change your status read the following text. Text describing briefly some of the basic concepts used further in this book (just to be sure that we discuss the same issues).

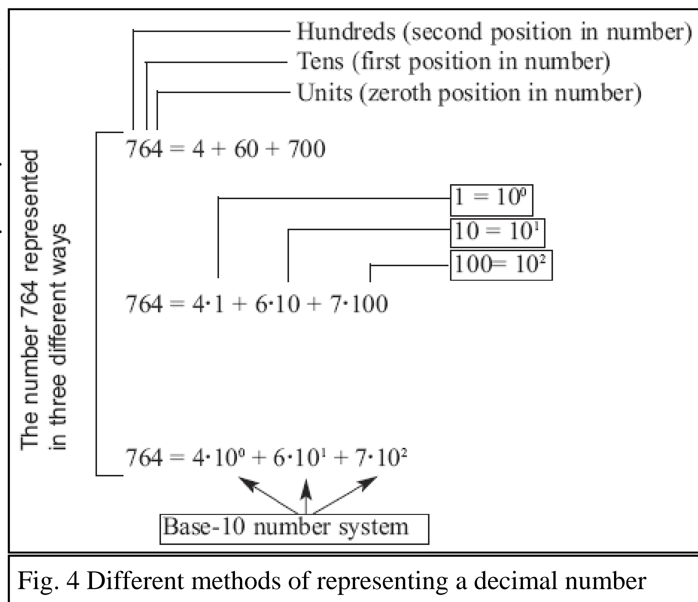


Fig. 4 Different methods of representing a decimal number

World of numbers

Mathematics is such a good science! Everything is so logical and is as simple as that. The whole universe can be described with ten digits only. But, does it really have to be like that? Do we need exactly ten digits? Of course not, it is only a matter of habit. Remember the lessons from the school. For example, what does the number 764 mean: four units, six tens and seven hundreds. Simple! Could it be described in a bit more complicated way? Of course it could: 4 + 60 + 700. Even more complicated? Naturally: 4*1 + 6*10 + 7*100. Could this number look a bit more "scientific"? The answer is yes: 4*10^0 + 6*10^1 + 7*10^2. What does it actually mean? Why do we use exactly these numbers: 100, 101 and 102 ? Why is it always about the number 10? That is because we use ten different digits (0, 1, 2, ... 8, 9). In other words, because we use base-10 number system, i.e. decimal number system. It is easier to work with decimal numbers, however computers can not do so, they use only two digits, 0 and 1. these are represented within a computer by presence or absence of volts on a specific line.

Binary number system

What would happen if only two digits would be used- 0 and 1? Or if we would not know to determine whether something is 3 or 5 times greater than something else? Or if we would be restricted when comparing two sizes, i.e. if we could only state that something exists (1) or does not exist (0)? Nothing

special would happen, we would keep on using numbers in the same way, but they would look a bit different. For example: 11011010. How many pages of a book does the number 11011010 include? In order to learn that, follow the same logic like in the previous example, but in inverse order. Have in mind that all this is about mathematics with only two digits- 0 and 1, i.e. base-2 number system (binary number system).

Clearly, it is the same number represented in two different ways. The only difference is in the number of digits necessary for writing some number. One digit (2) is used to write the number 2 in decimal system, whereas two digits (1 and 0) are used to write that number in binary system. Do you now agree with the first sentence in this text? Welcome to the world of binary arithmetic! Do you have any idea where it is used?

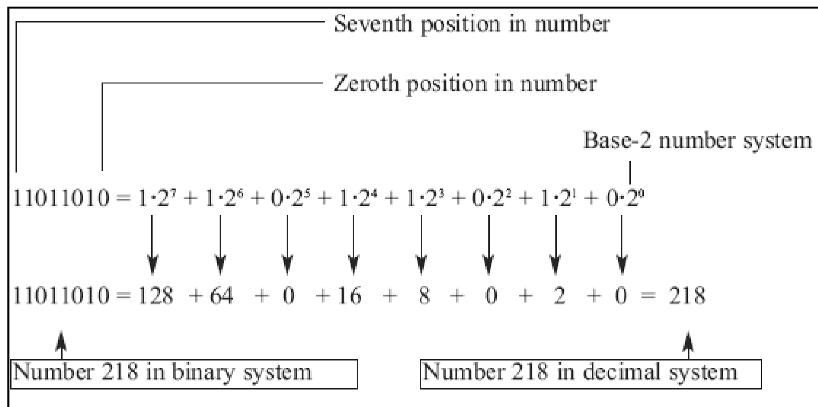


Fig. 5 Showing Representation of Binary Numbers

Excepting strictly controlled laboratory conditions, the most complicated electronic circuits cannot with accuracy determine difference between two sizes (two voltage values, for example) if they are too small (lower than several volts). The reasons for that are electrical noises and something quite uncertainly called “realistic working environment” (unpredictable changes of power supply voltage, temperature changes, tolerance to values of built in components etc.). Imagine a computer which would operate upon decimal numbers by recognizing 10 digits in the following way: 0=0V, 1=5V, 2=10V, 3=15V, 4=20V... 9=45V !? Did anybody say batteries? Far simpler solution is the use of binary logic where 0 indicates that there is no voltage and 1 indicates that there is voltage. Simply, it is easier to write 0 or 1 instead of “there is no voltage” or “there is voltage”. It is so called logic zero (0) and logic one (1) which electronics perfectly cope with and easily performs all those endlessly complex mathematical operations. It is apparently electronics which in reality applies mathematics in which all numbers are represented by two digits only and in which it is only important to know whether there is voltage or not. Of course, we are talking about digital electronics.

Hexadecimal number system

At the very beginning of the computer development it was realized that people had many difficulties in handling binary numbers. Because of that, a new number system which facilitated work has been established. This time, it is about number system using 16 different digits. The first ten digits are the same as digits we are used to (0, 1, 2, 3,... 9) but there are six digits more. In order to keep from making up new symbols, the six letters of alphabet A, B, C, D, E and F are used. In consequence of that, a hexadecimal number system consisting of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F has been established. What is the purpose of this seemingly bizarre combination? Just look how perfectly everything fits the story about binary numbers.

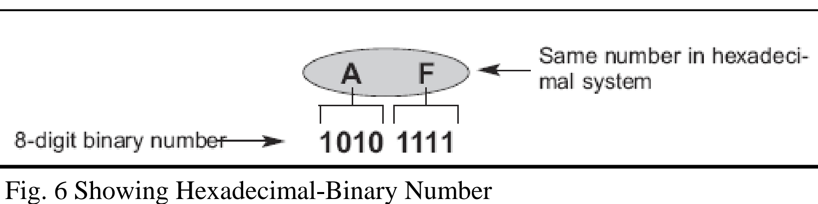


Fig. 6 Showing Hexadecimal-Binary Number

The largest number that can be represented by 4 binary digits is the number 1111. It corresponds to the number 15 in decimal system. That number is in hexadecimal system represented by only one digit F. It is the largest one-digit number in hexadecimal system. Do you see how skillfully it is used? The largest number written with eight binary digits is at the same time the largest two-digit hexadecimal number. Have in mind that the computer uses 8-digit binary numbers. Accidentally?

BCD code

BCD code is actually a binary code for decimal numbers only. It is used to enable electronic circuits to communicate in decimal number system with peripherals and in binary system within “their own world”. It consists of 4-digit binary numbers which represent the first ten digits (0, 1, 2, 3 ... 8, 9). Simply, even though four digits can give total of 16 possible combinations, only first ten are used.

Number system conversion

Binary number system is the most commonly used and decimal system is the most understandable while hexadecimal system is somewhere between them. Therefore, it is very important to learn how to convert numbers from one number system to another, i.e. how to turn series of zeros and units into values understandable for us.

Binary to decimal number conversion

Digits in a binary number have different values depending on their position in that number. Additionally, each position can contain either 1 or 0 and its value may be easily determined by its position from the right. To make the conversion of a binary number to decimal it is necessary to multiply values with the corresponding digits (0 or1) and add all the results. The magic of binary to decimal number conversion works...You doubt? Look at the example:

110 = 1*2^2 + 1*2^1 + 0*2^0 = 6

It should be further noticed that for decimal numbers from 0 to 3 it is enough to have two binary digits. For greater values, new binary digits must be added. Thus, for numbers from 0 to 7 it is enough to have three digits, for numbers from 0 to 15- four digits etc. Simply speaking, the largest binary number consisting of n digits is obtained when the base 2 is raised by n. The result should be afterwards subtracted by 1. For example, if n=4:

2^4 - 1 = 16 - 1 = 15

Accordingly, using 4 binary digits it is possible to represent decimal numbers from 0 to 15, including these two digits, which amounts to 16 different values in total.

A37E (number in hexadecimal system)

$14 \cdot 16^0 = 14 \cdot 1 = 14$

$7 \cdot 16^1 = 7 \cdot 16 = 112$

$3 \cdot 16^2 = 3 \cdot 256 = 768$

$10 \cdot 16^3 = 10 \cdot 4096 = 40960$

41854 (same number in decimal system)

Fig. 7 Showing Hexadecimal to decimal conversion

Hexadecimal to decimal number conversion

In order to make conversion of a hexadecimal number to decimal, each hexadecimal digit should be multiplied with the number 16 raised by it’s position value. For example:

Hexadecimal to binary number conversion

It is not necessary to perform any calculation in order to convert hexadecimal number to binary number system. Hexadecimal digits are simply replaced by the appropriate four binary digits. Since the maximal hexadecimal digit is equivalent to decimal number 15, it is needed to use four binary digits to represent one hexadecimal digit. For example:

E4 = 1110 0100

(1110 = E, 0100 = 4)

Marking numbers

Hexadecimal number system is along with binary and decimal number systems considered to be the most important for us. It is easy to make conversion of any hexadecimal number to binary and it is also easy to remember it. However, these conversions as well as common use of different number systems may cause confusion. For example, what does the statement “It is necessary to count up 110 products on assembly line” actually mean? Depending on whether it is about binary, decimal or hexadecimal system, the result could be 6, 110 or 272 products, respectively! Accordingly, in order to avoid misunderstandings, different prefixes and suffixes are directly added to the numbers. The prefix $ or 0x as well as the suffix h marks the numbers in hexadecimal system. For example, hexadecimal number 10AF may look as follows $10AF, 0x10AF or 10AFh. Similarly, binary numbers usually get the suffix % or 0b, whereas decimal numbers get

the suffix D. Commonly if no suffix is used the number is assumed to be decimal.

Bit

Theory says a bit is the basic unit of information...Let us neglect such a dry explanation for a moment and take a look at what it is in practice. The answer is- nothing special- a bit is a binary digit. Similar to decimal number system in which digits in a number do not have the same value (for example digits in the number 444 are the same, but have different values), the “significance” of some bit depends on the position it has in binary number. Therefore, there is no point to talk about units, tens etc. Instead, here it is about zero bit (rightmost bit), first bit (second from the right) etc. In addition, since the binary system uses two digits only (0 and 1), the value of one bit can be 0 or 1.

Do not let you be confused if you find some bit has value 4, 16 or 64. It means that bit’s values are represented in decimal system. Simply, we have got so much accustomed to the usage of decimal numbers that these expressions became common. It would be correct to say for example, “the value of the sixth bit in binary number is equivalent to decimal number 64”. But we all are just humans and a habit does its own...Besides, how would it sound “number: one-onezero- one-zero...”

Byte

A byte or a program word consists of eight bits placed next to each other. If a bit is a digit, it is logical that bytes represent numbers. All mathematical operations can be performed upon them, like upon common decimal numbers. As It is case with digits of any other number, byte digits do not have the same significance. The largest value has the left-most bit called most significant bit (MSB). The right-most bit has the least value and is therefore called least significant bit (LSB). Since eight zeros and units of one byte can be combined in 256 different ways, the largest decimal number which can be represented by one byte is

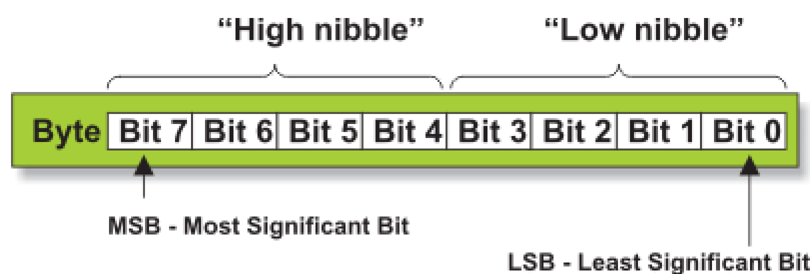


Fig. 8 High and Low Nibbles of a Byte

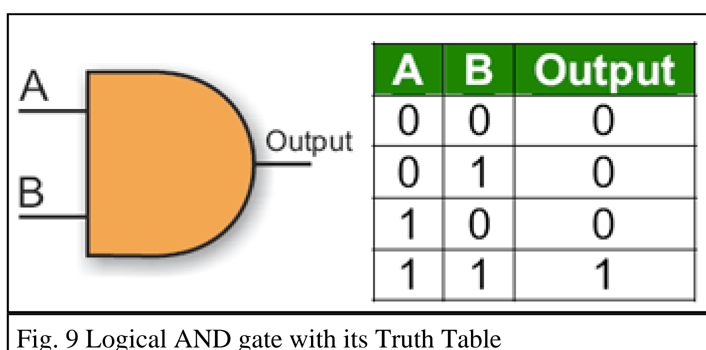
255 (one combination represents zero).

Concerning terminology used in computer science, a concept of nibble should be clarified. Somewhere and somehow, this term referred to as half a byte came up. Depending on which half of the byte we are talking about (left or right), there are “high” and “low” nibbles.

Logic circuits

Have you ever wondered what electronics within some digital integrated circuit, microcontroller or processor look like? What do the circuits performing complicated mathematical operations and making decisions look like? Do you know that their seemingly complicated schematics comprise only a few different elements called “logic circuits” or “logic gates”?

The operation of these elements is based on the principles established by British mathematician George Boole in the middle of the 19th century- meaning before the first bulb was invented! In brief, the main idea was to express logical forms through algebraic functions. Such thinking was soon transformed into a practical product which far later evaluated in what today is known as AND, OR and NOT logic circuits. The principle of their operation is known as



A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 9 Logical AND gate with its Truth Table

Boolean algebra. As some program instructions used by the microcontroller perform the same way as logic gates but in form of commands, the principle of their operation will be discussed here.

AND gate

A logic gate "AND" has two or more inputs and one output. Let us presume that the gate used in this case has only two inputs. A logic one (1) will appear on its output only in case both inputs (A AND B) are driven to logic one (1). That's all! Schematic symbol of AND gate is shown in the figure on the right.

Additionally, the table shows mutual dependence between inputs and output.

In case the gate has more than two inputs, the principle of operation is the same: a logic one (1) will appear on its output only in case all inputs are driven to logic one (1). Any other combination of input voltages will result in logic zero (0) on its output.

When used in a program, logic AND operation is performed by the program instruction, which will be discussed later. For the time being, it is enough to remember that logic AND in a program refers to the corresponding bits of two registers.

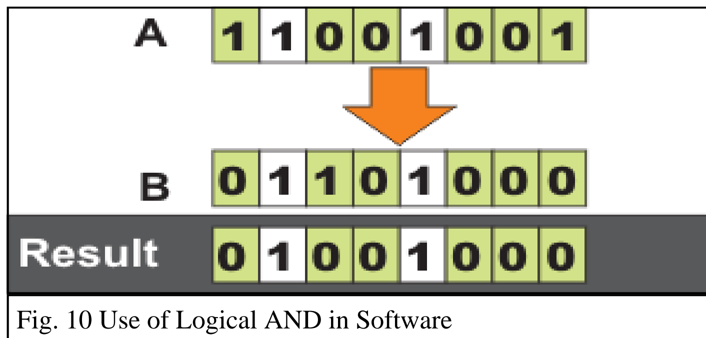
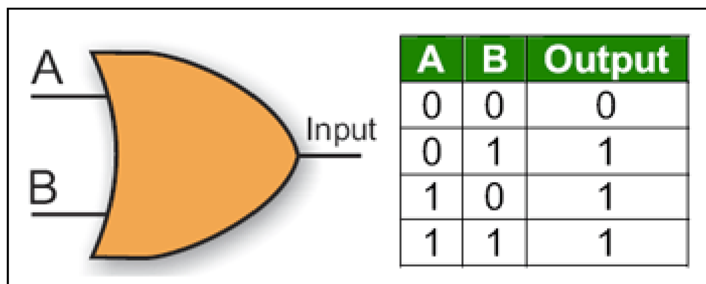


Fig. 10 Use of Logical AND in Software



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 11 The OR Gate with Truth Table

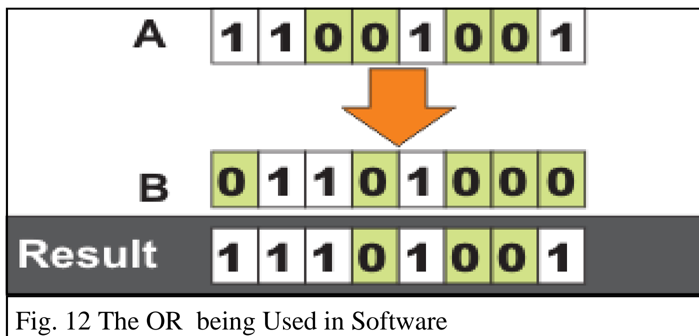
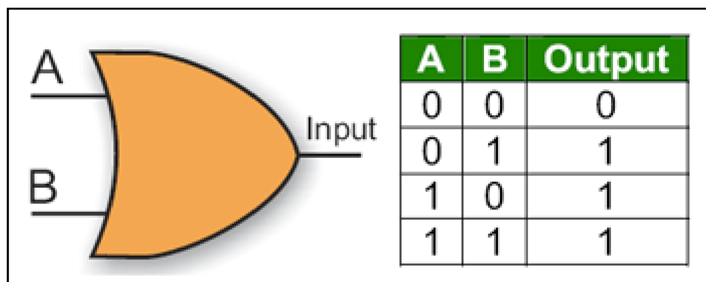


Fig. 12 The OR being Used in Software

OR gate

Similar to the previous case, OR gate also has two or more inputs and one output. The gate with only two inputs will be considered in this case as well. A logic one (1) will appear on its output in case either one or another output (A OR B) is driven to logic one (1). In case the OR gate has more than two inputs, the following applies: a logic one (1) appears on its output in case at least one input is driven to logic one (1). In case all inputs are driven to logic zero (0), the output will be driven to logic zero (0).

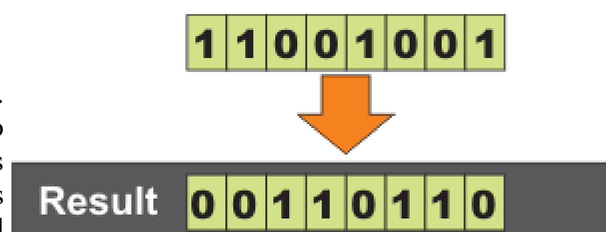


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 13 The NOT Gate

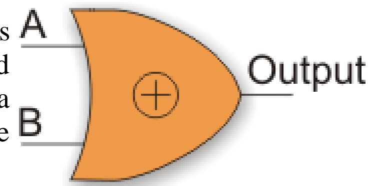
Not gate

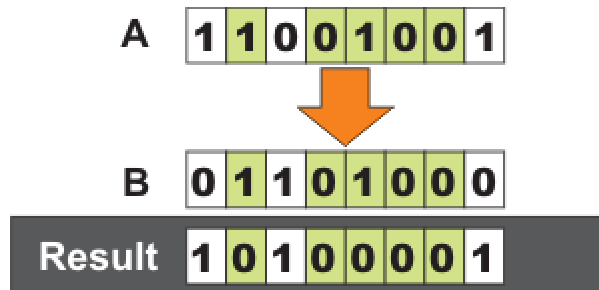
This logic gate has only one input and only one output. It operates in an extremely simple way. When logic zero (0) appears on its input, a logic one (1) appears on its output and vice versa. This means that this gate inverts signal by itself and because of that it is sometimes called



inverter.

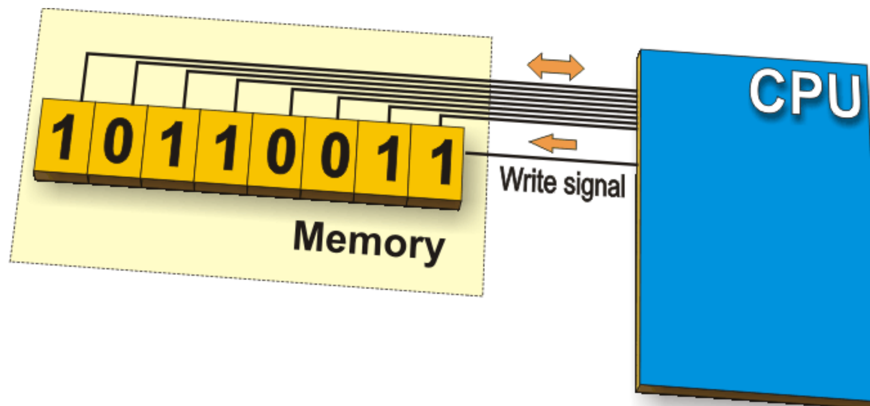
In a program, logic NOT operation is performed on one byte bits. The result is a byte with inverted bits. If byte bits are considered to be a number, inverted value is actually a complement of that number, i.e. The complement of a number is what is needed to add to it to make it reach the maximal 8 bit value (255).





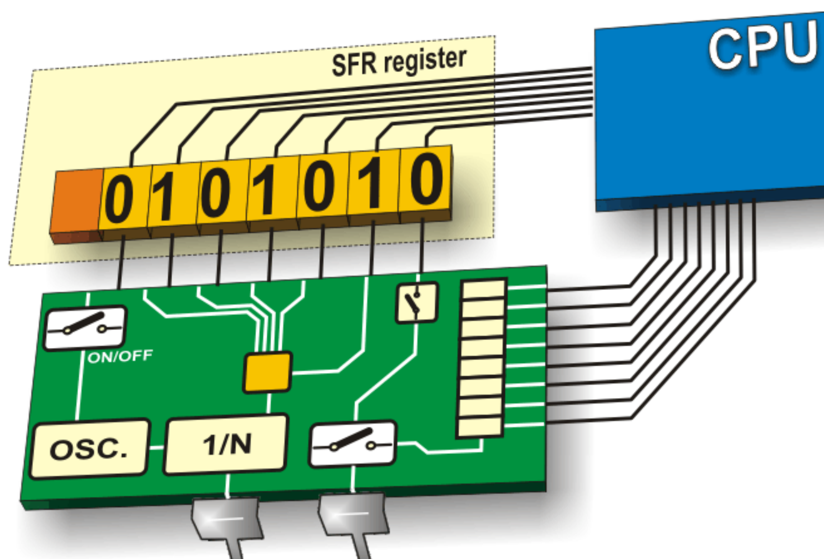
EXCLUSIVE OR gate

This gate is a bit complicated comparing to other gates. It represents combination of all previously described gates. It is not simple to define mutual dependence of input and output, but we will anyway try to



do it. A logic one (1) appears on its output only in case the inputs have different logic states.

In a program, this operation is commonly used to compare two bytes. Subtraction may be used for the same purpose (if the result is 0, bytes are equal). The advantage of this logic operation is that there is no danger



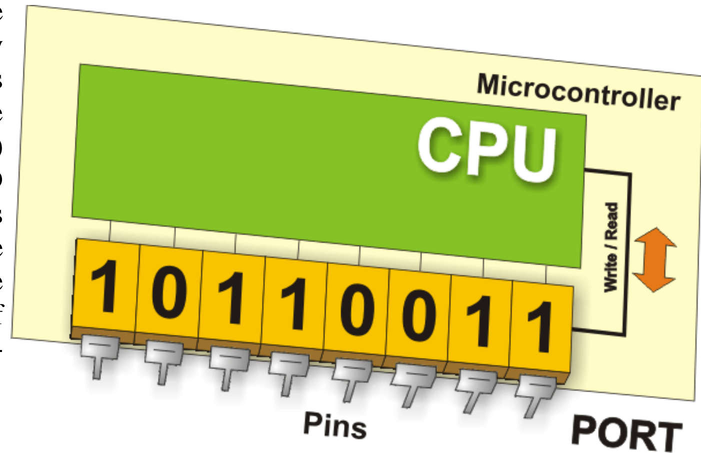
to subtract larger number from smaller one.

Register

A register or a memory cell is an electronic circuit which can memorize the state of one byte. In other words, what is a byte theoretically, it is a register practically.

Special Function Registers (SFR registers)

In addition to the registers which do not have any special and predetermined function, every microcontroller has also a number of registers whose function is predetermined by the manufacturer. Their bits are connected (literally) to internal circuits such as timers, A/D converter, oscillators and others, which means that they are directly in command of the operation of the microcontroller. If you imagine that as eight switches which are in command of some smaller circuit within the microcontroller- you are right! SFRs do exactly that!



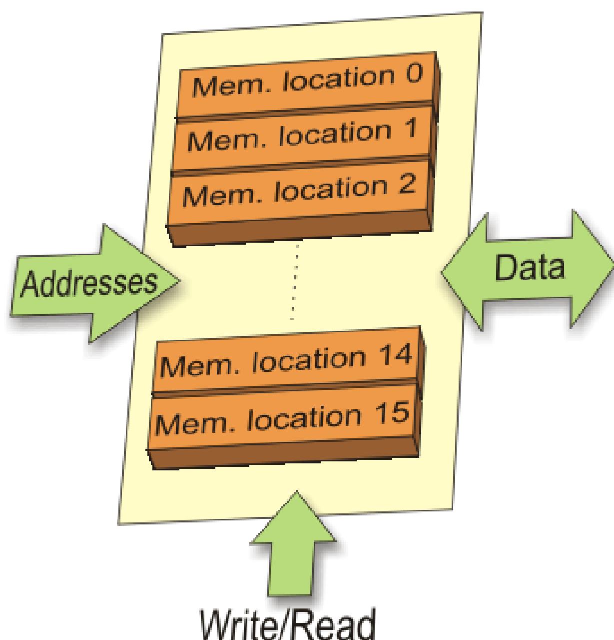
Input / Output ports

In order that the microcontroller is of any use, it has to be connected to additional electronics, i.e. peripherals. For that reason, each microcontroller has one or more registers (called “port” in this case) connected to the microcontroller pins. Why input/output? Because you can change the pin’s function as you wish. For example, suppose you want your device to turn on and off three signal LEDs and simultaneously monitor logic state of five sensors or push buttons. In accordance with that, some of ports should be configured so that there are three outputs (connected to LEDs) and five inputs (connected to sensors). It is simply performed by software, which means that pin’s function can be changed during operation.

One of more important feature of I/O pins is maximal current they can give/get. For the most microcontrollers, current obtained from one pin is sufficient to activate a LED or other similar low-current consumer (10-20 mA). If the microcontroller has many I/O pins, then maximal current of one pin is lower. Simply, you cannot expect all pins to give maximal current if there are more than 80 of them on one microcontroller.

Another important pin feature is to (or not to) have pull-up resistors. These resistors connect pin to positive power supply voltage and their effect is visible when the pin is configured as input connected to mechanical switch or push button. The later versions of the microcontrollers have pull-up resistors connected to and disconnected from the pins by software.

Usually, each I/O port is under control of another SFR, which means that each bit of that register determines state of the corresponding microcontroller pin. For example, by writing logic one (1) to one bit of that control register SFR, the appropriate port pin is automatically configured as input. It means that voltage brought to that pin can be read as logic 0 or 1. Otherwise, by writing zero to the SFR, the appropriate port pin is configured as output. Its voltage (0V or 5V) corresponds to the state of the appropriate bit of the port register.



Memory unit

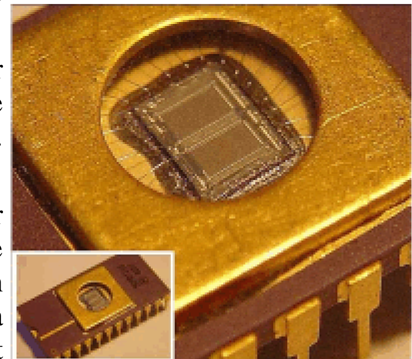
Memory is part of the microcontroller used for data storage. The easiest way to explain it is to compare it with a big closet with many drawers. Suppose, the drawers are clearly marked so that it is easy to access any of them. It is enough to know the drawer’s mark

to find out its contents.

Memory components are exactly like that. Each memory address corresponds to one memory location. The content of any location becomes known by its addressing. Memory consists of all memory locations and addressing is nothing but selecting one of them. This means that, on one hand it is necessary to select the desired memory location, on the other hand it is necessary to wait for the contents of that location. In addition to read, memory also has to allow writing to these locations. There are several types of memory within the microcontroller:

ROM memory (Read Only Memory)

ROM memory is used to permanently save program being executed. Clearly, the size of a program that can be written depends on the size of this memory. Today's microcontrollers commonly use 16-bit addressing, which means that they are able to address up to 64 Kb memory, i.e. 65535 locations. For the sake of illustration, if you are the beginner, your program will rarely exceed limit of several hundreds instructions. There are several types of ROM.



Masked ROM. Microcontrollers containing this ROM are reserved for the great manufacturers. Program is loaded into the chip by the manufacturer. In case of large scale manufacture, the price is very low. Forget it...

OTP ROM (One Time Programmable ROM). If the microcontroller contains this memory, you can download a program into the chip, but the process of program downloading is "one-way ticket", meaning that it can be done only once. If you after downloading detect some error in a program, the only thing you can do is to correct it and download that program to another chip.

UV EPROM (UV Erasable Programmable ROM) Both manufacturing process and characteristics of this memory are completely identical to OTP ROM. However, the package of this microcontroller has recognizable "window" on the upper side. It enables surface of the silicon chip to be lit by an UV lamp, which has for the result that complete program is cleared and a new program download is enabled. Installation of this window is very complicated, which normally affects the price. From our point of view, unfortunately- negative…

Flash memory. This type of memory was invented in the 80s in laboratories of INTEL company and were represented as successor of UV EPROM. Since the contents of this memory can be written and cleared practically unlimited number of times, the microcontrollers with Flash ROM are ideal for learning, experimentation and small-scale manufacture. Because of its popularity, the most microcontrollers are manufactured in flash version today. So, if you are going to buy a microcontroller, the right one is definitely Flash!

RAM memory (Random Access Memory).

Once the power supply is off the contents of RAM is cleared. It is used for temporary storing data and intermediate results created and used during the operation of the microcontroller. For example, if the program performs addition (of whatever), it is necessary to have a register representing what in everyday life is called "sum". For that purpose, one of the registers in RAM is called "sum" and used for storing results of addition.

EEPROM memory (Electrically Erasable Programmable ROM)

The contents of this memory may be changed during operation (similar to RAM), but remains permanently saved even upon the power supply goes off (similar to ROM). Accordingly, EEPROM is often used to store values, created during operation, which must be permanently saved. For example, if you design an electronic lock or an alarm, it would be great to enable the user to create and enter a password on his/her own. Of course, a new password must be saved upon power supply goes off. In such and similar cases, the ideal solution is the microcontroller with embedded EEPROM.

Interrupt

Most programs use somehow interrupts in regular program execution. What does it actually mean? The purpose of the microcontroller is mainly to react on changes in its surrounding. In other words, when some

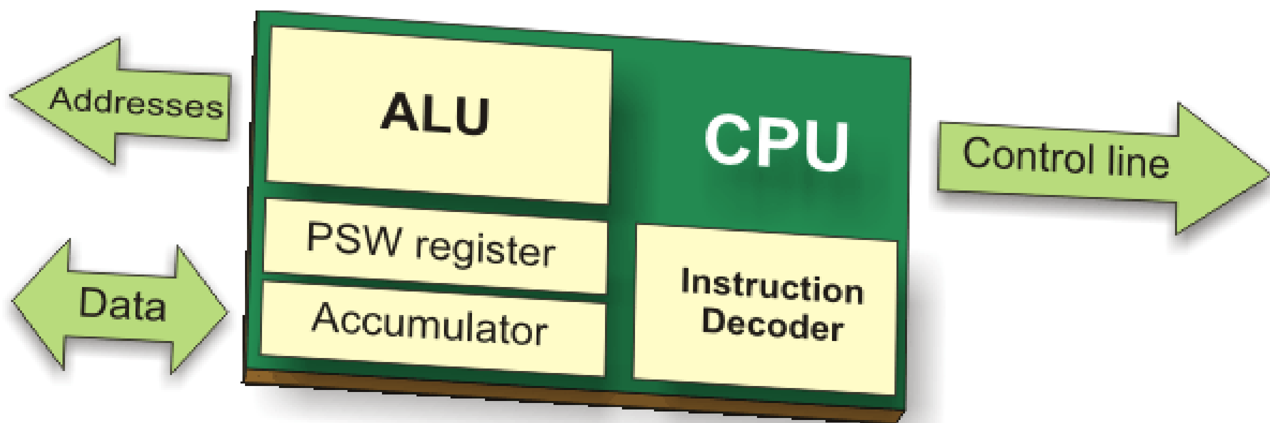
event takes place, the microcontroller does something... For example, when you push a button on remote controller, the microcontroller will register it and respond to the order by changing a channel, turn the volume up or down etc. The bottom line is that the microcontroller spends the most of its time in endlessly checking a few buttons- for hours, days... It's not practical, is it?

Because of and similar situations, the microcontroller has learned during its evolution a trick. Instead of checking each pin or bit constantly, the microcontroller has left the "wait issue" to the "specialist" which will react only in case something worth attention happens.

Signal which inform the central processor about such event is called an INTERRUPT.

Central Processor Unit - CPU

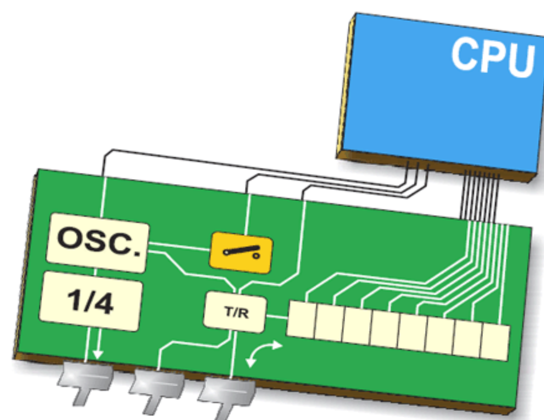
As its name indicates, this is a unit which monitors and controls all processes inside the microcontroller. It consists of several smaller units, of which the most important are:



- **Instruction Decoder** is a part of electronics which recognizes program instructions and runs other circuits on the basis of that. The "instruction set" which is different for each microcontroller family expresses the abilities of this circuit.
- **Arithmetical Logical Unit (ALU)** performs all mathematical and logical operations upon data.
- **Accumulator** is a SFR closely related to the operation of ALU. It is a kind of working desk used for storing all data upon which some operation should be performed (addition, shift/move etc.). It also stores results ready for use in further processing. One of SFRs, called Status Register (PSW), is closely related to the accumulator. It shows at any moment the "status" of a number stored in the accumulator (number is greater or less than zero etc.).

Bus

Physically, the bus consists of 8, 16 or more wires. There are two types of buses: address and data bus. The first one consists of as many lines as necessary for memory addressing. It is used to transmit address from CPU to memory. The later one is as wide as data, in our case it is 8 bits or wires wide. It is used to connect all circuits inside the microcontroller.



Serial communication

Connection between the microcontroller and peripherals via input/output ports is the ideal solution for shorter distances, up to several meters. However, in other cases - when it is necessary to establish communication between two devices on longer distances or when for some other reason it is not possible to use parallel connection - such a simple solution is out of question. In those and similar situations, serial communication is the solution imposing itself.

Today, most microcontrollers have built in several different systems for serial communication as a standard equipment. Which of these systems will be used in the very case depends on many factors of which the most important are:

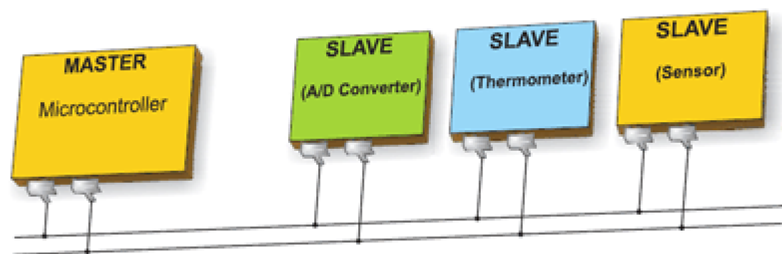
- How many devices the microcontroller has to exchange data with?
- How fast the data exchange has to be?
- What is the distance between devices?
- Is it necessary to send and receive data simultaneously?

One of the most important thing concerning serial communication is the *Protocol* which should be strictly observed. It is a set of rules which must be applied in order the devices can correctly interpret data they mutually exchange. Fortunately, the microcontrollers automatically take care of that, so the work of the programmer/user is reduced to simple write (data to be sent) and read (received data).

Baud Rate

The term *Baud rate* is commonly used to denote the number of bits transferred per second [bps].

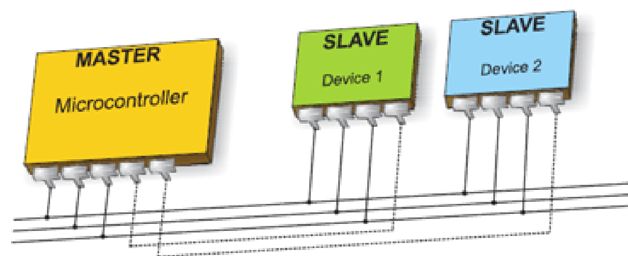
It should be noted that it refers to bits, not bytes! It is usually required by the protocol that each byte is transferred along with several control bits. It means that one byte in serial data stream may consist of 11 bits. For example, if the baud rate is 300 bps then maximum 37 and minimum 27 bytes may be transferred per second, which depends on type of connection and protocol in use.



The most commonly used serial communication systems are:

I2C Protocol (Two wire System)

I2C (Inter Integrated Circuit) is a system used when the distance between the microcontrollers is short and specialized integrated circuits of a new generation (receiver and transmitter are usually on the same printed circuit board). Connection is established via two conductors- one is used for data transfer whereas another is used for synchronization (clock signal). As seen in figure, in such connection, one device is always master. It performs addressing of one slave chip (subordinated) before communication starts. In this way one microcontroller can communicate with 112 different devices. Baud rate is usually 100 Kb.sec (standard mode) or 10 Kb/sec (slow baud rate mode). Systems with the baud rate of 3.4 Mb/sec have recently appeared. The distance between devices which communicate via an inter-integrated circuit bus is limited to several meters.



SPI (Three Wire Serial - Parallel Interface)

SPI (Serial Peripheral Interface Bus) is a system for serial communication which uses four conductors (usually three)- one for data receiving, one for data sending, one for synchronization and one (alternatively) for selecting device to communicate with. It is full duplex connection, which means that data are sent and received simultaneously. Maximal baud rate is higher than in I2C connection.

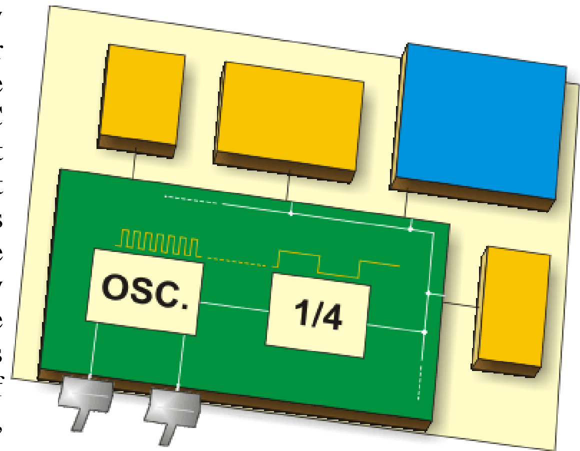
UART (Universal Asynchronous Receiver/Transmitter)

As seen from the name itself, this connection is asynchronous, which means that a special line for clock signal transmission is not used. In some situations this feature is crucial (for example, radio connection or infrared waves remote control). Since only one communication line is used, both receiver and transmitter operate at the same predefined rate in order to maintain necessary synchronization. This is a very simple

way of transferring data since it basically represents conversion of 8-bit data from parallel to serial format. Baud rate is not high and amounts up to 1 Mbit/sec.

Oscillator

Evenly spaced pulses coming from the oscillator enable harmonic and synchronous operation of all circuits of the microcontroller. The oscillator module is usually configured to use quartz crystal or ceramic resonator for frequency stabilization. Furthermore, it can also operate without elements for frequency stabilization (like RC oscillator). It is important to say that instructions are not executed at the rate imposed by the oscillator itself, but several times slower. It happens because each instruction is executed in several steps. In some microcontrollers, the same number of cycles is needed to execute any instruction, while in others, the execution time is not the same for all instructions. Accordingly, if the system uses quartz crystal with frequency of 20 Mhz, execution time of an instruction is not 50nS, but 200, 400 or 800 nS, depending on the type of MCU!



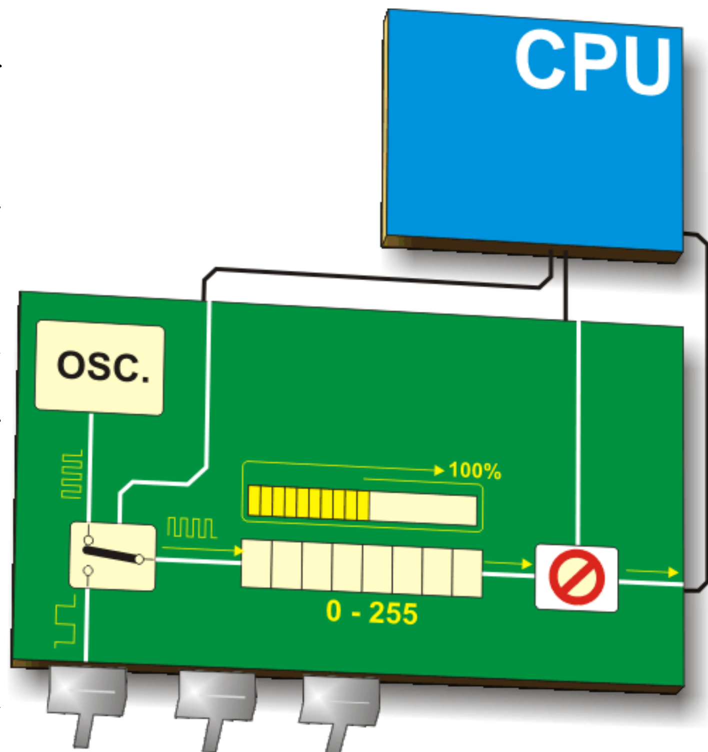
PIC divides the external oscillator frequency by 4 fosc/4, to execute. Thus if using an external oscillator of 4MHz, internally it is using 1MHz.

Power supply circuit

There are two things worth attention concerning the microcontroller power supply circuit:

Brown out is a potentially dangerous state which occurs at the moment the microcontroller is being turned off or in situations when power supply voltage drops to the limit due to powerful electric noises. As the microcontroller consists of several circuits which have different operating voltage levels, this state can cause its out-of-control performance. In order to prevent it, the microcontroller usually has built-in circuit for brown out reset. This circuit immediately resets the whole electronics when the voltage level drops below the limit.

Reset pin is usually marked as MCLR (Master Clear Reset) and serves for external reset of the microcontroller by applying logic zero (0) or one (1), depending on type of the microcontroller. In case the brown out circuit is not built in, a simple external circuit for brown out reset can be connected to this pin.



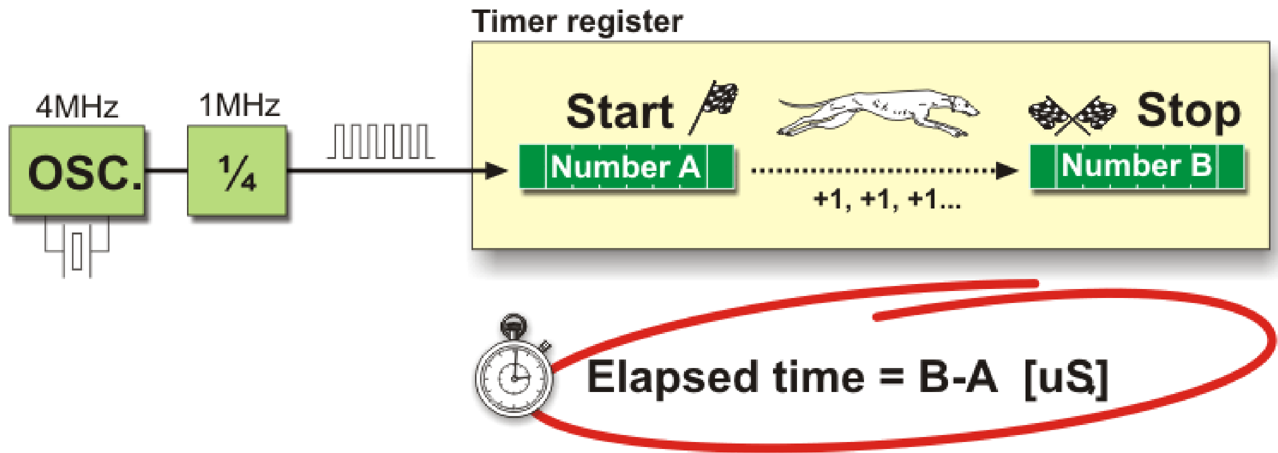
Timers/Counters

The microcontroller oscillator uses quartz crystal for its operation. Even though it is not the simplest solution, there are many reasons to use it. Namely, since the frequency of such oscillator is precisely defined and very stable, the pulses it generates are always of the same width, which makes them ideal for time measurement. Such oscillators are used in quartz watches. If it is necessary to measure time passed between two events, it is just enough to count pulses coming from this oscillator. That is exactly what the timer does.

Most programs use somehow these miniature electronic “stopwatches”. These are commonly 8- or 16-bit SFRs and their content is automatically incremented by each coming pulse. Once a register is completely

loaded - an interrupt is generated!

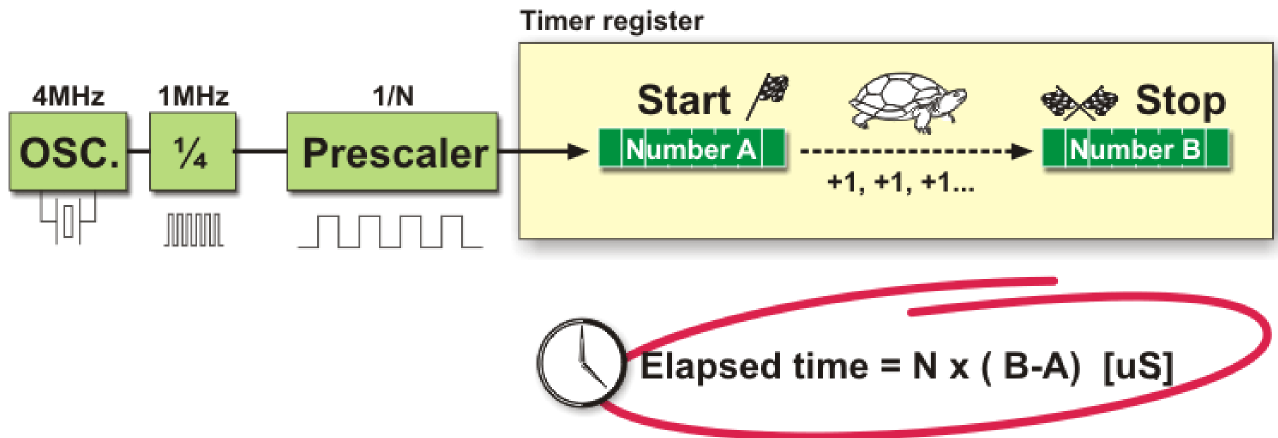
If the timer registers use internal quartz oscillator for their operation then it is possible to measure time between two events (if the register value is T1 at the moment measurement has started, and T2 at the moment it has finished, then the elapsed time is equal to the result of subtraction T2-T1). If the registers use pulses coming from external source then such a timer is turned into a counter.



This is only a simple explanation of the operation itself.

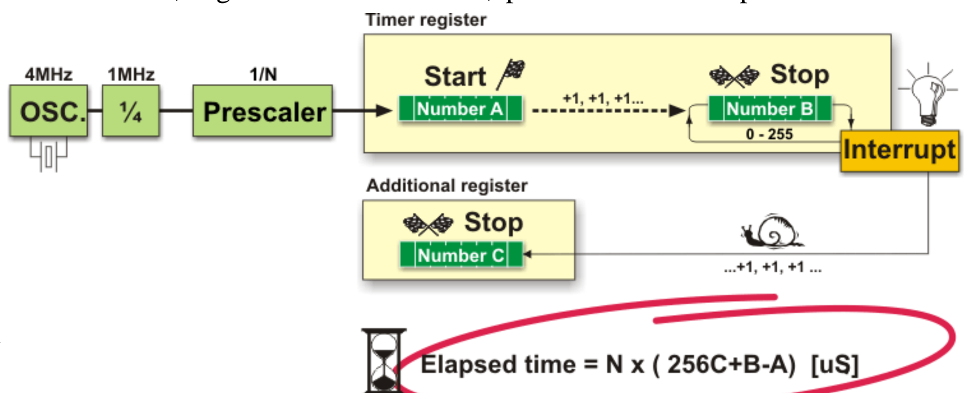
How does a timer operate?

In practice, everything works as follows: pulses coming from quartz oscillator are once per each machine cycle directly or via pre-scaler brought to the circuit which increments number in the timer register. If one instruction (one machine cycle) lasts for four quartz oscillator periods then, by embedding quartz with the



frequency of 4MHz, this number will be changed a million times per second (each microsecond).

It is easy to measure short time intervals (up to 256 microseconds) in a way described above because it is the largest number that one register can contain. This obvious disadvantage may be easily overcome in several ways by using slower oscillator, registers with more bits, prescaler or interrupts. The first two solutions have some weaknesses so it is more recommended to use prescaler and/or interrupt.



Using prescaler in timer operating

A prescaler is an electronic device used to reduce a frequency by a pre-determined factor.

Meaning that in order to generate one pulse on its output, it is necessary to bring 1, 2 , 4 or more pulses to its input. One such circuit is built in the microcontroller and its division rate can be changed from within the program. It is used when it is necessary to measure longer periods of time.

One prescaler is usually shared by timer and watch-dog timer, which means that it cannot be used by both of them simultaneously.

Using interrupt in timer operating

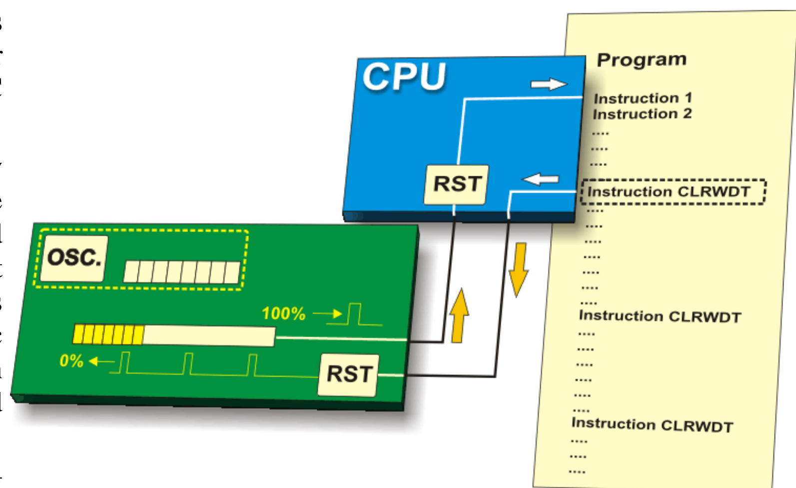
If the timer register consists of 8 bits, the largest number that can be written to it is 255 (for 16-bit registers it is the number 65535). If this number is exceeded, the timer will be automatically reset and counting will start from zero. This condition is called overflow. If enabled from within the program, such overflow can cause interrupt, which gives completely new possibilities. For example, the state of registers used for counting seconds, minutes or days can be changed in an interrupt routine. The whole this process (except interrupt routine) is automatically performed "in the background", which enables main circuits of the microcontroller to perform other operations.

Counters

If a timer is supplied with pulses over the microcontroller input pin then it turns into a counter. Clearly, It is about the same electronic circuit. The only difference is that in this case pulses to be counted come through the ports and their duration (width) is mostly not defined. That is why they cannot be used for time measurement, but can be used to measure anything else: products on an assembly line, number of axis rotation, passengers etc. (depending on sensor in use).

Watchdog Timer

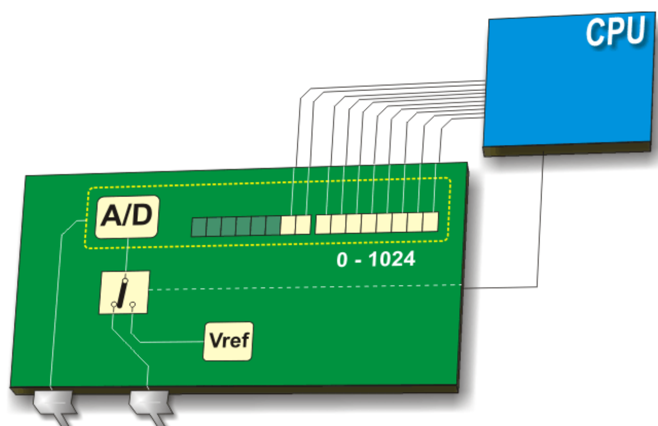
As name itself indicates a lot about its purpose. Watchdog Timer is a timer connected to a completely separate RC oscillator within the microcontroller.



If the watchdog timer is enabled, every time it counts up to end, the microcontroller reset occurs and program execution starts from the first instruction. The point is to prevent this from happening by using a specific command. The whole idea is based on the fact that every program is executed in several longer or shorter loops.

If instructions which reset the watchdog timer are set on the appropriate program locations, besides commands being regularly executed, then the operation of watchdog timer will not affect program execution. If for any reason (usually electrical noises in industry), the program counter "gets stuck" on some memory location from which there is no return, the watchdog will not be cleared and the register's value being constantly incremented will reach the maximum et voila! Reset occurs!

A/D Converter



External signals are usually fundamentally different from those the microcontroller understands (zero and one), so that they have to be converted in order the microcontroller can understand them. An analog-to digital converter is an electronic circuit which converts continuous signals to discrete digital numbers. This module is therefore used to convert some analog value into binary number and forwards it to the CPU for further processing. In other words, this module is used for input pin voltage measurement (analog

value). The result of measurement is a number (digital value) used and processed later in the program.

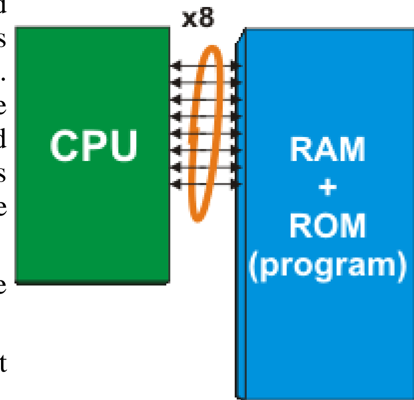
Internal Architecture

All upgraded microcontrollers use one of two basic design models called Harvard and von-Neumann architecture. What is it about?

Briefly, it is about two different ways of data exchange between CPU and memory.

Von-Neumann architecture

Microcontrollers using this architecture has only one memory block and one 8-bit data bus. As all data are exchanged by using these 8 lines, this bus is overloaded and communication itself is very slow and inefficient. The CPU can either read an instruction or read/write data from/to the memory. Both cannot occur at the same time since the instructions and data use the same bus system. For example, if some program line says that RAM memory register called "SUM" should be incremented by one (instruction: incf SUM), the microcontroller will do the following:

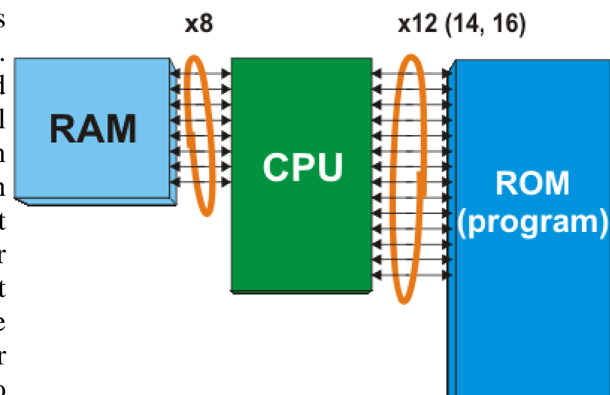


1. Read the part of the program instruction specifying WHAT should be done (in this very case it is the "incf" instruction for increment).
2. Read further the same instruction specifying upon WHICH data it should be performed (in this very case it is the "SUM" register).
3. After being incremented, the contents of this register should be written to the register from which it was read ("SUM" register address).

The same data bus is used for all these intermediate operations.

Harvard Architecture

Microcontrollers using this architecture have two different data buses. One is 8-bit wide and connects CPU to RAM memory. Another one consists of several lines (12, 14 or 16) and connects CPU to ROM memory. Accordingly, the CPU can read an instruction and perform a data memory access at the same time. Since all RAM memory registers are 8- bit wide, all data within the microcontroller are exchanged in the same such format. Additionally, during program writing, only 8-bit data are considered. In other words, all you can ever change from within the program and all you can affect will be 8- bit wide. A program written for some of these microcontrollers will be stored in the microcontroller internal ROM memory upon having being compiled into machine language. However, these memory locations do not have 8, but 12, 14 or 16 bits. The rest of bits- 4, 6 or 8- represents the instruction itself specifying to CPU what to do with an 8-bit data.



The advantages of such design are the following:

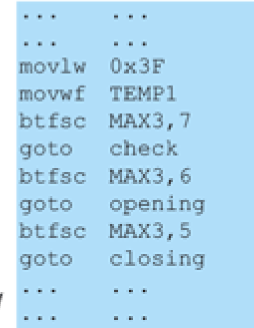
- All data in a program are one byte (8 bit) wide. As data bus used for program reading has several lines (12, 14 or 16), both instruction and data can be read simultaneously by using these spare bits (it is familiar at once WHAT and upon WHICH). Because of that, all instructions are executed in only one instruction cycle. The only exception is jump instructions which are executed in two cycles.
- Owing to the fact that a program (ROM memory) and temporary data (RAM memory) are separate, the CPU can execute two instructions simultaneously. Simply, while RAM memory read or write is in progress (end of one instruction), the next program instruction is being read via another bus.

When using microcontrollers with von-Neumann architecture one never knows how much memory is to be occupied by some program. In average, each program instruction occupies two memory locations (one

contains information on WHAT should be done, whereas another contains information upon WHICH data it should be done). However, it is not a rule, but the most common case. In microcontrollers with Harvard architecture, program bus is wider than one byte, which allows each program word to consist of instruction and data. In other words: one program word- one instruction.

Instruction Set

All instructions that can be understood by the microcontroller are known as instruction set. When you write a program in assembly language, you actually “tell a story” by specifying instructions in order they should be executed. The main restriction in this process is a number of available instructions. The manufacturers stick to one of the two following strategies:



RISC (Reduced Instruction Set Computer)

In this case, the idea is that the microcontroller recognizes and executes only basic operations (addition, subtraction, copying etc.). All other more complicated operations are performed by combining these (for example, multiplication is performed by performing successive addition). The constrains are obvious (as if you try, by using only a few words, to explain to someone how to reach the airport in some other city). However, there are also some great advantages. First of all, this language is easy to learn. Besides, the microcontroller is very fast so that it is not possible to see all the arithmetic “acrobatics” it performs. The user can only see the final result of all those operations. At last, it is not so difficult to explain where the airport is if you use the right words. For example: left, right, kilometer etc.

CISC (Complex Instruction Set Computer)

You already catch it- CISC is the opposite of RISC! Microcontrollers designed to recognize more than 200 different instructions can do really much and are very fast. However, one should know how to take all that such a rich language offers, which is not easy at all…

HOW TO MAKE THE RIGHT CHOICE?

Ok, you are the beginner and you have made decision to let yourself go on an adventure of working with the microcontrollers. Congratulations on the choice! However, it is not so easy to choose the right microcontroller as it looks like at first sight. The problem is not a small range of devices, but the opposite!

Before you start designing some device based on the microcontroller, think of the following: how many input/output lines it is necessary for operation, should it perform some other operations than to turn relay on/off, does it need some specialized module such as serial communication, A/D converter etc. When you create a clear picture of what you need, the selection range is considerably reduced, and it is time to think of price. Is your plan to have several same devices? Several hundreds? A million? Anyway, you catch the point.

If you think of all these things for the very first time then everything seems a bit confusing. For that reason, go step by step. First of all, select the manufacturer, i.e. the family of the microcontrollers you can easily provide. After that, study one particular model. Learn as much as you need, do not go into details. Solve a specific problem and something incredible will happen- you will be able to handle any model belonging to that family.

PIC microcontrollers

PIC microcontrollers designed by Microchip® Technology are likely the right choice for you if you are the beginner. Here is why...

The real name of this microcontroller is PICmicro (Peripheral Interface Controller), but it is better known as PIC. Its first ancestor was designed in 1975 by General Instruments. This chip called PIC1650 was meant for totally different purposes. Not longer than ten years after, by adding EEPROM memory, this circuit was transformed into a real PIC microcontroller. Nowadays, Microchip Technology announces a manufacturing of the 5 billionth sample...

In order you can better understand the reasons for its popularity, we will briefly describe several important things.

Family	ROM [Kbytes]	RAM [bytes]	Pins	Clock Freq. [MHz]	A/D Inputs	Resolu-tion of A/D Con-	Com-par-ators	8/16 – bit Timers	Serial Comm.	PWM Out-puts	Others
Base-Line 8 - bit architecture, 12-bit Instruction Word Length											
PIC10FXXX	0.375 -	16 - 24	6 - 8	4 - 8	0 - 2	8	0 - 1	1 x 8	-	-	-
PIC12FXXX	0.75 - 1.5	25 - 38	8	4 - 8	0 - 3	8	0 - 1	1 x 8	-	-	EEPRO M
PIC16FXXX	0.75 - 3	25 - 134	14 - 44	20	0 - 3	8	0 - 2	1 x 8	-	-	EEPRO
PIC16HVX	1.5	25	18 - 20	20	-	-	-	1 x 8	-	-	Vdd =
Mid-Range 8 - bit architecture, 14-bit Instruction World Length											
PIC12FXXX	1.75 -	64 - 128	8	20	0 - 4	10	1	1 - 2 x 8	-	0 - 1	EEPRO
PIC12HVX XX	1.75	64	8	20	0 - 4	10	1	1 - 2 x 8 1 x 16	-	0 - 1	-
PIC16FXXX	1.75 - 14	64 - 368	14 - 64	20	0 - 13	8 or 10	0 - 2	1 - 2 x 8 1 x 16	USART I2C SPI	0 - 3	-
PIC16HVX XX	1.75 - 3.5	64 - 128	14 - 20	20	0 - 12	10	2	2 x 8 1 x 16	USART I2C SPI	-	-
High-End 8 - bit architecture, 16-bit Instruction Word Length											
PIC18FXXX	4 - 128	256 - 3936	18 - 80	32 - 48	4 - 16	10 or 12	0 - 3	0 - 2 x 8 2 - 3 x 16	USB2.0 CAN2.0 USART	0 - 5	-
PIC18FXXJ XX	8 - 128	1024 - 3936	28 - 100	40 - 48	10 - 16	10	2	0 - 2 x 8 2 - 3 x 16	USB2.0 USART Ethernet	2 - 5	-
PIC18FXXK XX	8 - 64	768 - 3936	28 - 44	64	10 - 13	10	2	1 x 8 3 x 16	USART I2C SPI	2	-

All PIC microcontrollers use harvard architecture, which means that their program memory is connected to CPU via more than 8 lines. Depending on the bus width, there are 12-, 14- and 16-bit microcontrollers. The table above shows the main features of these three categories.

As seen in the table on the previous page, excepting “16-bit monsters”- PIC 24FXXX and PIC 24HXXX- all PIC microcontrollers have 8-bit harvard architecture and belong to one out of three large groups. Therefore, depending on the size of a program word there are first, second and third category, i.e. 12-, 14- or 16-bit microcontrollers. Having similar 8- bit core, all of them use the same instruction set and the basic hardware ‘skeleton’ connected to more or less peripheral units.

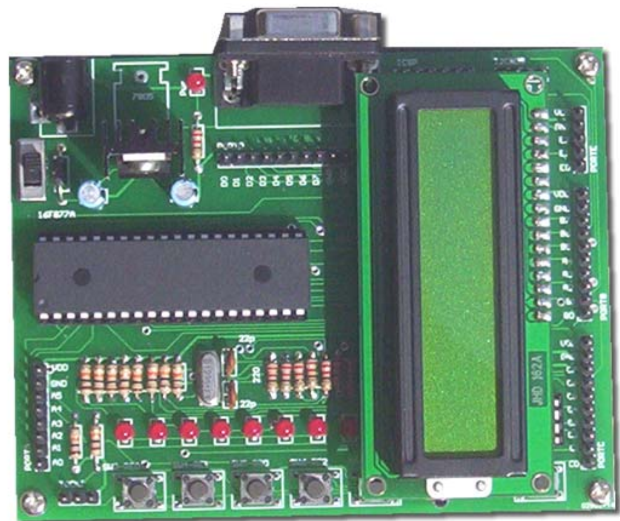
So far we have gone through an overall overview of the microcontrollers in general and PIC microcontrollers in specific.

We shall be talking about these various aspects in appropriate sections.

Chapter 2
Understanding Hardware

Well now we are into the actual business. Before working on the microcontrollers we must have appropriate hardware and software. This section will guide you about various hardware options. Software will be dealt with in next chapter.

In order to do various experiments with PIC microcontroller it is advisable to have a development board. The development boards have a wide variety of peripheral devices incorporated either on board, or as separate daughter boards. This makes it a complete, well almost, integrated development environment. In case you don't have a development board available, you can make a simple board using bread-board or vero board.



Microtronics Pakistan, PIC-Lab-II Development board

We have already discussed in Chapter 1 about the barely necessary circuit. A basic circuit needs only an external crystal oscillator, two 22pf capacitors and a 10-47K resistor on MCLR pin. Rest of all I/O lines are then available for experimenting. Development board on the other hand contains commonly used devices like push switches, infra-red sensor, LEDs, LCD and EEPROM etc. all on board, so that you don't have to worry about the wiring, and it is easier to manage.

A variety of development boards are available, having various levels of peripheral devices on them. A universal type of board is never possible, so you will always need either a couple of boards with differing peripherals, or make additional peripherals by yourself and plug them into the main board.

Most of the pins on microcontroller have designated functions, and therefore the associated peripherals are usually connected to them. However other devices can be attached to any I/O lines, like push switches, Pizzo buzzer and LEDs etc. Therefore it is important to know the architecture of your board so that you can change the program code presented in this book accordingly.

Since this manual is about Microtronics PIC Lab-II, we shall discuss in detail the hardware of this board.

In addition to the main development board, you need another piece of hardware, called Programmer. A large number of programmers are available, differing in speed and price, yet essentially all have the same function. We would prefer you to have the simplest programmer, as it makes the job easier and simpler.

Microtronics PIC Lab-II Features

Microtronics PIC Lab-II is an entry level development board, containing most commonly used devices, so that you can experiment with them easily. We shall discuss them one by one, however detailed discussion will follow in appropriate sections where they will be used.

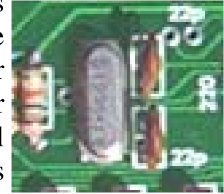
Microcontroller Socket

Obviously this is the most important part of the board. This board contains one socket for 40-pin DIP microcontrollers. Since all microcontrollers from Microchip® with similar pin counts are pin compatible, as far as power supply, MCLR and PORT pins are concerned, you have the liberty to use any microcontroller of your choice.

Microtronics PIC Lab-II comes with standard 18F452 microcontroller. You can use others like 18F4550,

18F4620 etc. if your particular application demands the power of those microcontrollers. Even you can use, very popular 16F877. In this manual we will be using PIC18F452 as an example. The microcontroller is placed in its socket, and does not need to be taken out for programming, as the board offers in circuit serial programming. We shall discuss this issue later.

Oscillator

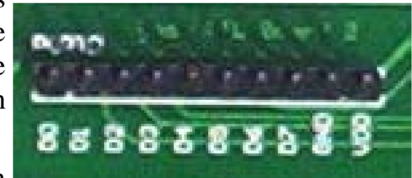


As previously said that every microcontroller needs an oscillator to synchronize its functions. A crystal oscillator has been used to give necessary oscillating frequency to the microcontroller. Faster is the oscillator, faster is the processing speed. However remember fast processing also requires more current. Since current is not an issue in our projects we will use the best frequency choice. The board comes with 20MHz crystal oscillator. This is the highest speed for 16F877 series, and most 18F series microcontrollers. However 18F series have an internal mechanism to multiply the clock frequency by 4 and generate an internal frequency 4 times that of the crystal being used. The highest frequency for 18F452 and many others is 40MHz. Thus if you want to run the 18F452 at 40MHz, speed using internal PLL multiplier, you must use 10MHz crystal on board.

Well don't worry, the crystal on PIC Lab-II has been mounted on a base, and can easily be pulled out and replaced with any other crystal of your choice. Presently in our manual we will use a 20MHz crystal frequency without internal PLL multiplier.

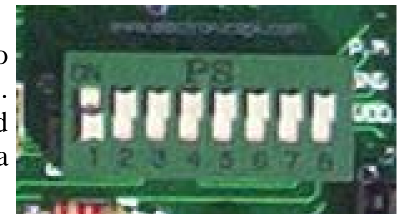
I/O PORTS

The 18F452 has 5 I/O ports, which we shall discuss later. These ports are named as PORTA, PORTB, PORTC, PORTD and PORTE. These ports are connected to various devices on board, which we shall mention later. However the ports have also been made available through headers, for use in other projects. Like if you make an LED sign, and want to control it with this board, you can connect appropriate header pins to the sign board.



The I/O port headers have been properly labeled on board, along with their pin names, like B0, B1 etc. each port header also contains 5V regulated power supply labeled as GND and VCC so that the external board may be powered up from the main MCU board.

DIP Switch



There is an 8 switch dip switch on board. These switches are attached to various devices on board, and used to enable and disable those devices. This has been done so, that many devices share common I/O lines, and mutually they may interfere in each others function. Thus disabling a particular device, would free its I/O lines.

For example LEDs on board are all connected to PORTC. The serial communicator UART also uses two pins of PORTC. If LEDs are enabled, they steel the signal from UART communication, and UART fails. Thus make sure if serial communication through UART is taking place the appropriate DIP switch for LEDs is turned off.

LEDs (DIP-SW-1)

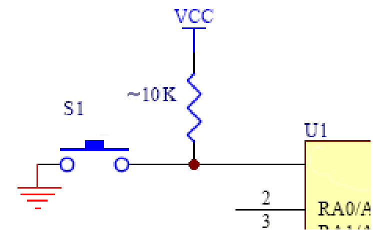
There are 8 LED indicators on board connected directly (through 220 Resistances) to the 8 I/O lines of PORTC. These LEDs can be used to monitor the status of PORTC, or other events. In general they will be

used to show you how to control an I/O line, so that in your real world projects, this line can be used to turn a relay or device ON and OFF.

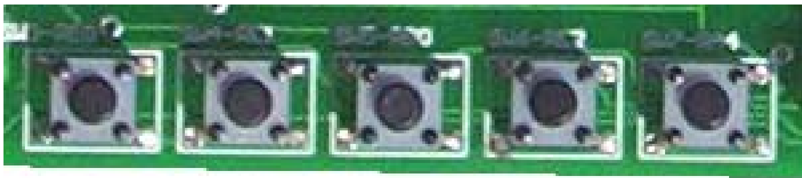
LEDs can be disabled through DIP-SW-1. Since PORTC is very commonly used by other devices, if the other device is not functioning properly disable LEDs.

Push Switches (DIP SW 4,5,6,7,8)

There are 5 general purpose push switches on board. These switches are connected to different port pins, which is indicated on board as well. There are two ways to connect the push switches to microcontroller. One is called active high, and other active low. In active high configuration switch is connected to VCC line, so that when switch is pressed a logical '1' is present on the I/O line. Whereas in active low the switch is connected to GND, so that when switch is pressed a logical '0' is present on I/O line. The active low configuration is most commonly used, and has been adapted on this board.



There are specific reasons for using different port pins with these switches. For example, PORTB.0 (Bit 0 of PORTB) can be programmed to sense an external interrupt, therefore a switch SW-5 has been wired to this pin to experiment this behavior as well. Similarly PORTA.4 can also be used to give clock signals to internal counters, SW-7 has been wired to this pin to facilitate these experiments.



Starting from the left SW-3 can be disconnected from I/O line using DIP-SW4, SW-4 Can be disabled by DIP-SW5 and so on.

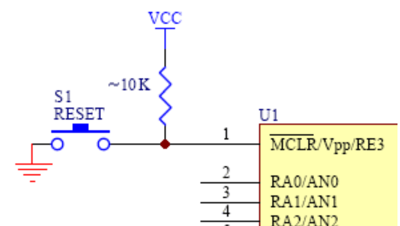
These switches are connected to following I/O lines:

SW3: PORTE.0 | SW4:PORTE.1 | SW5:PORTB.0 | SW6:PORTE.2 | SW7:PORTA.4

Since these switches are active low, a switch push must be checked as logical '0'

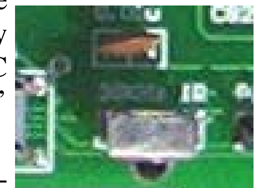
Reset Switch

Apart from Input switches there is a another switch labeled as RST. This switch has been wired to MCLR pin, and when pushed give logical '0' or GND to this pin. This has an effect to reset the microcontroller, clear all RAM and start program execution again. This button is specially useful when programming the microcontroller using an advanced technique called Boot-Loader. We shall explore this function later.



Infra-Red Sensor (DIP SW-3)

Infra-red communication is very commonly used in today's electronics. This can be a remote controlled application, or a full communication system. Infra-red waves exist around us in the form of heat, any ordinary infra-red sensor would erroneously pick up these stray beams. Commercially therefore a 38KHz modulated beam is used to communicate. PIC -Lab-II is equipped with 38KHz, modulated sensor, which will response as logical '1' only if a 38KHz modulated I/R signal is received.



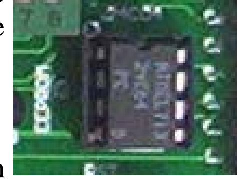
The I-R sensor is attached to PORTA.3 and can be enabled or disabled through DIP-SW3

I2C EEPROM (PORTC.3, PORTC.4)

I2C communication bus is very commonly used in electronics devices. More and more devices are coming up with I2C protocol support. The board contains an EEPROM Chip, (24c08) which is 8K EEPROM. This chip is attached to microcontroller using PORTC.3 and PORTC.4. The 40 pin, PICs have these pins attached to internal hardware of I2C communication. PORTC.3 is SCL and PORTC.4 is SDA. Due to built in I2C communication hardware, the software overhead is very much reduced.

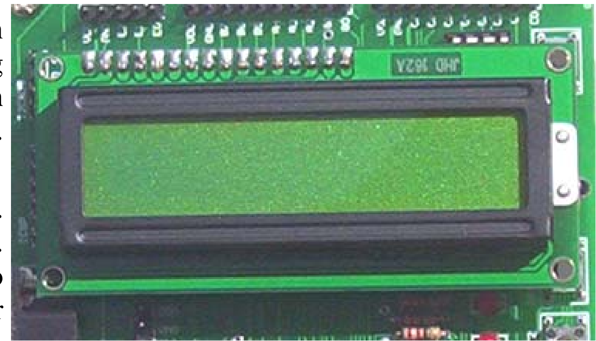
The I2C bus contains 10K pull-ups on both SDA and SCL pins. EEPROM is placed on an 8 pin base, it can be removed and another one inserted, if more memory is required. The EEPROMs have three address pins,

which can be used to connect many more EEPROMs in parallel. The board has however given this a permanent address of 000 . The I2C code for EEPROM is 1010. next three bits would be EEPROM chip address, and last bit is direction. Therefore the complete address to write on EEPROM would be: 1010 000 0 and to read it would be : 1010 000 1



Character LCD (PORTD.2,3,4,5,6,7)

LCDs are becoming more and more popular in electronic devices to communicate with user. There are several LCD controllers, each having its own unique communication protocol. Hitachi HD44780 is a very popular and industry standard LCD Communication controller. This controller is built right on to the LCD module. Many high level programming languages provide ready to use libraries for communication with this device.



PIC Lab-II uses the same protocol, and contains an adapter to accept standard LCD modules. You can plug into the LCD module when required, and change it with another one with more lines and characters if required. The standard board comes with 16 x 2 character LCD.

The board is configured to drive the LCD in 4 bit mode. We shall talk about LCD modes later in section on LCDs. Here just remember that the LCD will be attached to following I/O lines, and you will need to tell your software about the wiring.

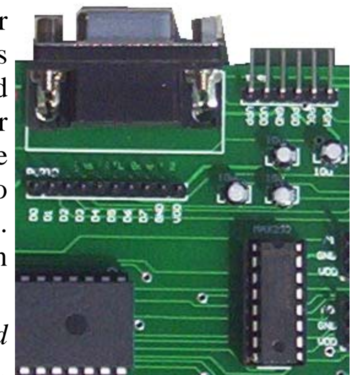
The module uses 4 bit mode, in which the highest 4 bits of PORTD are connected to data pins of LCD. PORTD.2 is connected to Enable Pin, and PORTD.3 to RS-Pin of LCD. Thus following declarations must be used before initializing the LCD (Proton Basic).

```
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
```

LCD can be enabled or disabled by using DIP-SW2.

UART (Universal Asynchronous Receiver and Transmitter) PORTC.6,7

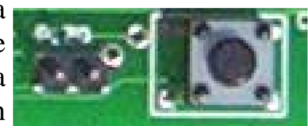
The board contains a standard universal Serial Asynchronous Receiver and Transmitter. Many devices use this protocol to communicate with other devices. The communication is hardware independent, and just needs two wires, one for transmission and one for receiving data. PCs and some other devices, use a level translator, to redefine the standard signals for logical 0 and 1. this is done so, to minimize noise interference as well as prolong communication distance. To use these signals, they must be converted back to TTL level logic. The PIC-Lab-II board contains Rs-232 level converter which converts these signals to TTL level, and to transmission levels while sending data. Most PIC microcontrollers contain an internal hardware to manage this communication, so that software development becomes easy. PORTC.6 and PORTC.7 are configured as hardware USART communication pins.



NOTE: since the PORTC is also connected to LEDs, if LEDs are enabled receiving data from USART is interfered. It is therefore mandatory to disable, LEDs while using UASRT.

PIZO Buzzer (PORTA.5)

The board contains a connector for PIZO. The PIZO buzzer, module consists of a transistor and two resistors. The transistor connects directly to PORTA.5. The buzzer has to be given oscillatory signals, like a train of 0s and 1s to make a sound. Unlike other buzzers which produce a fixed note of 1KHz, when given power, this board uses a raw PIZZO, or even a small speaker, to control the oscillating frequencies. The buzzer connector is located next to the Reset Switch.



In Circuit Serial Programming Connector

Microchip offers in-circuit serial programming in its newer chips. PIC Lab-II has been designed to comply with it. This feature requires two pins of Portb, B6 and B7 along with MCLR pin. At the time of programming, B6 and B7 must not be connected to any other device, like LEDs or a driving circuit which may interfere with the programming data. This board by itself keeps these two pins free. However the pins are part of Portb and as such are available through port header for expansion boards. In case your expansion board is using these pins, and ICSP fails, disconnect the board cable before programming.

In Circuit Debugger

Microchip has introduced the debugging facility into the newer chips. For this purpose microchip has introduced a newer device called ICD-2. This device is both a programmer as well as debugger, and can be used to inspect the various microcontroller registers and variables, as well as watch program execution, right in its circuit. The same header as for ICSP is used for ICD-2 connector. The board fully supports this feature.

In addition to the regular lines of MCLR, VCC, GNG,PGD and PGC the connector also has a line for B3, which is for Low Voltage programming, as well as for debugging with some other third party debuggers.

I2C Bus Connector

I2C is a commonly used communication system in electronic devices. This board uses EEPROM as I2C device. Although you can use any I/O lines to act as I2C bus, Microchip has introduced hardware integration of this service into its newer chips. Thus pins connected to PORTC.3 and PORTC.4 are also internally connected to hardware driver of I2C. Since I2C can support up to 7 different devices connected to the same two lines, we have provided these two lines along with Power as a connector, so that if you have another device with I2C communication, it can be directly plugged into the bus, instead of connecting them individually to I/O lines and power supply. You can use this connector with applications requiring two I/O lines as well. Its not hard coded for use with I2C. However remember these lines have two pull resistors on them and EEPROM is also connected.

TOCKI Connector

Timer 0 Clock Interrupt Input. PORTA.4 pin is also used as a clock input pin for Timer 0 module. This can be used in applications requiring an external input to count the pulses. Although connected to PORTA.4, a separate connector has been provided along with power supply to use this pin directly.

This connector can also be used in applications where only PORTA.4 is required, like setting up a serial communication between two boards, or connecting to a modulated I/R Led etc. the left most pin is RA4, middle GND and Right most is VCC.

Remember SW-7 is also connected to the same pin, therefore if required free this switch using DIP-Switch.

Pulse Width Modulation Connector (PWM)

Pulse width modulation is a common technique used in electronics to control the amount of DC power delivered to a device. Although you can use any digital line to produce PWM, it will however require the attention of microcontroller all the time to regulate it. Newer PIC devices have two or more PWM modules built into the chip, which are connected to specific output pins. When properly configured they continuously give PWM signals on their specific pin, without attention of main CPU cycle. We have taken out PWM1 (or CCP1) pin which is same as PORTC.2 as a connector along with power for daughter boards. This facilitates the connections. If second PWM is required you can get the connection from appropriate port header.

Analog Input 0 and 1

Analog input is commonly required in many applications. Although 40 pin PICs have 8 analog inputs on different lines. If required those pins can be directly used in applications. However to facilitate the job, PIC Lab-II has two analog input pins (AN0 and AN1) directly taken out along with power for external projects. You can directly connect analog inputs of up to 5V to these pins. Like a variable resistor, or LM35 temperature sensors can be directly connected.

Power Supply

PIC Lab-II requires 5V power to operate. For this purpose a 7805 regulator has been used. Power input can be given through an adapter jack. You may also use a 9V batter with a suitable adapter for this purpose. The adapter should be from 6-12V, preferably 9V. Center pin should be Positive. A blocking diode is there to prevent reverse polarity.

Power supply from ICD-2

Microchip ICD-2 has the option of powering the target board. If connected this board can take power supply from ICD-2. Before connecting the board to ICD-2 make sure the connector is oriented correctly. As wrong polarity on power pins will damage your board. The connector provided on PIC Lab-II is same as defined by microchip®.

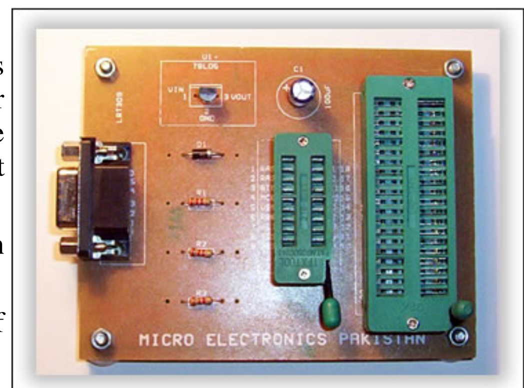
PIC Programmer

So far you had a tour of the PIC Lab-II anatomy. Now you know what devices are there on board, and where their connectors are located. Now we come to the second part of hardware device to start with. This device is called Programmer.

Programmer is a device, or piece of hardware which will accept the compiled program from your computer and write it into the program memory of your microcontroller. Since this memory is flash based, once the microcontroller is programmed, you do not need the programmer. Whenever you will turn the power on, the program in microcontroller memory will start. However whenever you make changes to the software, the newly compiled program has to be written back into the microcontroller. You will again need the programmer device to do so.

There are hundreds of programming devices available, in market. Each having its own merits and de-merits. One of the most popular device is one from Microchip® itself. A number of commercial third party devices are also available in market. All these devices differ in the list of supported devices. A few designs are available for students which make use of very few components yet do the job. Price therefore is another factor in choosing a Programmer. We shall introduce you various programming devices available from Microtronics® Pakistan for use with PIC Lab-II. This list is by no means final, and for the latest devices do visit the web site, www.electronicspk.com.

PIC PG-I



PIC programmer-I is the simplest programmer possible. This programmer is connected to the serial port of your computer and the microcontroller to be programmed is inserted into the ZIF sockets. After programming the microcontroller is taken out and inserted into the application board to run the program.

The programmer has sockets for both 18 pin as well as 40 pin PIC microcontrollers.

It does not require external power supply, and programs most of the commonly used microcontrollers.

Since this programmer takes its power supply from the PC serial port, some PCs, specially Laptops do not have enough power available on serial port and therefore it can not be used with Laptop computers. Secondly since it does not support In circuit Programming, you will have to remove the microcontroller every time from your host board, program it and re-insert back. Although a boring job, yet its good for a beginner for the price its offered. Moreover its general purpose, and can be used to program your chips for use in other projects. This design does not support 18F series of devices.

PIC PG-II

PIC-PG-II is the next version of PIC-PG-I programmer. It supports In-circuit serial programming. Moreover it can also program the 18F series of microcontrollers. However since this programmer also takes its power supply from host PC serial port,

it does not work with laptop computers. Considering the simple design, low cost and In-Circuit programming capabilities this programmer is recommended for beginners with PIC-Lab-II. In order to program ex-circuit PIC microcontrollers you will need an adapter board for use with this programmer.

Anyway in order to use with PIC-Lab-II you just need this programmer and adapter board is not necessary.

PIC 16 QL-2006 Programmer

This is a professional quality commercial programmer. This programmer supports a wide range of PIC microcontrollers. The ZIF socket allows all types of 8, 12, 18, 28 and 40 pin PIC microcontrollers to be inserted and programmed as ex-circuit. However the programmer also has In-Circuit programming option, a cable is connected to the standard ICSP connector and it works as ICSP as well.

The programmer has its own power supply, which makes it work with laptops as well.

This programmer has dual input, and can work with Serial port as well as USB ports. When connected to USB, it can even take supply from USB.

This programmer is recommended for more serious developers.

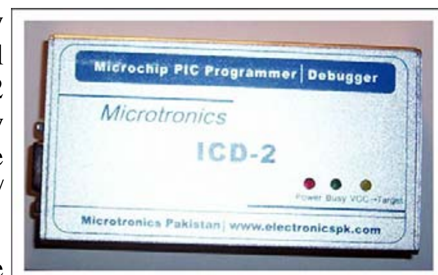
It can be directly connected to PIC Lab-II ICSP connector.

Microchip In-Circuit Programmer / Debugger –2

Microchip® the manufacturer of PIC microcontrollers have produced their ICD-2. This device can be connected to serial as well as USB ports and can program a huge range of microcontrollers, in circuit. Not only that it can program, but it can also debug the software running inside the microcontroller. The device is controlled from microchip software MPLAB©. From the MPLAB you can stop the program, step over, step into, animate and halt the software. You can then examine the status of various registers and program variables.

This device is invaluable for experienced programmers and developers making complex software. Debugging a complex software is not an easy job!

Microtronics ICD-2 Clone



ICD-2 is a product from Microchip®, its expansive and not easily available in local markets. Considering the usage and beneficial features of ICD-2 Microtronics Pakistan® produced their own ICD-2 Clone. This ICD-2 works from serial port and has 100% compatibility with Microchip® ICD-2. Available in a price much less than the original, and availability in local market, makes this a programmer/ debugger of choice for the professional.

Well now you have a choice of a number of programmers, all of these will work, however to start with we suggest using PIC-PG2 programmer, and later thinking of upgrading to Microtronics ICD-2 clone.

Microchip® Self-Programming System

Microchip has introduced recently a new technology in its newer microcontrollers. This capability allows these microcontrollers to acquire the new program through its serial port connection, right in-circuit. This feature does not require any external programmer, and is quite fast and reliable. However this feature requires to load a piece of software called ‘Boot-Loader’ into the microcontroller using a conventional programmer. Once the Boot-loader is there, it can take new programs, using serial port, and write them into the program memory. The Boot-loader itself remains unaffected by new program.

A number of companies, including Microchip® are providing Boot-Loader software. PIC lab-II comes with a Boot-Loader program as well, and in this manual we shall learn, how to use both conventional programming, use ICD-2 and Boot-loader.

As you can see there are number of methods to program the microcontroller. Remember, if many solutions exist for a given job, each has merits and de-merits. There are advantages and disadvantages of all these methods, so you must be prepared to choose the right one for a given situation.

For now we will be using PIC-PG-II programmer, connected to the PIC Lab-II board. In order to work, we have to install the necessary supporting software, which will communicate between the PC and PIC-PG-II programmer.

Chapter 3
Setting Up The Programmer

Microtronics PIC PG-II is a cost effective, simple and trust worthy programmer. This programmer is based upon a popular design called JDM. This design requires a serial port from your computer and draws all the necessary current and power from the serial port. Most desktop computers can be easily used with this programmer. However some laptops, may have low power on serial ports and therefore can not be used. Moreover newer laptops do not have serial port at all, and they have only USB support. USB to Serial adapters also do not work. In that situation you have to get a USB based programmer.

Nevertheless a hobbyist usually practices all this in his laboratory, where a desktop computer with serial port is available.

A Note on Programming

By programming here we mean a mechanism to transfer the compiled program into the microcontroller program memory. PIC microcontrollers have a separate area of memory called program memory. The size of this memory differ in various chips. PIC16F628 has 2K program memory, whereas 16F877 has 8K program memory and 18F452 has 32K program memory. Just to make you understand, do not consider these small memories. As students from PC world are used to talking about megabytes. Most of your programs, will not exceed few hundred bytes! What to talk about kilobytes? These devices are not meant to run windows, but to control a specific device based upon certain input and logic.

In order to put your microcontroller into program mode, the MCLR pin has to be driven up to 12-13.5V. This is referred as VPP. The VPP is generated by programmer. Once VPP is applied to MCLR pin, the processor stops functioning and accepts data from programmer on PGD and PGC pins, which are RB7 and RB6 pins on microcontroller. The programmer first erases the old program memory and then writes new program and EEPROM data if required. After the program is transferred it is verified. After successful programming, the VPP must be taken down, so that the program may be started.

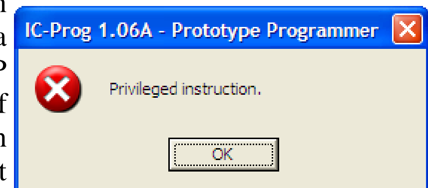
In order to program the PIC, your RB6 and RB7 pins must be free from any devices. By default they are free on PIC Lab-II board, however these pins are available on port header for daughter boards, if a daughter board is using these pins, disconnect the board before programming.

Installing The Software

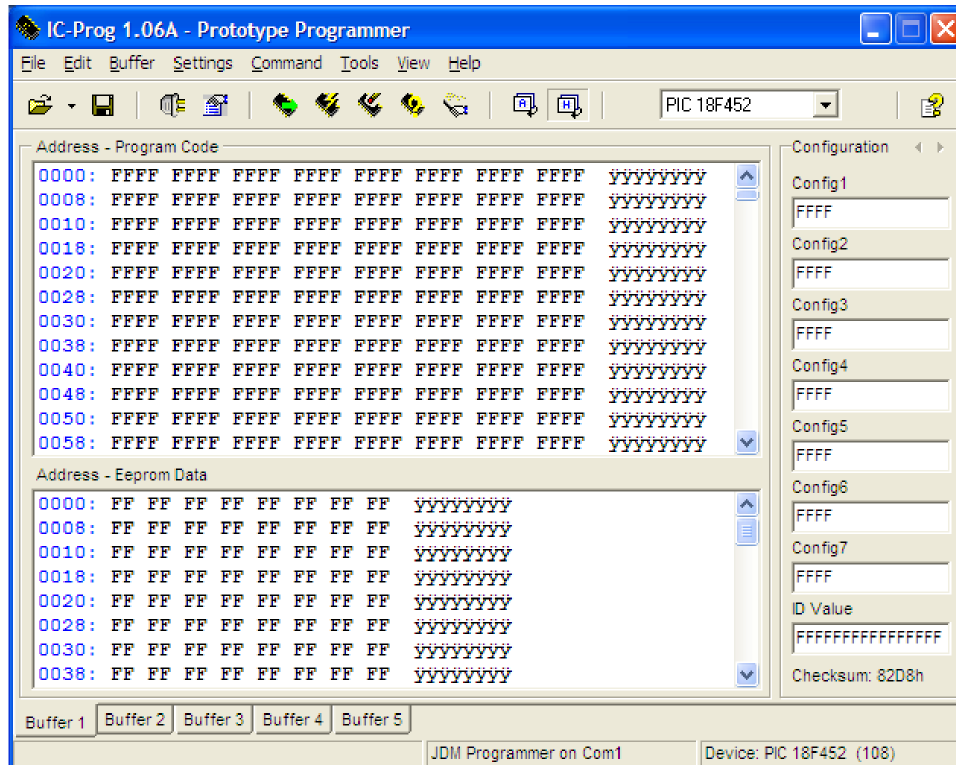
Before you use this programmer, you have to install an application software on your PC. There are many available, on internet. However some commercial programmers have their own software. There is nothing different, the purpose and method of using is almost same.

One of the most popular software used by many is ICPROG. This is a freeware and can be easily downloaded from internet (www.ic-prog.com). The software has been provided on the accompanying CD of PIC Lab-II. In order to work on Windows XP, you will also need to download the IC-PROG NT/2000 driver. For your convenience they have been included in the ICPROG Folder on the CD.

Just copy the ICPROG105D folder, or whatever is present in your CD into some suitable location on your computer. I prefer copying it to D:\ICPROG105D

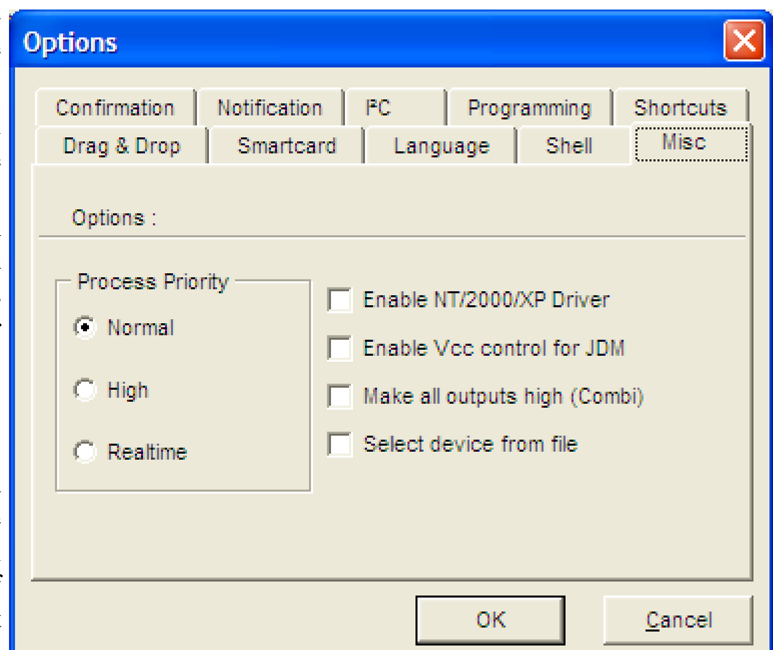


Now open the folder and run ICPROG.exe file. First time when you run it on your computer, this will give an error message, indicating a privileged instruction error. This error is indicative of windows XP driver, not installed. Just click over OK button, and the main screen of ICPROG would appear. Now click over the Settings menu, and then on Options. The options dialog box would appear with many tags. Select the Misc tag, it will show various options. Select the Enable NT/2000/ XP driver check box, this will immediately install the windows XP driver and IC-PROG will restart. This time it should not give the above error.



Next you have to setup which COM port your computer will use for communication with Programmer, and to indicate to IC-PROG that we will be using JDM Type programmer.

Again Click on Settings menu, and this time select Hardware, a dialog box will again appear, in the combo box a number of programmer types are listed, select JDM Programmer, and below there will list of all COM ports your computer has, select the one to which you have plugged in the serial cable.

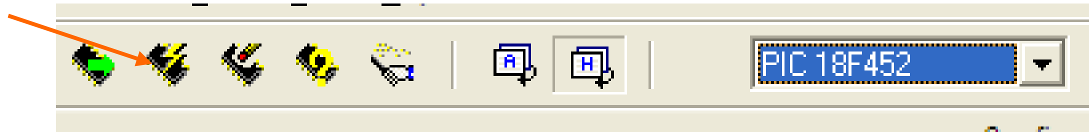


Leave everything else as such. Now you have to select the Microcontroller you are going to program. Again click on Settings, Device and then on Microchip PIC. A long list of supported devices would appear, click on more and more devices would appear, locate the PIC18F452 (or 16F877 / 16F877A if you are using those).

That is all. Your IC-PROG is now set. Your toolbar now should look like this:

Notice the name of selected microcontroller in drop-down box. The first icon from left with green arrow will be used to read the contents of microcontroller program memory. Next

Icon will be used to program the new software and third icon will be used to erase the contents of



microcontroller. Normally you will only use the Program icon/button, this will automatically erase, program, and then read to verify the microcontroller.

Now attach the programmer to the serial cable, and serial cable to your computer serial port. Attach the programming cable, which has 6-pin connector on both sides to the programmer, header. Note the header has clearly marked pin labels, which are according to the microchip specifications. Attach other end of the cable to ICSP connector on PIC Lab-II board. Make sure this is connected in proper direction, so that the VPP pin on PIC Lab-II is connected to MCLR labeled pin on PIC PG-II. Once connected, the power LED on PIC Lab-II may light up, as board will be receiving 5V supply from programmer. Normally this 5V supply is enough to program, the microcontroller alone, if connected to an adapter, however, since board has a number of other devices, this 5V supply is not enough. Connect the power supply of your PIC Lab-II board, and turn the power switch ON. (PIC Lab-II must be powered while programming).

Now your things are setup, notice the two panels on ICPROG main screen showing all FFFF indicating blank. Now click over the Read button (or press F8). The red LED on programmer should turn on, and a progress bar to indicate reading data from microcontroller should appear. After reading, a small pre-programmed program, should appear in the IC-PROG Program code area. Some numbers other than FFFF. That indicates your IC-Prog has successfully contacted the microcontroller through JDM programmer and is able to read data from the IC.

Writing Program into the Microcontroller

Writing a program into the microcontroller is fairly easy. All you need is the compiled file. The compiled file has an extension .hex. This file contains instructions which are understandable by the processor. Keep it in mind that the internal structure and codes of commands will differ from processor to processor. In other words a .hex file is compiled for a specific processor. Thus the hex file for 16F877 will not work on 18F452.

Well now you have the right .hex file, open Icprog click on file and open the .hex file of your choice. Make sure the programmer (PIC PG-II) is connected to the development board, and the board power is ON. Click on the second Icon from left on toolbar of Icprog. This will write the contents of loaded hex file into the microcontroller. Once its loaded and verified, a message indicating successful programming appears. That is all. **Now disconnect the programmer from development board**, and turn the development board ON. The program should start running and producing any output it is designed to.

As an example use test.hex file located in samples folder. This file has been compiled for 18F452 microcontroller, running at 20MHz. This file will make all LEDs blink. Thus make sure that DIP SW-1 is ON to enable on-board LEDs.

Chapter 4

Setting Up The Proton Basic Compiler

So far so good. You have setup your hardware, setup the programming software, that will transfer the hex file into the microcontroller using PIC PG-II programmer. Now you must be thinking how to create the hex file? What if I want to change the speed of blinking LEDs so on and so forth? The answer is that you will have to write a program in some programming language, and then using a translator, called compiler, convert the program written in English into processor understandable hex file.

A programming language itself is nothing, but a collection of words, called commands or statements and a group of rules to use them. Just like any other language. Like English has words, also called vocabulary and a set of rules, called grammar to use it. The rest of story lies on you, the software developer how you use these commands and grammar to make anything useful.

A number of programming languages are available, these include Assembly, C / C++, BASIC, PASCAL, JAL and many others. All of these languages differ in the set of commands.

Assembly Language

Assembly is the most generic programming language, and until recently this was the only language available for microcontrollers. Assembly language has command set which matches one to one with the processor understandable instructions. However you write those instructions in English like manner, like MOV AX, 2 tells the processor to write a value of 2 into AX register. This language is very powerful in terms of control over the processor. Essentially you have total control over the processor. Writing applications in assembly is difficult however, as you have to remember the functions of specific registers, and memory locations etc. the program lacks structured approach and is quite lengthy. Any way the source program which you write in assembly has an extension .ASM this text file containing assembly language instructions is assembled using a software called assembler into the .hex file. Assembler is a software that has to be installed on your PC. The assembler is specific for PIC microcontrollers and can be downloaded from Microchip® site. We shall not use this method of programming.

C/C++ Languages

C is the language of professionals. This is a high level language, and contains many powerful commands, which would otherwise require lots of commands in assembly. A number of compilers using C as a programming language are available. Their light-versions can also be downloaded to work with them. Differences among various compilers is in the quality and types of libraries offered by the manufacturer. Libraries are pre-compiled codes, in other words commands available to us. The more extensive is the library, more easier it will be for you to write software.

BASIC Language

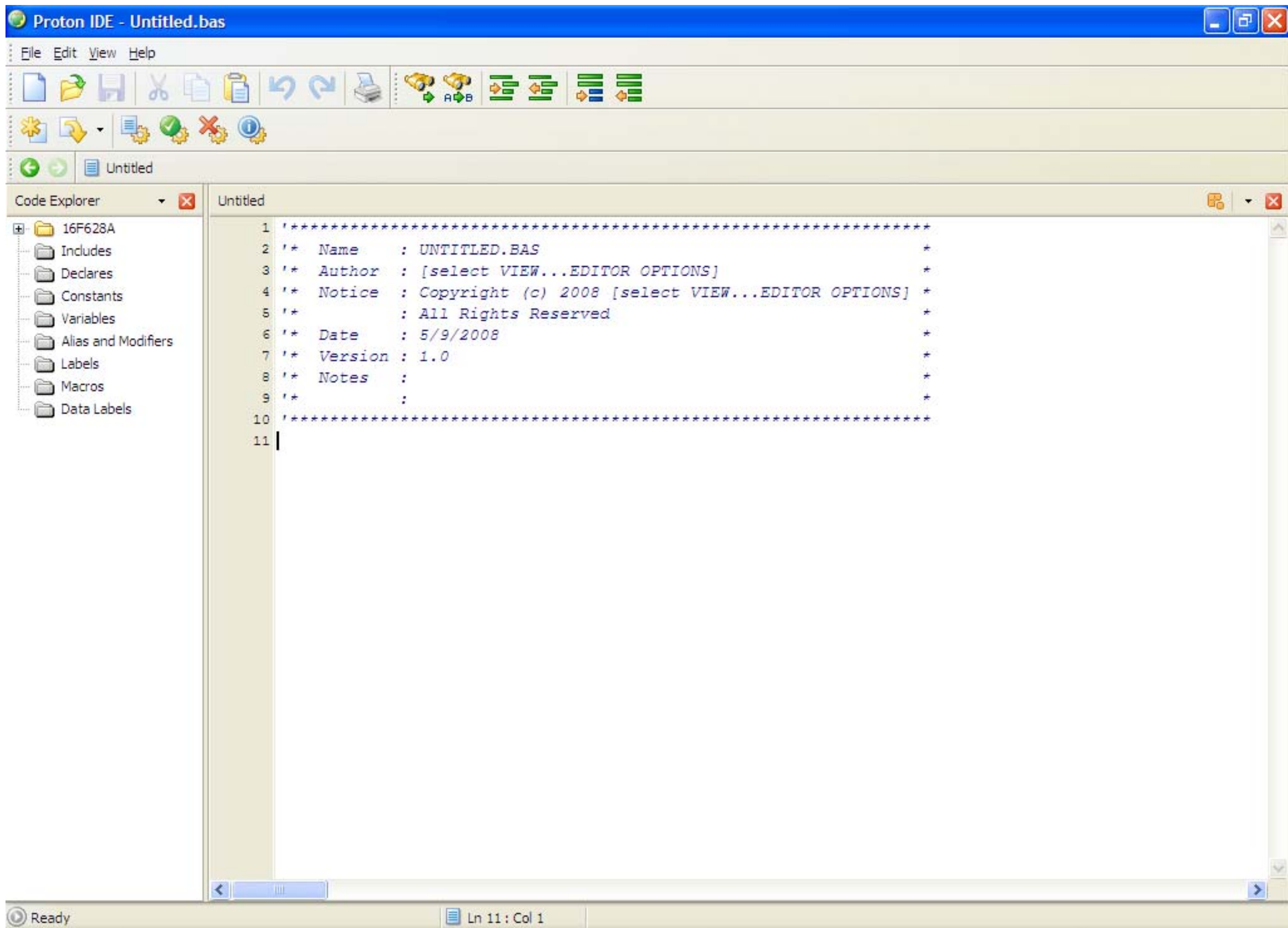
BASIC stands for beginner’s All Purpose Symbolic Instruction Code. This is a very popular programming language, both for microcontrollers as well as PCs. The commands and syntax of language is fairly simple, and English like. The beginner therefore finds it the best to start with.

In this manual we will be using BASIC language compiler and integrated development environment from Crown hill inc. UK. This complete suit is called PROTON BASIC. You can also find many other companies providing compilers for BASIC language, like MikroBasic from mikroelektronica. You can download the trial version of Proton BASIC from www.picbasic.org which is the official site of BASIC language compilers for PIC Microcontrollers. Keep it in mind while BASIC language will remain the same but the compiler will be different for other series of microcontrollers, like ATMEL, or ARM etc. So make

sure whatever compiler you use, is meant for PIC microcontrollers.

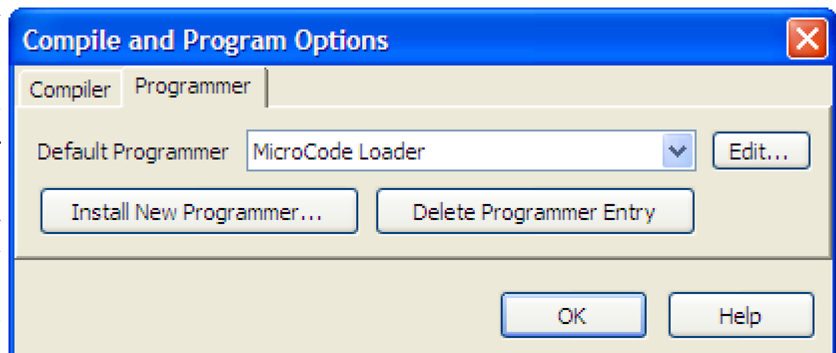
Note, that the free or light version of PROTON BASIC has some limitations. It supports only 16F628A and 16F877 microcontrollers, it does not support 18F series at all. Secondly the source file is limited to 50 lines of code which is OK for beginner but not for real applications. The CD ROM with PIC LAB-II contains full version of PROTON BASIC compiler. Instructions on installing and setting up this compiler are located in the readme file in the appropriate folder.

When successfully installed the proton Basic IDE (integrated Development Environment) would look like



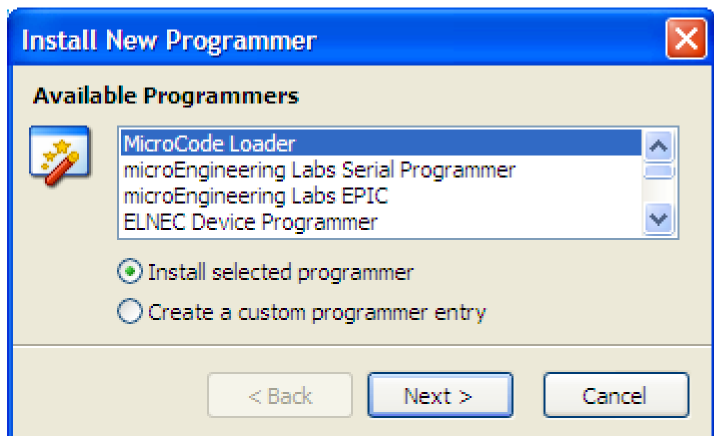
this. There are two panels one larger panel on right, is the main editor where you will write and edit your BASIC language source program. The smaller left panel is called 'Code Explorer' and shows various labels, variables and registers etc available in the program. This is only to facilitate development, otherwise it can be turned off..

This software will compile the basic language program into the .hex file. After that you will load the ICPROG and open the .hex file to be transferred into the microcontroller. This IDE can facilitate a bit more, that you can set your programming software into this IDE. So that after compiling the IDE will automatically load ICPROG and open the just compiled .hex file ready to be transferred into microcontroller.



To make this setting click on view and then on Compile and program Options. Select the Programmer tag. The default programmer selected here as shown in this figure is Microcode loader. Click on Install New programmer button. A series of pre-defined programmers is listed

as shown below, our ICPROG is not listed in it. Select the Create a Custom Programmer entry and click Next. A display name will be asked, enter anything you like, let it be Microtronics and click Next. In the programmer File name enter, ICPROG.EXE and click next. Now a dialog box appears to locate the folder where your ICPROG.EXE is located. I suppose its in D:\ICPROG105D folder. You can choose Find Automatically or Manually. If you press Find Manually button a tree will appear and you will have to locate yourself the folder where ICPROG was copied. After that you select OK and then click next.



Now its asking for parameters. These are the parameters that will be passed to ICPROG when its called. Enter -L$hex-filename$ Write this as such, including the dash before L and two $ signs. Click Finish.

Now your ICPROG is also integrated with the PROTON BASIC IDE. If for some reason you fail to do so. Don't worry, all you will have to do, is after compiling the program, manually load ICPROG and open the hex file.

Notice these two buttons on the top tool bar of IDE. The left button is for compile only. When pressed it will only compile the program and produce hex file. The other button is for compile and program. Notice a small arrow on its side, click this arrow and a list of installed programmers would appear. Select Microtronics, which is the one we have just configured. After that whenever you will need to compile and upload the program, you will just press this button.

Writing Your First Program

Well, finally you are all done, and time to test if we can write our own program. We shall be saving all our programs in a separate folder let be: D:\PICPROJECTS. Proton Basic has a known issue, that it does not allow a space in file name, or its path. So don't save your programs into 'My Documents' or any other folder with a space in its name, you can use an underscore. If you are using Lite version it does not allow a number as last character of file name.

Well in the IDE editor window enter the following program. Notice the commands are automatically highlighted and colored while you type, this makes program reading easy.

```
Device = 18F452
XTAL = 20
ALL_DIGITAL=true
Output PORTC
PORTC=255
End
```

After entering the program save it to your folder D:\PICPROJECTS and name the file as 'Test.Bas' the .BAS indicates that this is the source file of BASIC. Now click on the Compile and Program button. This will invoke the compiler, which will translate these English like commands into processor understandable .hex file, if everything is OK, it will automatically load ICPROG, and the contents of test.hex are already loaded into it. Now make sure your programmer (PIC PG-II) is connected to the PIC Lab-II and PIC Lab-II power is ON. Click on Program All button. This will transfer the program into microcontroller. When Success message is displayed, Turn the power OFF, and **disconnect the programmer from PIC Lab-II**. Now turn the PIC Lab-II ON, make sure that DIP Switch SW-1 is On to enable LEDs. All LEDs connected to PORTC should light up. If you get this result you are done, and ready to proceed to regular experiments. If it does not, recheck the entire process, there must be something wrong somewhere.

You can not proceed until this test succeeds, which indicates that your hardware and software have been properly set.

Chapter 4

BASIC Programming Language A Primer

Basic is a simple and easy to learn programming language. It has only few rules and control structures which define its grammar. In this section we will learn about some basic principles of this great programming language. The codes presented here are not meant to be programmed into the microcontroller as such, but are given to explain the topic. Once you have gone through this introduction to the BASIC language, only then you can jump into the specific areas of your interest. Whatever is presented here will be repeated many times through next chapters.

Structure of BASIC program

The basic program consists of :

- Program Header
- Declarations
- Symbols and Identifiers
- Statements and commands

In addition to these basic structures, some compilers also allow object oriented programming as well as procedures and functions. However Proton Basic does not allow procedures and functions in the true sense as well as does not support objects.

It has simple and straight forward approach called Top-Down approach. The program starts at top, and proceeds towards bottom. However it can branch back towards top, to allow repetitions.

The first few lines of BASIC program would tell the compiler about the hardware. Since different PIC microcontrollers differ in memory, EEPROM, number of PORTS and registers etc. It is therefore necessary to inform the compiler about the microcontroller to be used. Secondly processing speed depends upon the crystal frequency. Therefore in order to calculate the timing accurately for delay functions it is also necessary to inform the compiler the crystal frequency.

Thus the BASIC language programs will usually begin like:

Device = 18F452

Xtal = 20

The first line is indicating the processor and second line tells that hardware will be using 20MHz crystal.

Declares

Declares are special instructions about various devices to be used, this helps the compiler to generate specific instructions. For example if we are using an LCD and it is connected on PORTD, then we have to inform the connections of our LCD. We shall declare this type of setup usually after the header section using declare commands.

```
Declare LCD_DTPIN PORTD.0
```

There are a number of declares, however only the ones required in current project are usually set.

Identifiers

Identifiers are special text symbols used to represent something. They can be used as labels to mark certain locations in the program, so that program can be made to jump to those labels and then continue the program thereafter. Similarly Identifiers can be used to name certain memory locations. These are usually called variables, and are the most important identifiers in programming. Identifiers can also be used to alias certain text, so that instead of writing the text you can just write the identifier, and during compilation the

compiler will insert relevant text in its place.

Statements and Commands

There are three main types of statements:

- Comparison and conditional statements
- Repetition and looping statements
- Library Commands

Comparison and conditional statements allow us to compare two or more variables, ports, port pins or special function registers and then make a decision to execute one set of instructions or other set. Considering the importance of these statements, BASIC language provides many different constructs of this. We shall explore these below.

Repetition and looping is one of the greatest advantage of microprocessors. We can instruct the microcontroller to continuously repeat certain instructions, either endlessly, or till a certain condition exists. For example to keep an LED on, till the temperature is high from a set point. Again since they are important structures a number of methods exists to control loops.

Library commands, are not truly speaking commands of BASIC language, but are provided by the manufacturer of compiler to do the common tasks. For example a library command to display some data on LCD or to read analog data from an input pin. The more extensive is the library, more powers you have and shorter is the programming time.

With this review, lets talk about individual topics one by one, in context of Proton Basic.

Labels

In order to mark statements that the program may wish to reference with the goto, call or gosub commands, PROTON+ uses line labels. Unlike many older BASICs, PROTON+ doesn't allow or require line numbers and doesn't require that each line be labelled. Instead, any line may start with a line label, which is simply an identifier followed by a colon (**:**).

```
LAB:
PRINT "Hello World"
GOTO LAB
```

Label names can be up to 32 characters in length and may contain any alphanumeric character, but they must not begin with a numeric value. For example:

```
LABEL1:
```

is perfectly valid, however:-

```
1LABEL:
```

will produce a syntax error because the labels starts with the value 1. A label that contains more than 32 characters will produce a syntax error pointing out the offending label. Underscores are also permitted as part of the label's characters. This helps create more meaningful label names. For example:-

```
THISISALABEL:
```

does not have the same clarity of meaning as:-

```
THIS_IS_A_LABEL:
```

Variables

Variables are used to temporarily store data or to hold numbers to be used in calculations. The number of variables which can be used in a program depends upon the RAM of your microcontroller. In Harvard architecture, the RAM part of memory is separated from program memory. Therefore if you have 16K of program memory and 256 bytes of RAM, you can not use the free program memory to store data. Variables are therefore nothing but memory bytes. To facilitate this job, compiler allows you to give these memory locations names, called variable names. It will then automatically compute the correct address of RAM, when a memory variable is used in your program.

Although variables exist in RAM, as a sequence of bytes, yet they can be grouped together to make larger organizations to hold different kinds of data.

Data types are defined as various types of data which can be manipulated by our compiler. Compilers from different manufacturers differ in this facility, however certain standard data types are supported by all.

The variables are declared using a Dim statement, followed by variable name and its data type. Dim statements can appear anywhere however it's a good programming practice to place all Dim statements after the declares at top of the program.

```
Dim Dog As Byte             ' Create an 8-bit unsigned variable (0 to 255)
Dim Cat As Bit              ' Create a single bit variable (0 or 1)
Dim Rat As Word             ' Create a 16-bit unsigned variable (0 to 65535)
Dim Large_Rat As DWord      ' Create a 32-bit signed variable (-2147483647 to
+2147483647)
Dim Pointy_Rat As Float     ' Create a 32-bit floating point variable

Dim ST As STRING * 20       ' Create a STRING variable capable of holding 20
characters
```

The data types as Bit, Byte, Word, DWord, Float and String define the number of bytes reserved for the variable. This also defines the number range, which can be stored, as well as the nature of stored number. The stored numbers can be signed or un-signed as well as they may contain a decimal point. A string on the other hand, is a collection of Byte sized variables, to hold character data.

The compiler itself will use some memory to store its internal variables. The amount of RAM used by compiler depends upon the complexity of the program. As there are more and more control structures, and loops, so the compiler will use more and more RAM. Keep this fact in mind that compiler is sharing your RAM.

Variable names follow the same general guide lines as identifiers. However there are certain reserved words which can not be used as variable names. See documentation of your compiler for details of reserved words.

Accessing Part of a Variable

Many a times part of a variable needs to be accessed, either for reading or writing. Most of the times in a Byte sized variable a particular bit needs to be accessed. A Byte consists of 8 bits numbered from 0 to 7. Bit 0 being the least significant and bit 7, the most significant. An individual bit of a variable is accessed by a period followed by bit number in a variable name. Thus if we have x as a byte sized variable, its least significant bit can be accessed by x.0 and most significant bit by x.7.

```
Dim x As Byte
Dim y As Byte
x.0 = 1
y.7 = x.0
End
```

In this example x and y are byte sized variables, x.0 = 1 sets the bit 0 of x as high, and y.7 = x.0 reads the value of x.0 and transfers it into y.0

In case of word sized or DWord sized variables, the same can be done, in addition the high order byte and low order byte, or Byte0, Byte1,Byte2 etc can be separately accessed.

```
DIM DWD as DWORD              ' Declare a 32-bit variable named DWD
DIM PART1 as DWD.WORD0        ' Alias PART1 to the low word of DWD
DIM PART2 as DWD.WORD1        ' Alias PART2 to the high word of DWD
```

Symbols

Symbols are in fact a way to simplify things. It assign an alias to a register, variable, or constant value. That alias will then be used in your program, the compiler will replace the alias with actual data before compilation.

```
Symbol LED = PORTB.0
High LED
```

In this example a symbol LED has been defined and equated to PORTB.0. thus whenever we use the word LED in our program it would mean PORTB.0 this makes program easier to understand, and make it more logical.

Arrays

Array is a common structure used in programming. The concept is to use multiple variables, with same name, but having an index number to refer them. Since an index number itself can be a variable, it is easier to walk through a huge array of variables, just by changing the index.

To declare a variable as an array, we have to mention its length.

```
Dim Temp[20] As Byte
Dim x As Byte
For x=0 To 19
Temp[x]=0
Next x
```

In this example a variable named Temp has been declared as an array of 20 variables, each being a byte sized. The index number of these 20 variables will be from 0 to 19. Thus to access first element of array, we will use Temp[0] instead of just Temp. The index number itself can be a variable.

Strings

Strings are a series of alphanumeric data, which not to be used in any mathematical calculation. For example your name, Country, Address are all strings. Strings are nothing but arrays of bytes. However when such arrays are to be used as strings, the last byte of your data should contain a 0.

```
Dim String1[5] As Byte       ' Create a 5 element array
 Dim String2[5] As Byte       ' Create another 5 element array
 Str String1 = "ABCD" , 0     ' Fill the array with ASCII, and NULL terminate it
 Str String2 = "EFGH" , 0     ' Fill the other array with ASCII, and NULL termi-
nate it
 Str String1 = Str String2    ' Copy String2 into String1
 Print Str String1            ' Display the string
```

The use of a prefix Str before array name tells the compiler to deal the array as a string.

Numeric Representation of Numbers

As we have previously discussed, the same number can be represented as decimal, binary or hexadecimal format. However the compiler has to told that the number is decimal, or binary etc. this is done by prefixing the numeric figure with certain symbols.

Binary is prefixed by %. i.e. %0101

Hexadecimal is prefixed by $. i.e. $0A

Character byte is surrounded by quotes. i.e. "a" represents a value of 97

Decimal values need no prefix.

Floating point is created by using a decimal point. i.e. 3.14

Accessing Ports and Registers

Ports as previously mentioned are special internal registers which map bit by bit to the external pins of microcontroller. Ports have been named as PORTA, PORTB, PORTC and so on. All ports are bi-directional, i.e they can be used to read the state of pin or set the state of pin. Most ports are 8 bit, but some

are smaller. Individual ports and its bits can be accessed the same way as variables. They can also be used in mathematical expressions.

In addition to ports there are number of internal registers with specific functions, moreover the functions are allocated to specific bits of these registers. Although these registers have special addresses, in memory and these addresses are used to access them. Basic makes it easier to access these registers by their names. And then treat them just like any other variable.

These names are predefined, and vary according to the microcontroller being used. Your Basic compiler knows which ports and registers are available in the selected microcontroller.

```
PORTA = %01010101           ' Write value to PORTA
VAR1 = WRD * PORTA          ' Multiply variable WRD with the contents of PORTA


TMR0=0          ' setting the Timer 0 to 0
RCSTA.5=1       ' Setting the bit 5 of RCSTA Register high
```

We shall talk about these various registers in appropriate sections. Here I just want to mention that these special function registers can be treated like ordinary variables, to set or read their values and bits.

Decision Making

Most programs require some sort of decision making based upon certain inputs or conditions. The decisions always evaluate as true or false. The program then executes certain groups of instructions in either case. This is achieved by IF Then EndIF Statement.

The general format of IF is to take a comparison, and to execute a batch of instructions if that comparison evaluates to true. The end of IF is marked by End IF statement.

```
Symbol LED PORTC.0
 If x > 10 Then
    High LED
 End If
```

In case the value of x is not greater than 10 the program will jump to statements below End If. In case the value of x is greater than 10, it will make the LED turn ON, and then continue with statements after end If.

Another form of If uses ELSE. This form has two batches of instructions, one which is executed if the comparison evaluates to true, and other is executed if comparison is evaluated to false. In both cases, either of the batch is executed, after that the statements below End If are executed.

```
Symbol LED PORTC.0
 If x > 10 Then
    High LED
 Else
    Low LED
 End If
```

Repetition or Loops

As we have seen that our program executes from top to bottom. However if a set of instructions have to be repeated again and again some how control has to be transferred back to some statement at top. This process is called a loop.

The simplest loop can be constructed by using a label, and then using a Goto Statement to jump to the label.

```
Device=18F452
 Symbol LED = PORTC.0
Again:
 High LED
 DelayMS 500
 Low LED
 DelayMS 500
 GoTo Again
```

In this program we have defined a label, called Again, notice the colon after it. After doing something when

we want to transfer the control back to a statement at some point on top, we issue the "Goto Again" command. The Goto will transfer the control to the label, named Again and program will continue down. This cycle will repeat endlessly. There is no way the program can get out of this loop.

Most of the times we want a controlled loop, in which the group of instructions have to be repeated in such a way that after a given condition if true the loop is terminated. This can be done by combining an If statement. For example we want the LED to blink 10 times and then continue to the rest of program.

```
Device=18F452
 Symbol LED = PORTC.0
 Dim x As Byte
 x=0
 Again:
 If x = 10 Then
    GoTo Cont
 End If
 High LED
 DelayMS 500
 Low LED
 DelayMS 500
 x=x+1
 GoTo Again
 Cont:
 ' rest of program
```

In this program, we have taken a control variable, named x, and set its value to 0. During the loop we increment its value by 1, on each cycle, and test if the value of x has reached 10. when its value has reached 10, we jump to a label outside and below the looping statement to terminate the loop, and continue with rest of the program.

Since this is such a common scenario, Basic has introduced a number of ways to do so, with same ideology, but in more structured and controlled way. One of these is called For … Next loop.

```
Device=18F452
 Symbol LED = PORTC.0
 Dim x As Byte
 For x=0 To 10
     High LED
     DelayMS 500
     Low LED
     DelayMS 500
 Next x
```

Here is the same program, but with For Loop. In a for loop, you give a range to a variable, and the statements mentioned between For and Next statement are repeated, each time the value of control variable is incremented and when the condition has reached the upper limit, control is transferred to line below the Next statement. You can also use the value of control variable within your loop. However you can not change the value by yourself within the loop body. What if we want to increment the value of control variable by 2? Just mention step 2 in for statement.

```
For x=0 To 10 Step 2
```

Similarly if we want to decrement the value, lets say by 1, from 20 to 0 :

```
 For x=20 To 0 Step -1
```

This kind of loop is very commonly used to initialize arrays.

Well the For … Next statement is wonderful, for repeating the instructions a finite number of times. Sometimes it is not known, for how long the instructions have to be repeated. For example we want to repeat certain instructions till a key is pressed. We do not know when a key will be pressed. So the loop must monitor the state of key.

```
Device=18F452
 Symbol LED = PORTC.0
 Symbol SW3 = PORTE.0
 While SW3 <> 0
```

```
    High LED
Wend
Low LED
```

This program uses yet another method of looping, The While and Wend. While checks for a condition if it is true, the body of loop is executed. This cycle is repeated till the while condition evaluates to false. We have tested the press of key with 0, because our PIC Lab-II uses Active low push buttons. This means they give a logical 0, when pressed, and logical 1 when not pressed.

Another method of looping is Repeat … Until. This is similar to While … wend, but the condition is tested after the loop statements are executed. Thus while tests the condition before starting the loop statements, and repeat tests after them.

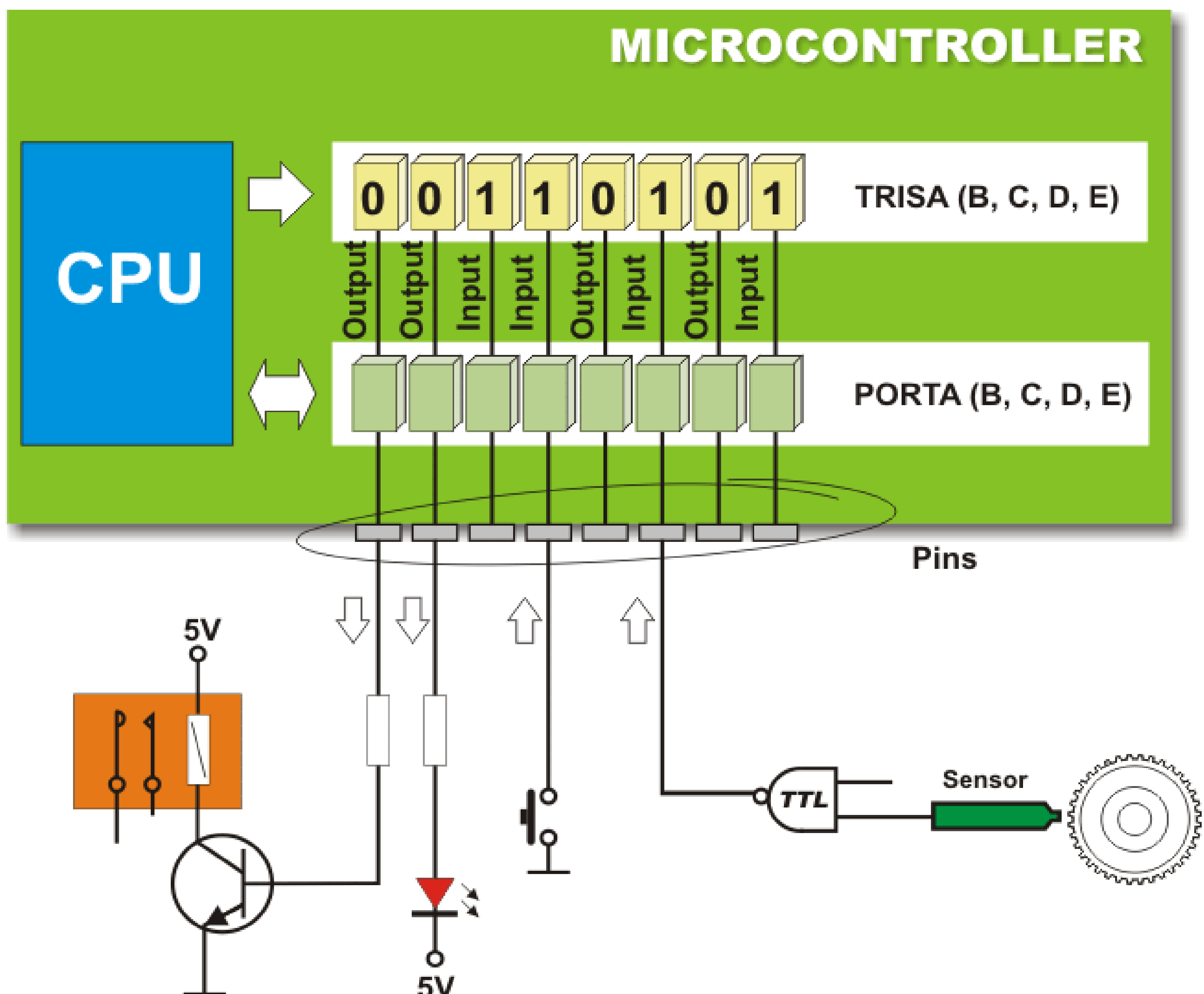
That was an introduction to the Basic language, you will learn more about these rules and commands during the course of this manual. This was however the bare minimum one should know to start with.

Chapter 5
The I/O Ports

Microcontrollers have dual worlds. An internal world, comprising of registers, timers, CPU and other integrated devices, and an external world, which consists of other devices, like LC D, Keypads, speakers, sensors and what not. In order to communicate with these devices microcontroller uses its pins, also called I/O lines. The number of these I/O lines is one of the major characteristics of a microcontroller. The more I/O lines it has, more devices and sensors be connected to it. In Our case, since we are using 18F452 microcontroller, which is 40 pin device, it has one MCLR pin, 4 Power supply and two for oscillator. The rest of 33 I/O lines are available for connection to other devices.

In order pins' operation can match internal 8-bit organization, all of them are, similar to registers, grouped into five so called ports denoted by A, B, C, D and E. All of them have several features in common:

- For practical reasons, many I/O pins have two or three functions. In case any of these alternate functions is currently active, that pin may not simultaneously use as a general purpose input/ output pin.
- Every port has its "satellite", i.e. the corresponding TRIS register: TRISA, TRISB, TRISC etc. which determines performance, but not the contents of the port bits.



By clearing some bit of the TRIS register (bit=0), the corresponding port pin is configured as output. Similarly, by setting some bit of the TRIS register (bit=1), the corresponding port pin is configured as input. This rule is easy to remember 0 = Output, 1 = Input.

Most other programming languages require you to set the appropriate bits of TRIS registers before using the port. Although this is supported in Proton Basic, yet this has a nice simple command, which internally does the same thing.

```
Device=18F452
 Output PORTC
 Input PORTE.0
```

In this program the output command has set the entire portc as output, and the Input command has selected only PORTE.0 as input. This could also have been accomplished using associated TRISC and TRISE registers.

Analog and Digital Pins

As we have seen, that each pin on microcontroller has more than one functions. Although most of the data and communication is in digital format, yet analog features are also very important. A large number of sensors give their output as analog. Thus analog input is essential to work with these devices. PIC18F452 has a number of pins, which can acquire analog data. The same pins however can also be configured as digital, if not to be used as analog.

PORTA

Lets consider the PORTA, which is most commonly used to acquire analog data. By default this port, is configured as analog, when processor is reset. In order to configure entire port, or some of its pins as digital, certain registers have to be set.

Just like TRISA register, which configures the direction of individual pins, there is also an ADCON0 register. This register has three bits which correspond to the 7 analog input channels. Internally there is one Analog to digital converter, so only one channel can be accessed at a time. By changing the number in ADCON0 register all channels are sampled one by one if you want.

In case you do not want to implement analog function at all, you can simply issue:

```
ALL_DIGITAL true
```

This will configure all lines as digital and turn the analog function off. This is implemented in ADCON1 register. If you need a mix of analog and digital pins then you will have to play with this register. In order to use a pin as analog input the corresponding TRIS bit must be set as 1, or issue INPUT command for that port pin, so that it can acquire analog data.

PORTB

PORTB is the second most commonly used port. This is also a bidirectional port, and has an associated TRISB register. The bits of TRISB register corresponding to PORTB bits determine if the port pin will act as input or output. This port does not have analog inputs, however various other functions are associated with individual pins. These functions will be referred in appropriate sections.

	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	Features
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	Features
TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

-	Bit is unimplemented
R/W	Readable/Writable bit
(x)	After reset, bit is unknown
(1)	After reset, bit is set

RB6 & RB7 Pins

These pins deserve a special note. RB6 and RB7 pins of PORTB are also used for programming the microcontroller. While its not a problem when the microcontroller is being programmed in the socket of a stand-alone programmer. However since more and more programmers and specially in case of prototyping where the microcontroller has to be programmed a number of times while testing the software, are using In Circuit programming.

This method uses the microcontroller while it is in the main development board. The RB6 and RB7 then must be free from interference. If certain circuitry or devices have been attached to these pins, which tend to give low resistance and therefore steel the programming signals, it is likely that programming will experience problems. It is therefore advisable, either to keep these pins free, or if used in your design, there should be an intervening 4.7K resistor to the device so that it does not interfere.

PIC Lab-II provides facilities of in circuit programming, and does not use these pins in any circuit. However since this is a general purpose board and you may use these pins in your projects through headers. In that case keep this in mind, while programming. If your device is interfering, either disconnect it while programming of interpose a 4.7-10K resistors in your project.

RB3, LVP

Although most programmer use High Voltage Programming mode, which means the microcontroller needs 12V on MCLR pin to put it into programming mode. However some programmers use Low Voltage Programming. In order to use a low voltage programming mode the RB3 pin must be connected to VDD, or pulled high. PIC-Lab-II allows LVP mode, and RB3 is connected to the programming header as PGM pin. It is the responsibility of programmer to give logical ‘1’ on this pin to use LVP. So keep this fact in mind while using RB3 in your projects, that if your programmer is LVP it will give a logical ‘1’ to this pin while programming.

Microtronics PIC-PG II as well as ICD-2 do not use this pin, so you are free to use it as you like, it will not interfere with these programmers.

RB0 (Interrupt)

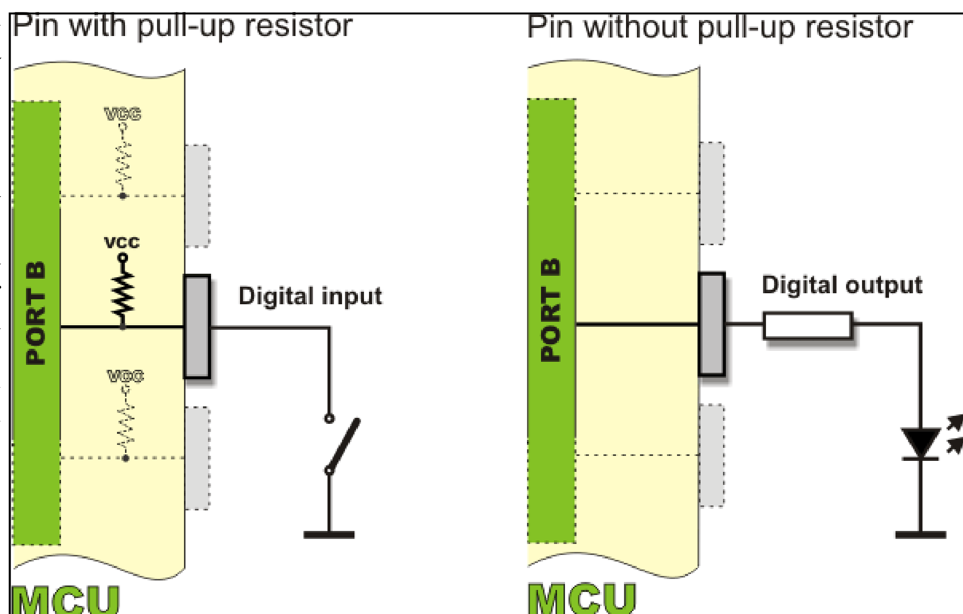
Normally processor is executing one instruction at a time, and while it is executing an instruction it can not monitor another event, like push of a button or coming signals. This problems has been overcome by using interrupt mechanism. We shall talk about this later in appropriate section. RB0, can be configured using internal registers not only to act as input pin, but also to fire an interrupt event whenever its status is changed. To facilitate this type of experiments a push button SW5 has been provided on RB0.

Internal Pull-Up Resistors

Many input devices like switches, keypads etc. require a pull-up resistor, which gives a logical ‘1’ to the pin when there is no ‘0’ from the input device. PORTB has internal pull-up resistors which can be enabled through special function register, or issuing a BASIC command :

```
Declare PORTB_PULLUPS true
```

Using a matrix keypad requires pull-up resistors on columns. If connected to another port, your keypad circuit must have its pull-up resistors. However it can be directly connected to PORTB, by enabling its internal pull-up resistors.



PORTC

PORTC is similar to PORTB, as its also a bi-directional digital port. It has an associated TRISC register which determines the direction of port pins. PORTC has number of additional functions associated with its

PORTC	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	Features
	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

TRISC	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	Features
	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

R/W	Readable/Writable bit
(x)	After reset, bit is unknown
(1)	After reset, bit is set

pins. These functions will be referred to in relevant sections.

PIC Lab-II has 8 LED indicators connected to this port via 220 ohms current limiting resistors. These LEDs are there to show an experiment how to control a pin on and off. In your actual projects you can however connect a small circuitry to turn a relay on or off. The LEDs when turn ON drain a significant amount of current and therefore may interfere with other devices if being used. For example RC6 and RC7 are used for USART communication. This fails if LEDs on PORTC are enabled. You can disable LEDs using SW1 on DIP switch.

PORTD

PORTD and TRISD registers are same as PORTC, other relevant functions will be discussed in appropriate sections. PIC Lab-II uses this port for LCD.

PORTE and TRISE

PORTE is a 4 bit wide port, it is both digital as well as analog. By default these are analog, to use them as digital appropriate register must be set. Or an All_digital True statement used.

Chapter 6
Writing Your First Program

Well with this basic background knowledge, we are now in a position to start experimenting and writing some programs. These programs may not actually do anything practically useful, but they are a good starting point, and a way to explore how to get a job done. In our this section we will be using LEDs on PIC-Lab-II board as output devices to see how our program is going.

It is expected that you have setup your working environment. Proton BASIC has been installed, and ICPROG configured to use PIC18F452. Your programmer and development board are in place. Make sure that after uploading every new program, you disconnect the programmer from PIC-Lab-II motherboard.

"Hello World"

Almost every book on programming has its first program, named 'Hello world' this is usually a way to put a simple text "hello world" on the screen or some output device, in the simplest and easiest way, to show the entire cycle, and to be sure that the whole thing is working.

In our case since we do not have the screen yet (We shall use LCD later), we will use LEDs to show our message. We just want to turn all LED's connected on PORTC to turn ON. All we need to do is to write a logical '1' to all the bits of PORTC. This will turn all the LEDs on. Make sure that the LED enable SW1 on

```
Device=18F452          'Select Microcontroller
 XTAL 20               'Select Clock frequency 20MHz
 ALL_DIGITAL true      'Make All lines digital as we are not going to use Analog
 Output PORTC          'Make All bits of PORTC as output Same as TRISC=0
 PORTC= %11111111      'Assign all bits of PORTC a logical 1
 End                   'Put processor into an endless loop
```

DIP Switch is ON.

Now write this program in PROTON BASIC, and save it into your PIC projects folder as 'Hello.bas' compile the program and upload into your microcontroller. Make sure that the power of PIC Lab-II is ON while programming. After programming turn it off and disconnect the programming cable. Now turn the PIC Lab-II ON. Make sure SW1 is ON all LEDs on PORTC will turn ON, saying "Hello World".

Now lets dissect this program one statement at a time, to know these commands.

```
Device=18F452
```

This line informs the compiler that we will be using 18F452 microcontroller. Since memory mapping, availability of ports, and port bits as well as other on-board devices differ among microcontrollers, it is important to inform the compiler. This is usually the first line in your program. Every compiler, may it be BASIC, C or whatever needs to be told about the device one way or the other. Another important thing is to verify that your particular device is being supported by your compiler. Not every device is supported by these compilers.

```
XTAL 20
```

This statement informs the compiler about the speed of board. So that compiler can effectively calculate the internal logic which implements various kinds of delays and baud rate etc for serial communication.

```
ALL_DIGITAL true
```

As you know there are number of analog inputs on these devices, and we discussed how to enable and disable them. We shall talk more about this issue in appropriate section. In programs which do not need analog input and instead want to use these I/O as digital, this command will disable the analog input on all lines, and make them digital. Although in this program we are not going to use any line, used for analog,

still it is good practice to include this statement.

```
Output PORTC
```

As we have previously discussed, PORTC is bidirectional. This means it can get data from outside as well show data to outside. This is accomplished by accompanying TRISC register. This command effectively uses TRISC internally, to set all bits of PORTC for output. So that any value set to PORTC register by the software, will be immediately reflected on the associated pins. Thus we have configured PORTC as output.

```
PORTC= %11111111
```

Now we are going to assign a value to PORTC register. PORTC is 8 bit register, so we can set its value using any number having a value ranging from 0 to 255. %11111111 is the binary number, setting each bit as logical ‘1’. You could write the same statement as:

```
PORTC= 255
```

255 is decimal number, equivalent to %11111111, or you could use hexadecimal number, like:

```
PORTC= $FF
```

Notice the $ sign before hexadecimal number, and a % sign before binary number. Compilers vary in these signs, beware of your compiler.

```
End
```

As soon as you set the PORTC register with this value the effect is reflected on pins. As we do not want anything else, whereas processor always needs to do something. It can not be made to stop. The End statement actually creates an endless loop, keeping the processor busy, so that it can not take any action till the program is reset. Whatever output has been set on port pins will remain set even on end statement.

Now try changing the value being assigned to PORTC. Assigning 0 will turn all LEDs off, and a % 10101010 will make alternate LEDs on and OFF. I hope this is fairly simple and clear.

Blinking LEDs

In our next project, we are going to make these LEDs blink. Blinking has two phases, an ON and OFF. Both phases have a particular time period, before switching over to the other state. This determines the frequency of blinking. If the ON and OFF times are same, we can call it a square wave. Just keep it in mind

```
Device=18F452        'Select Microcontroller
 XTAL 20              'Select Clock frequency 20MHz
 ALL_DIGITAL true     'Make All lines digital as we are not going to use Analog
 Output PORTC         'Make All bits of PORTC as output Same as TRISC=0
 Loop:
     PORTC= 255           'Assign all bits of portc a logical 1
     DelayMS 1000         'let the LEDs remain ON for 1 second
     PORTC= 0             'Turn LEDs OFF
     DelayMS 1000         'Keep Them OFF for 1 Second
 GoTo Loop            'Go back and repeat the process
```

that it is not always required to blink the LEDs but to send such pulses at a precise rate to another device. In order to see these pulses, we are using LEDs however.

Most of the program is understandable, however lets discuss the Loop statement. We have declared a label called Loop: and a corresponding Goto Loop statement. The instructions between these two statements will be repeated endlessly.

PORTC = 255 turns all LEDs ON. However before turning them OFF, we have to impose a delay.

```
DelayMS 1000
```

This statement gives a delay in millisonds of the number. Thus 1000ms delay is equal to 1 second, similarly 500ms delay is half second, and 100ms is 1/10th of a second. You can also make delays in micro-seconds using delayus statement.

So LEDs will turn on, and processor will, delay for 1 second, then PORTC=0 will turn the LEDs OFF. Again we insert a delay, to keep them OFF for 1 second, before turning them ON. After a delay of 1 second, when LEDs are OFF, the goto command transfers control back to LOOP statement, which is followed by PORTC=255 which will again turn LEDs ON. The cycle of ON / OFF with 1 second intervening delay will continue for ever.

In other words we can say a square wave of 1Hz is being generated on all pins of PORTC.

Compile the program and upload into microcontroller. Run it, and all LEDs should be blinking, precisely

Do it Yourself:
LEDs ON for half second and OFF for 1 Second
LEDs On for 200ms and OFF for 2 Seconds
Create a square wave frequency of 10Hz.
Blink LED Twice, with a delay of 500 ms followed by an Off state of 2 seconds
(See samples on CD)

ON for 1 second and OFF for 1 second.

Now you can experiment in various ways:

So you have seen that we can assign various values to PORTC, which are reflected on LEDs as ON or OFF.

```
'EX1.Bas
'Show binary numbers 0 to 255 on LEDs
 Device=18F452       'Select Microcontroller
 XTAL 20             'Select Clock frequency 20MHz
 ALL_DIGITAL true    'Make All lines digital as we are not going to use Analog
 Output PORTC        'Make All bits of PORTC as output Same as TRISC=0
 Dim x As Byte
 Loop:
 For x=0 To 255      'Start a Controlled Loop of variable x
    PORTC=x          'Assign x to PORTC
    DelayMS 200
 Next x
 GoTo Loop
```

Now lets try showing some binary numbers on LEDs. Here we would take a variable, assign it some value and show it on LEDs. That's fairly simple. What if we want all the values from 0 to 255 to be shown on LEDs, with an intervening delay of 200ms? Certainly its not a good idea to write 255 blocks of code with each containing a different value. Here use of variables and loops comes handy. Lets look at this example:

In this program we have declared a variable named x, which is byte sized variable. Using a For … Next loop the value of x is allowed to vary from 0 to 255. each time, the PORTC is assigned a value of x, which will be immediately shown on LEDs, a 200ms delay is inserted so that we can see the status of LEDs. Next x cause the counter to increment value of x by 1, this whole process is repeated each time with a new value of x, after the value has reached 255, the For loop is terminated and control is transferred to the statement after next x. here goto statement transfers the control back to For loop, which is started again from 0 to 255.

- Start numbers from 65 and end at 190 (Ex2.Bas)
- Start Numbers from 0 to 255 increment by 2 (Ex3.Bas)
- Start numbers from 0 to 255, each time delay is also increased by the same factor (Ex4.Bas)
- Start number from 255 and decrement to 0 (Ex5.Bas)
- Start from 0 to 255, keep all LEDs ON for 2 seconds then decrement from 255 to 0, keep all LEDs OFF for 2 seconds and then repeat the process (Ex6.bas)
- Start from 0 to 255 then Blink LEDs 5 times and then decrement to 0 wait 2 seconds and repeat

So this example shows two things, 1: The utility of loop, to vary the value of a variable and then use the variable within loop. Secondly use of two loops, one (the For … Next) loop is limited to 255 repetitions and the second endless loop to keep this loop, starting again and again after its finished.

Such loops are called 'Nested Loops'. Now try these variations:

In Ex3 notice: `For x=0 To 255 Step 2 the step 2 informs compiler to increment by 2.`

Bit-Wise Management

That's great, you have gone through a number of exercises, to assign various numbers to the port. Up till

now we have been assigning values to the entire port, affecting all 8 bits at a time. In our real world projects entire port is hardly used, indeed individual bits of a port are attached to different devices. For example, if bit 0 of PORTC is connected to a relay controlling Fan, and Bit 1 connected to a relay controlling second fan and bit 3 attached to heater relay. So we would like to control all three bits individually. Although we can do so by assigning properly formatted number to the port, yet it better to control individual bits directly without affecting other bits. This is even more useful, when some of the bits on a port are acting as inputs. An individual bit can be referred to not only on a port, but also on special function registers and even memory variables. This makes programming very easy and manageable.

The bits of port are referred by placing a period after port name and a number to indicate the bit number. For example to refer bit 7 (highest bit) on PORTB, we use PORTB.7, notice the period after port name and

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC.7
loop:
High PORTC.7
DelayMS 500
Low PORTC.7
DelayMS 500
GoTo loop
```

the number 7 indicating bit number. Similarly to access bit 0 of status register for example, we use STATUS.0 similarly to make bit 0 of PORTB as input, using TrisB register, TRISB.0 = 1

In order to set the port bit high or low, you can assign it a value of 1, or 0. You can also use the command, High and Low to do the same. High and Low commands only work on Port register bits and not on general variable bits.

In this program as you can see, we have directly accessed bit 7 of PORTC. Using the high and Low

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC.7
High potc.7
loop:
DelayMS 500
Toggle PORTC.7
GoTo loop
```

commands, the bit is alternating between logical 1 and 0.

Toggle Command

Proton Basic has a nice command called Toggle. This command accepts a port bit, and changes it to opposite level. So if the bit was 1, it is changed to 0 and if it was 0 its changed to 1.

This program essentially produces the same results, however using a toggle command.

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC
PORTC=0
Loop:
High PORTC.0
Dim x As Byte
For x=0 To 7
DelayMS 500
PORTC=PORTC << 1
Next x
DelayMS 500
GoTo Loop
```

Bit Wise Shift Operators

Bits can be shifted within a variable or port to left or right, by as much positions as required. Shifting the bits towards left has a mathematical effect of multiplying by 2, and shifting the bits to right has divide by 2

effect. Shift left is indicated by << operator and shift right by >> operator.

This example turns bit 0 of PORTC high, then after every 500ms shifts the bits left, new bits with logical 0 are placed at the right, least significant bit, when bits are shifted.

Bit Wise Operations

AND, OR, XOR and NOT are the logical gates, however they are also implemented in the same logic within software. Two variables, or numbers can be compared together using these logical operators and the result is used either directly to make a decision, as in IF, or While loops or stored in another variable for use later in program.

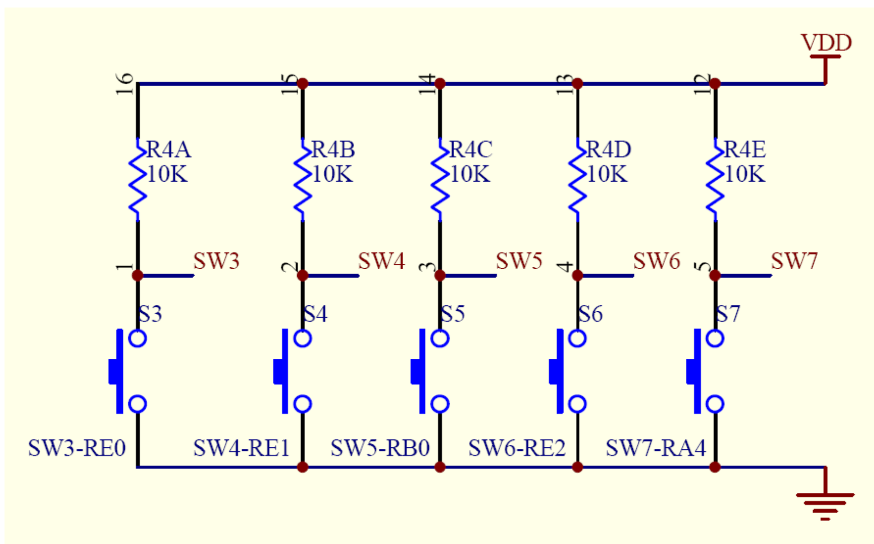
Specific examples will be given in relevant sections.

Chapter 7

Reading Switches

A typical scenario in microcontroller world consists of some input, a processing and some output. Reading input from a number of sources is very important. The input into a system can be analog, which varies with voltage, or digital, which is either true or false. Switches, or other devices can give signals to the microcontroller as logical 0 or 1. Here we shall talk about the general scheme of connecting push switches to various I/O lines of the microcontroller.

Switches can be connected in one of the two ways. Either they are active high, or active low. This means that when a switch is pressed, it connects the I/O line to VCC or 5V line, to give a logical high. And when released the line is left open. This is called active High, the reverse is active low. However in order to prevent an open line, The line is connected using resistors to VCC or GND.



PIC Lab-II has 5 input switches. These switches are configured as active Low. Thus they have associated Pull-Up resistors connected to VCC. The corresponding line will read logical 1, when the switch is NOT being pressed and a logical 0 when it is being pressed.

SW3 to SW7 are DIP switches, through which these lines are connected to the specific I/O lines of microcontroller.

The specified line has to be declared as digital and as input before using the switch.

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC
Input PORTE.0
Symbol LED = PORTC.0
Symbol SW3 = PORTE.0
Low LED
Loop:
If SW3=0 Then
    DelayMS 300
    High LED
End If
GoTo Loop
```

(This panel shows program in two columns only to save space, write the program continuously)

This program loops around, and monitors the PORTE.0 on which SW3 has been attached. When the switch is in open state the I/O line reads it as high or logical 1, as soon as the switch is pressed, and the IF line is executed, it reads the I/O line as 0 and executes the instructions enclosed within the structure of IF … End IF.

Notice the small delay provided, this has been done so, that once the switch press has been detected,

Make a program that should turn the LED ON and OFF on Each press of a button.
Make a program to Turn LED ON when SW3 is pressed and OFF when SW4 is pressed.

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC
Input PORTE.0
Input PORTE.1
Symbol LED = PORTC.0
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.4
Dim x As Byte
x=100
Low LED
Loop:
DelayMS x* 10
Toggle LED
If SW3=0 Then
    x=x+5
    DelayMS 200
EndIf
If SW4=0 Then
    x=x-5
    DelayMS 200
EndIf
GoTo Loop
```

sometime be given to release the switch, otherwise the software will repeat the same steps thousands of time before switch is released.

This program will read two switches, SW3 and SW4, and when detected to be pressed will increment or decrement the value of a variable x by 5. the x in turn is used in the delay. Thus the speed of blinking will be changed by pressing SW3 or SW4.

Debounce

When a button is pressed, the contacts make or break a connection. A short (1 to 20ms) burst of noise occurs as the contacts scrape and bounce against each other. **Button**'s Debounce feature prevents this noise from being interpreted as more than one switch action. The delay after reading the input state, effectively does that.

In real world applications, the buttons are supposed to perform more than just a press. The software is supposed to be smart to behave in a variety of manners to a button press. For example in case of selecting an input value, the button when pressed is supposed to increment the number, however if pressed for a certain period, it starts behaving as if the button is being pressed repeatedly. This is called auto repeat feature. To implement these features is not very easy in software, however Proton BASIC has provided us a very beautiful command called Button. This command will monitor a specific port pin for input, and read

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC
Input PORTE.0
Input PORTE.1
Symbol LED = PORTC.0
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.4
Dim x As Byte
x=0
loop:
Button SW3,0,100,250,x,0,aa
Toggle LED
DelayMS 100
aa:
GoTo loop
```

the value as active low or high, it also accepts a number which is the delay to wait, after which an auto-repeat will be issued. The frequency of auto repeat can also be mentioned in the statement.

This program uses the button command to read the state of a button, SW3. If the button is pressed, it will toggle the LED, if you keep the button pressed, it will start auto repeat after a little delay.

In addition to push buttons, many types of switches, like jumpers or DIP switches can be used in the hardware to indicate various configurations. They can be read just like any switch, as logical 0 or 1, depending upon the hardware setup.

Reading a combination of buttons

Sometimes in a program it is necessary to read a combination of buttons, this is used to expand the range of

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC
Input PORTE.0
Input PORTE.1
Symbol LED = PORTC.0
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.1
loop:
If SW3=0 And SW4=0 Then
    Toggle LED
    DelayMS 300
EndIf
GoTo loop
```

input capabilities, as well as provide a safety mechanism for important functions.

The combination can be read in the same line, using AND operator, or as nested Ifs.

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTC
Input PORTE.0
Input PORTE.1
Symbol LED = PORTC.0
Symbol LED2 = PORTC.7
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.1
loop:
If SW3=0 Then
    High LED2
    DelayMS 2000
    If SW4=0 Then
        DelayMS 200
        Toggle LED
    EndIf
 Low LED2
EndIf
GoTo loop
```

This program will monitor SW3 and SW4 keys. It will toggle the LED only if both keys are pressed.

In a second situation we want SW4 to follow SW3.

This program reads SW3 as primary switch, when pushed, it will turn LED2 (PORTC.7) ON and wait for two seconds, this gives time to press SW4, at the end of 2 seconds it will check if SW4 is still pressed, if yes then it will toggle the LED. Whether or not SW4 is pressed, at lapse of 2 seconds the LED2 will go off, and cycle restart.

This kind of mechanism, ensures that a particular sequence of keys are pressed before an important action, like turning off a motor etc is taken place.

Special Switches on PIC Lab-II

Among the five push switches, which are individually selectable through SW4-SW8 on Dip Switch, there are two switches which need special attention.

SW-5 PORTB.0 Interrupt

SW-5 is a general purpose switch, located on PORTB.0. Although it can work as a normal push switch as described above, the PORTB.0 can be configured to fire an internal interrupt procedure whenever a change, occurs on the pin. We shall talk about this later in relevant section. Here I just want to mention that this switch can be used to not only as general purpose input switch, but also as a means to test the interrupt.

SW-7 PORTA.4 T0CKI

Switch 7, which is connected to PORTA.4 can also be used in another way. PORTA.4 pin can be configured as input for Timer 0. Timer 0 can be configured to count the number of pulses coming on PORTA.4. In order to experiment with this we can use SW7 to give external pulses for this counter.

A special header, is also provided on PIC Lab II, consisting of PORTA.4, and power supply. You can make an external source, like a 555 based oscillator, and connect it to this connector, to act as external events, which can be counted.

Chapter 8

Character LCD

Liquid crystal display, or LCD is a very commonly used device in electronics projects to display data and interact with users. LCDs contain s special crystals, which change their optical characteristics, when electric current is applied to them, this makes them visible, on a contrast background. Liquid crystals do not emit light by themselves, like LEDs. Therefore you need light to see them, usually the surrounding light is enough to read the display, yet in case of dark environments it is hard to read the display. Most LCDs therefore contain an optional backlight, to produce sufficient contrast, which makes reading easy in dark environment.

There are two basic types of LCDs available, Character LCDs and Graphic LCDs. Of these two varieties character LCDs are more commonly used. However there are situations where graphic LCD is more suitable. We will talk about the differences in later sections. Briefly speaking, character LCDs have a predefined set of characters, including numbers, alphabets and special characters like coma, semi-colon etc. the microcontroller, and therefore programmer only needs to send appropriate data to be displayed. Graphic LCDs on the other hand, give you individual pixels, or dots, and the software has to make characters, images etc for display. In routine use, where the device only needs to communicate text data with the user, a character LCD is best suited, whereas an application that is going to show the waveforms of some data, a graphic LCD is better suited. The choice is all yours. Microcontroller has no objections. Even color graphic LCDs are available, which you can use to show true images, and full color display.

Character LCD

Character LCDs are manufactured by a number of manufacturers, in various sizes. Bigger is not always better, chose the one which is most appropriate for the project. The characteristics of LCD are defined by the number of text lines it has, and the number of characters per line. Thus a 20 x 4 character LCD would have four lines of text data, having 20 characters per line. 16 x 2 is the most convenient size and most commonly used in electronics projects. There is absolutely no difference, as far as usage is concerned. So if you learn to use one type of character LCD, you will use another type with same ease.

Second most important thing is the LCD controller. This is a complete microprocessor in itself, which is embedded within the LCD. This microprocessor does everything for us. It accepts data and control commands from the parent application, and manages the display to show it. Different manufacturers can have their own controller circuits, which may vary in protocol of communication. In that case, it is mandatory to read the data-sheet and user manual of the particular LCD before using it.

Hitachi 44780 Controller

Hitachi a popular electronic device manufacturer, came up with a very simple, yet powerful LCD controller called HD44780. This LCD controller is by far now the industry standard controller for character LCDs. The actual manufacturer may be anyone, if the controller is HD44780 compatible, the display will work the same way. Most of the microcontroller compilers contain built-in libraries of code to send appropriate codes to HD44780 compatible devices. So you do not have to worry about the registers, and control codes required. However a general understanding of these under the hood processes is helpful in getting most out of your display. From now onwards, we shall be talking about this standard 44780 based character LCD.

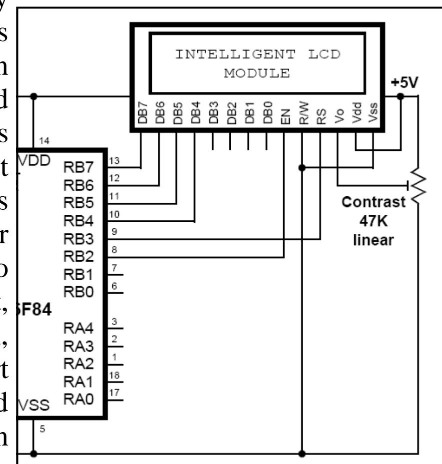
LCD Hardware.

The character LCDs, contain onboard controller, with a connector to communicate with the parent microcontroller. There are usually 14 pins for communication and two pins for a backlight LED, if that is there. Thus a total of 16 pin connector is usually required. This connector can be a single line straight connector, or it can be an IDC like two row 8+8 connector. Which ever is the case, it is important to identify various pins of the connector so that they can be sent appropriate date. All 44780 compliant controllers have following pin definitions.

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
(+)	(-)	D7	D6	D5	D4	D3	D2	D1	D0	E	RW	RS	VEE	VCC	GND

VEE is the contrast adjust volts, to adjust the visibility of characters. RS stands for Register Select pin. RW is for Read / write operation, E is enable, D0 to D7 are eight bits of data communication, (-) and (+) are the Backlight LED connections. The pins will be referred in programs and discussion by these names.

The hardware design, only requires a pot to adjust the contrast. A 50K is enough, connected between VCC and GND. The center tape is connected to VEE Pin. The RW pin selects if we want to read in the contents of LCD display. This is rarely required, so this pin is usually permanently connected to GND, which means a Write mode is selected. D0 to D7 are 8 bits of data. We can operate the display in either 8 bit mode or 4 bit mode. In 8 bit mode all 8 bits are connected to the microcontroller, on a single port. This mode is fast as it sends one byte at a time. However consumes expansive I/O lines. The 4 bit mode connects data pins, D4 to D7 to the microcontroller. This spares the I/O lines. The data is sent in two chunks. You can connect the four bits to either the upper or lower 4 bits of the selected port. Other two control pins, RS and E can be connected to free pins of the same port, or some other port. Every compiler has its own default configuration, however you are not bound to follow it, and you can chose any port and pins you want, the compiler can be instructed to use the specified pins. The configuration shown in figure is the default for Proton compiler.



So we are going to use the LCD display in 4 bits mode, connected to higher 4 bits of PORTB, RS connected to PORTB.3 and E to PORTB.2 if you use a different setting, see your compiler manual for defining your custom settings. In proton Basic However this can be done by issuing various declares, before issuing a print command.

Basics of Character LCD

Whenever you are using a third party, product for interfacing with microcontrollers, you must know clearly, the requirements and communication language used by that system these are usually referenced in the accompanying data sheets. If you do not know, what data to transfer, how can you communicate with that device. This is true not only for LCDs, but for every other device that you are going to use. Certainly you are not going to make everything yourself. Just imagine, that you have a hard-drive available, and you want to use it in your project to store and retrieve information. Now if you do not understand the interface language, you can not communicate with it. This does not mean electronic details, that is how the motors, and heads are connected, but to k=now what sequence of bytes are to be set on the IDE port of hard drive, to turn it on, and then issue another sequence of bytes to read sector no 0, so on and so forth.

Similarly we are going to use character LCD, we should know, that the 14 I/O lines of LCD expect what kind of data and how does it behave to this data. This becomes more important if you have a device, which is not standard.

Using Pre-Built Libraries.

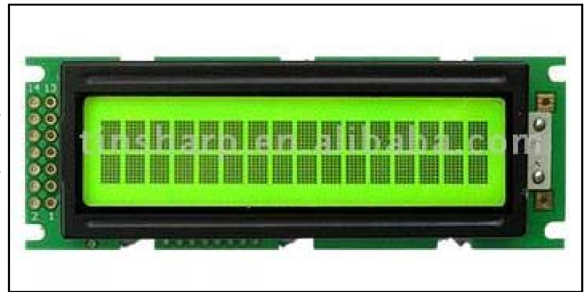
In order to facilitate the job of programmer, companies which make compilers, provide pre-built tested libraries for various commonly used devices. Thus you are given a set of library calls, which you can call from your program, along with various parameters to use those devices. These libraries conceal from you

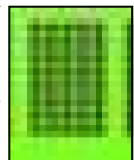
the complexities of low level communication, and instead give you a high level layer of commands to use. Therefore whenever selecting a compiler for your applications, one of the major factors is its pre-built libraries. Nevertheless an understanding of deep underlying process is still helpful if you want to get most out of the device.

Since HD44780 based LCDs are quite commonly used, therefore almost every compiler offers a library of routines to use it. Since library commands, are not part of the BASIC language as such, but they are an extension of services by the manufacturer of compiler, they differ in syntax, and types of arguments and parameter from one compiler to another. So don't get confused if the documentation of your compiler differ from what we are using. Remember all these commands, are going to be translated into the same thing eventually, as far as the microcontroller is concerned.

The basic commands to be sent to a 44780 based LCD are grouped into two categories. Data commands, which will send the actual characters to be displayed, and control commands, which can be used to issue various behaviors built into the display. These behaviors can be for example to clear the display, or to turn the cursor on and off etc. the commands are all sent via various combinations of 1s and 0s on the data, as well as E and RS pins.

I shall not go into the very details of these bit combinations, as we have excellent libraries to do the task. Lets consider a 2 line display with 16 characters per line. Lets have a closer look at an LCD. As you can see there are two lines of liquid crystals. Each line is further composed of 16 boxes, with a small gap between boxes. A closer look at these boxes shows that they are further composed small dots, arranged in the form of a matrix, or an array. This matrix, has dots of liquid crystal, which can be turned ON or OFF. You can not increase or decrease their intensity. They will be just ON or OFF. So this is an 5X8 matrix. All characters to be displayed are mapped within the memory of display controller. All characters which can be shown and their font, is pre-defined. There is however some extra memory available, in which we can define our custom characters. Each character position is mapped to a certain location of memory within the display. In order to correctly position the display data, it must be sent to appropriate memory location. Fortunately this is all taken care of by the compiler library. Although the display has 16 characters per line, the memory inside is actually a 20 characters per line. The extra characters do not show up, but can be scrolled one character at a time to display the entire 20 characters.





As you already know, microcontroller is quite busy in its own processes, and after sending the data for display, it continues its other processes. The LCD is intelligent enough, and once data has been sent for display, it remains there, while microcontroller does other jobs, till new data command is sent by the controller. This is the main reason, why, these LCD modules are so popular. On the other hand, the 7-segment displays, often used in microcontroller applications, require constant contact with the microcontroller to show numbers.

These LCD modules require 5V regulated power supply, and drain a considerable amount of energy, specially if backlight is ON. So your motherboard should be able to handle this heavy current drain. Now let us explore the LCD library with PROTON Basic, to display some data on it.

The first three lines of this program are same as before, the next 4 lines are however new, and important. These lines are defining your hardware setup, of LCD display.

```
LCD_INTERFACE 4
```

This line is actually a declare, or setting and not a command as such. It tells the compiler that our hardware will use 4 bit mode, as shown in the hardware connection diagram above. The other is 8 bit mode, since we want to conserve microcontroller pins, we opted 4 line interface. This is the default mode of PROTON Basic, so if not mentioned it is assumed to 4 line interface.

```
LCD_DTPIN PORTD.4
```

This declaration informs the compiler that our 4 data pins will be connected to PORTD, starting from bit 4. remember either you can use upper 4 or lower 4 bits of microcontroller port. You can not chose for example bit, 2,3,4,5 for data. PORTD.4 therefore indicates to the compiler that the data pins of our display are connected to bits 4,5,6 and 7 of PORTD. If you are using 8 bit mode, then certainly the entire PORT will be

your data PORT, so this statement would become PORTD.0.

```
LCD_ENPIN PORTD.2
```

Have a look at the pin descriptions of LCD, you will find an Enable pin. This pin when set to 1 will allow the display to accept data. This is done automatically by the compiler, so it is mandatory to tell the compiler that where your Enable is connected. In this case it is connected to PORTD.2.

```
LCD_RSPIN PORTD.3
```

The last setting before an LCD can be used is to inform the compiler where, the RS pin of LCD is connected. In this case to PORTD.3. Again this can be any pin on your microcontroller. RS pin of LCD selects the register. It indicates if the data on data pins is control command, or a data to be displayed.

In order to use the LCD, we first have to inform the compiler about our hardware connections. This is done using the three declares. In case you are using some other hardware,

```
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Print Cls
Print "Microtronics"
End
```

make necessary adjustments in the code.

```
Print Cls
Print "Microtronics"
```

These two are the library commands for LCD display. They are not part of BASIC language as such, and different compilers will have different syntax for them. Cls stands for ‘Clear Screen’ this command is usually the first to use in your application. It does two very important jobs. First, when the power is turned on, the LCD electronics, need some time to become stable, this command, inserts an internal little delay, secondly it clears all the registers, and display buffers to 0 and positions the writing cursor to line 1, column 1. after that he display is ready to accept any data.

The **Print** command is very versatile in PROTON BASIC. In its simplest form as shown above, it accepts a parameter or argument, which is a string. Strings are text constants and are always enclosed within inverted commas. The inverted commas, themselves are not part of the string, and therefore are not displayed. The print command will display the text supplied on LCD, starting from the current cursor location, which CLS set to line 1, position 1. The cursor, which is a blinking sign, is itself turned OFF, by default. We can turn it on, as we shall see later.

The end command, as the name indicates, is a BASIC command, that will put the microcontroller into an endless loop. (There is no end or stop for microprocessor, it has to do something all the time).

Notice, after the data has been displayed, and microcontroller is busy in the end statement, your data, on LCD is still there, this is the beauty of 44780 controller. After it has received data, it frees the parent microprocessor to carry on other tasks.

Now how to control the position of displayed text? That is fairly simple, we have two options, one is using **AT** modifier with the print command, and the other is use of **Cursor** command. The AT modifier is most commonly used and is most convenient.

```
Print At 2,10, "OK"
```

This command will first move the cursor to line number 2 and then position 10, and then start displaying the text ‘OK’.

Well so far so good. We have displayed text, as well as control its position, how to display variables, and format them. This is usually the most tricky part of microcontroller programming. The things are complex because some numbers are byte sized, some word sized, some have double precision while still others have a negative sign with them. To complicate the issue further we have floating point variables as well. Now

just consider a byte sized variable, and it has been assigned a numeric value $FF. this variable would have an internal representation of %11111111. Whereas to us it should be displayed as 255 in decimal, FF in hexadecimal. The digits, 2,5,5 and F shown above are characters by themselves having their own codes. So converting a numeric number in digital format into human readable text format is not easy. I shall not go into its details, as this would be beyond the understanding of an average person, I would only say, thanks to the rich library of compiler, that has made this task just a flash.

```
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Dim x As Byte
x=$FF
Print Cls
Print At 1,1, "X:", Dec x
End
```

In this program we have assigned a value of $FF to a byte sized variable x. notice how simple it is to get the value of a variable got displayed on screen

```
Print At 1,1, "X:", Dec x
```

To display a value of a variable, just prefix the variable name with the format modifier, in this case **Dec** this modifier instructs the compiler to display the value of x as a decimal number. Also note we can use as many displays within the same print statement as we want, just separate them with comas, and prefix them with necessary format modifiers. If you want to display the number in Hexadecimal format use **Hex** as prefix.

Number of digits and leading zeros can also be formatted. For example, we want to show numbers from 0 to 255, now some numbers are single digit, some have two digits and still others have 3 digits. We may want that the displayed numbers should be set in three digits, with leading 0s if the number is small. This is simply done by modifying the format modifier. **Dec3** is all that is required. Similarly to show a signed number, like –102 use **SDec** as the modifier. To display a binary number use **Bin** as format modifier. To display floating point variables just use **Dec** Modifier.

```
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Dim x As Float
x=3.1419
Print Cls
Print At 1,1, "X:", DEC2 x
End
```

In the above example x has been declared as Float type variable. So it can be assigned decimal fractions as well. We have assigned it a value of 3.1419. While printing a modifier **Dec2** has been used this will display 2 digits after decimal point, like 3.14 if we omit the 2 from modifier the entire number will be displayed.

In addition to displaying data, there are certain control commands, which affect the behavior of display. These commands do not display anything by themselves. Print command is used to send these controls to the display. These commands are numbers preceded by $FE., `Print $FE,$0F` this command will turn the blinking cursor ON. Blinking cursor is useful when getting input from user, and simultaneously displaying it on display.

Here is a complete program, this program will accept an input from user to select a number. An initial value

```
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.1
Symbol SW5 = PORTB.0
Input SW3
Input SW4
Input SW5
Dim k As Byte
Dim c As Byte
k=10
Loop:
Print Cls
Print At 1,1, "Select A Num:"
' getting Input Value
While SW5 <> 0
```

```
Print At 2,1,"K:", DEC3 k
    If SW3=0 Then
        k=k+1
        DelayMS 200
    EndIf
    If SW4=0 Then
        k=k-1
        DelayMS 200
    EndIf
Wend
```

Custom Characters

The font and character set displayed by character LCD is hard coded and defined within the memory of LCD. This memory is called CGRAM, or Character generator RAM. CGRAM contains characters defined as array of 5x7. Each bit of array, is filled with 0 or 1, which are shown as pixels on display. Not entire ASCII character set is present in this array. For example the character \ is not present. Similarly control characters from 0 to 7 are empty. You can exploit this deficiency, by writing bytes to CGRAM, addresses for these characters and define your own custom characters.

You examine the character set of your display using this program:

```
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.1
Symbol SW5 = PORTB.0
Dim x As Byte
Print Cls
For x=0 To 255
Print At 1,1,Dec x, ":", x
DelayMS 1000
Next
End
```

Notice printing a variable, without prefix modifier, has the effect of showing that character.

```
Print 65
```

Will not display number 65, but display character for ASCII 65, that is ‘A’.

Using a Different Character LCD

The Header for LCD on PIC Lab II has been designed according to HD44780 controller based LCDs. In case you come across a different one, just go through its data sheet, and then send low level commands to its lines. You will not be able to use the library command like Print, because it is written for HD44780 based character LCDs.

Chapter 9
Using Graphic LCD

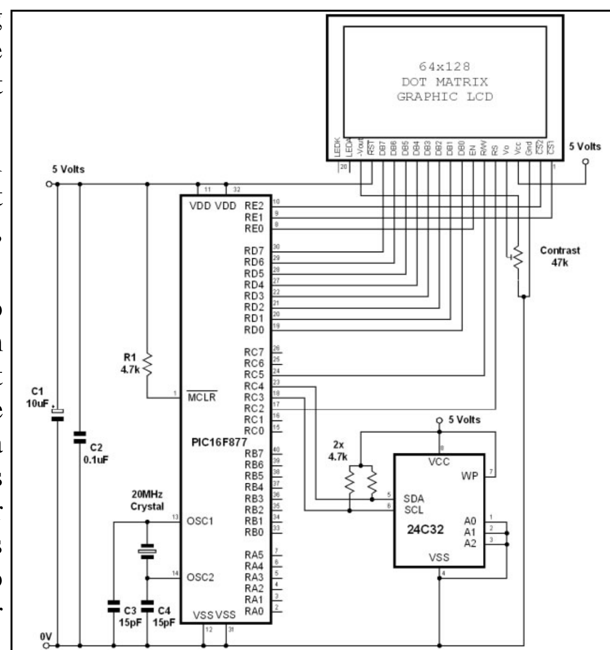
Graphic LCDs are fun to work with. Unlike character LCDs where unit of function is a character, here the unit of function is a pixel, or a dot. The graphic LCDs come in a variety of sizes, their size is described as number of dots in a row, and the number of rows it has. Hobbyists and most electronic projects use medium sized display, usually a 128 x 64 dots.

There are two major classes of Graphic LCDs, one are monochrome, or single color, and other are various forms of color LCDs. Both of these classes are important in their own place. The present chapter however will concentrate on most commonly used graphic LCDs. These are monochrome, with or without backlight.

Graphic LCDs do not have a pre-defined character set, and font table. In order to use these LCDs, you have to define the character set yourself. Commonly the character table is stored in a separate EEPROM, however if the microcontroller being used has enough on-board RAM, it can be stored in it as well. We will talk about EEPROM in later chapters. Here my objective is only to explain that the fonts need to be stored in some memory location before use.

Chip Controller

A large number of manufacturers are making graphic LCDs with their own controllers. However two commonly used controllers are, Samsung S6B0108 or Toshiba T6963. You will need appropriate libraries to address each controller. Proton Basic has built in library that supports Samsung controllers, and therefore our rest of discussion will be centered around using graphic LCD with this or compatible controller. The LCD has two chips, one controlling left and other right half of the LCD.



The LCD works in 8-bit data mode, and therefore will require 8 I/O lines from microcontroller. Apart from that the LCD has, RS (Register Select), R/W (Read / Write), Enable, CS1 and CS2(Chip Select) lines.

Graphic LCDs require negative volts, typically –6V to adjust the contrast. Generating negative volts from standard 5V supply is little trouble some. Most commonly 7660 IC is used to do so. However now some LCDs have the same circuitry built on LCD board, and a contrast out of –10V is available. The contrast –In is connected with –10V contrast through a potentiometer to adjust the brightness of dots. External EEPROM is not as such a part of LCD circuit, however it is better to have it, so that the fonts can be permanently stored for use. Without EEPROM, 18F452 has enough internal RAM to work with.

Declaring LCD Pin Connections

Just like using character LCD, graphic LCD electrical connections with microcontroller need to be defined. This is done using declare statements.

```
LCD_DTPORT = PORTD
LCD_RSPIN = PORTC.0
```

```
LCD_ENPIN = PORTC.2
LCD_RWPIN = PORTC.1
LCD_CS1PIN = PORTE.0
LCD_CS2PIN = PORTE.1
LCD_TYPE = GRAPHIC
```

We are going to connect the LCD data to PORTD, all 7 bits of PORTD will be used. The control pins will be connected to pins on PORTC, and chip select pins, CS1 and 2 on PORTE. The LCD_TYPE must be mentioned as Graphic otherwise by default character LCD is assumed by Proton Basic Compiler. You may chose any other pins as per your project. Once these connections have been defined the PRINT command can be used to send text data, and a bunch of graphic commands to send graphic data to LCD.

Font Declaration

After declaring the electrical connections, we have to inform the compiler about location of fonts table. The fonts table can be located on internal RAM or external EEPROM. Secondly more than one sets of fonts can be defined in EEPROM. The starting address of font table to use also has to be defined.

```
INTERNAL_FONT = On
FONT_ADDR = 0
```

These two declarations define that we are going to use on-chip EEPROM, and Font table will be starting from byte 0 of EEPROM.

The font is described in a specific format like:

```
' Font CDATA table
' Copy and paste this table into your own program
' if an internal font is required.
Font:-  CData $00,$00,$00,$00,$00,$00           'Graphic character 0
        CData $FF,$FF,$FF,$FF,$FF,$FF           'Graphic character 1
        CData $07,$07,$07,$00,$00,$00           'Graphic character 2
        CData $00,$00,$00,$07,$07,$07           'Graphic character 3
        CData $E0,$E0,$E0,$00,$00,$00           'Graphic character 4
```

This must be declared at the bottom of program. Notice the **Font:-** before starting data. The CData defines ready to read data within the program memory, starting from location where its been declared. The data itself consists of 6 bytes, the bit pattern of which defines a complete character to be displayed. Fortunately Proton Basic comes with two predefined font files, located in installation folder. You can include this file in your program, by copying and pasting the entire table, or by copying the file in your project folder and using include statement.

Some displays internally invert the signals of chip1 and chip2, which result in malformed or mal-aligned data. If this is the case with your display also include this declare before using the LCD.

```
Declare GLCD_CS_INVERT true
```

In my case the display I use has CS inverted so I use this statement.

Notice the include statement in the program. The included file name is FONT.INC. This program will print the text on graphic LCD. Notice the print statement is exactly the same. Internally it has dealt with the complexity of graphic LCD.

```
Device = 18F452
XTAL = 20
ALL_DIGITAL=true
LCD_DTPORT = PORTD
LCD_RSPIN = PORTC.0
LCD_ENPIN = PORTC.2
LCD_RWPIN = PORTC.1
LCD_CS1PIN = PORTE.0
LCD_CS2PIN = PORTE.1
LCD_TYPE = GRAPHIC
Declare GLCD_CS_INVERT true
INTERNAL_FONT = On
FONT_ADDR = 0
Print Cls
Print "This is Graphics Test"
End

Include "FONT.INC"
```

Well there are a few things more that you can supply on print command. You can use the AT modifier just like character LCD. And keep it in mind that the numbers after AT are again line and character position not

dots.

```
Print At 2,2,Inverse 1, "This is Graphics"
```

The inverse 1, will display the text in reverse, that is white on black background.

```
Print Cls
Plot 20,50
```

The plot command is graphic library command, it will accept y,x position coordinates and display a dot on location. You can use this command in a variety of ways to control the location of coordinates and display dots. Like making a waveform, from data read on analog port to show its fluctuations.

The opposite of Plot command is Unplot. It clears the pixel at y,x location. The above program, first draws a line, by writing individual dots, and then clears them one by one.

```
Print Cls
```

```
Device = 18F452
XTAL = 20
ALL_DIGITAL=true
LCD_DTPORT = PORTD
LCD_RSPIN = PORTC.0
LCD_ENPIN = PORTC.2
LCD_RWPIN = PORTC.1
LCD_CS1PIN = PORTE.0
LCD_CS2PIN = PORTE.1
LCD_TYPE = GRAPHIC
Declare GLCD_CS_INVERT true
INTERNAL_FONT = On
FONT_ADDR = 0
Dim xpos As Byte
Print Cls
Again:
 For xpos = 0 To 127
 Plot 20 , xpos
 DelayMS 10
 Next
 ' Now erase the line
 For xpos = 0 To 127
 UnPlot 20 , xpos
 DelayMS 10
 Next
 GoTo Again

End
```

```
Line 1,0,0,127,63
```

The line command is used to draw a line between two coordinates. The first argument 1 in this example indicates if line is to be drawn or erased. 1 is to draw. The next two pairs are x1,y1, x2,y2 two sets of coordinates to indicate line end positions.

This example draws a sequence of lines, increment x axis by 2. since lines are calculated mathematically, there is slight ragging in slanting lines. This ragging appears as a pattern on display.

```
Print Cls
```

```
Device = 18F452
XTAL = 20
ALL_DIGITAL=true
LCD_DTPORT = PORTD
LCD_RSPIN = PORTC.0
LCD_ENPIN = PORTC.2
LCD_RWPIN = PORTC.1
LCD_CS1PIN = PORTE.0
LCD_CS2PIN = PORTE.1
LCD_TYPE = GRAPHIC
Declare GLCD_CS_INVERT true
INTERNAL_FONT = On
FONT_ADDR = 0
Dim x As Byte
Print Cls
For x=0 To 127 Step 2
Line 1,x,0,127-x,63
Next x
End
Include "FONT.INC"
```

```
Box 1, 60,30,20
```

The box command will make a box on screen. First parameter is again, weather to show or clear, the next two are x and y coordinates of center of box, and last parameter is the size of box.

Displaying Bitmap Images

Bitmap images are converted into appropriate format for display on graphic LCD first. A number of free to download programs are available on internet. The converted data is then sent to graphic LCD to show the bitmap image. Similarly there are tools available that can produce fonts data for graphical LCD from your system fonts.

Color LCDs and Touch Panels

Apart from the commonly used graphic LCDs, there are color LCDs as well. These LCDs have varying controller on board and you will need to look at the data sheets of your particular display to use it. The color LCDs differ in depth of color. This means the number of colors which can be displayed per pixel. Most common are 65K color LCDs, however true type image quality displays called TFTs are also available with 262K color depth.

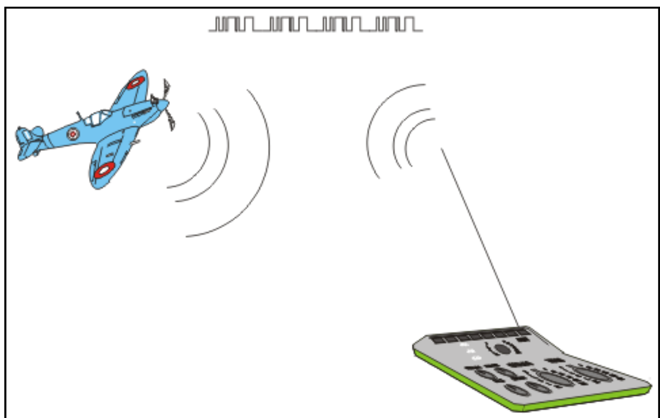
Touch panels are now commonly used in conjunction with graphical LCDs to get user input. Touch panel consists of a separate sheet of transparent membrane, incorporating thousands of touch sensitive points. It is available separately, or sometimes incorporated right over the display. It has its own driver circuit, which needs to be calibrated with the display.

Chapter 10

Asynchronous Serial Communication

Communication with other devices is an important task in the embedded world. Most devices need to communicate with other devices, which may be present on the same board, same device or they may be separate individual devices. The other device may not be based upon the same microcontroller as you are using. Indeed it may be your personal computer, an industrial device which is based upon some other processor. Thus this communication has to be hardware independent. It should not matter, as to what is inside the device, there has to be a protocol for communication.

Ideally a communication system must synchronize its data transmission and receiving with a clock signal. In certain situations like in radio controlled wireless applications it is difficult or sometimes impossible to establish a separate channel for data and clock. In these situations single wire transmission is more effective.



A large number of communication protocols exist, these are implemented one way or other in many devices. Here we shall discuss one of the oldest and time tested serial protocol, called USART. This stands for Universal Asynchronous Receiver and Transmitter protocol. This system uses two I/O lines one fro receiving data and other for transmitting. The data is sent and received without any clock synchronization, therefore its called Asynchronous. The serial port on your PC uses the same protocol to communicate with various devices.

In this chapter we will explore, how to make an effective asynchronous communication system, using our PC as one device and PIC Lab-II as another device. Later we will use two PIC Lab-II boards to establish serial communication.

The USART protocol is very simple, its data consists of either 8 bits or 9 bits, and every start of byte has a start bit and an end bit.

Briefly, each data is transferred in the following way:

- In idle state, data line has high logic level (1).
- Each data transmission starts with START bit which is always a zero (0).
- Each data is 8- or 9-bit wide (LSB bit is first transferred)
- Each data transmission ends with STOP bit which always has logic level which is always a one (1).

The USART can be configured as a full duplex asynchronous system that can communicate with peripheral

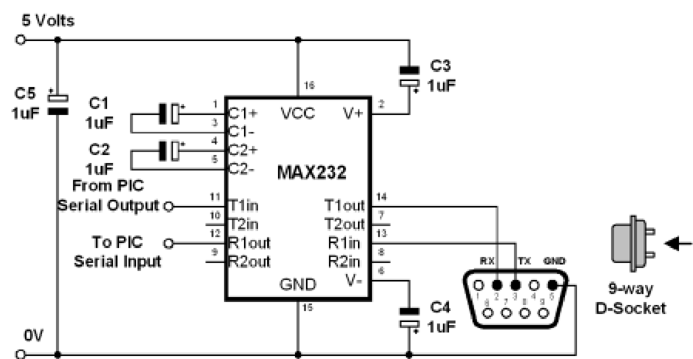
devices, such as CRT terminals and personal computers, or it can be configured as a half-duplex synchronous system that can communicate with peripheral devices, such as A/D or D/A integrated circuits, serial EEPROMs, etc.

As implementation of USART is to follow a protocol only, it can be implemented using software techniques. However this will require quite a detailed code to manage every aspect of transmission and receiving. Since USART is very commonly used in microcontrollers, most of them provide a built in hardware module for that. The hardware module has built-in circuitry to handle most of these tasks automatically, this makes life of software engineer easy, as all you have to do is to send appropriate requests to the hardware module, and the microcontroller will do the rest of job.

The hardware module of PIC18F452 has external pins on RC6 and RC7. Therefore a device which needs to implement the USART using internal hardware, must use these two lines, one for Tx and other for Rx. However using software techniques you can implement connectivity through any digital I/O line.

RS232 Level Conversion

Although devices can communicate directly if their Rx and Tx pins are connected with each other. However to reduce noise interference and increase transmission distance over wire the voltage levels of logical 0 and 1 are changed. This is usually implemented using a TTL level conversion chip, called Max-232. this chip accepts the converted high voltage values and convert them to TTL logical values and present them to microcontroller, it also accepts the logical value from microcontroller and converts it to high voltage levels for transmission.



Before establishing a connection with a device make sure if its using plain USART communication, or using RS-232 level conversion. All PCs having serial ports have this level conversion on them. Therefore in order to communicate with them you must have a RS-232 compliant port. PIC-Lab-II implements this. It has a DB-9 connector for serial cable, the DB-9 connector is connected to Max-232 level converter and outputs of MAX-232 are connected to microcontroller hardware module, pins RC6 and RC7.

Note: Since RC6 and RC7 are also connected to LEDs on board, if LEDs are enabled they tend to interfere with the communication. So make sure that LEDs have been disabled before communicating on USART using RC6 and RC7.

RCSTA and TXSTA registers

Since RC6 and RC7 pins are also general purpose digital I/O lines as well, in order to connect and enable

TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D
bit 7							bit 0

bit 7 **CSRC:** Clock Source Select bit
Asynchronous mode:
Don't care

bit 6 **TX9**: 9-bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN**: Transmit Enable bit
1 = Transmit enabled
0 = Transmit disabled

Note: SREN/CREN overrides TXEN in SYNC mode.

bit 4 **SYNC**: USART Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

bit 3 **Unimplemented:** Read as '0'

bit 2 **BRGH**: High Baud Rate Select bit
Asynchronous mode:
1 = High speed
0 = Low speed
Synchronous mode:
Unused in this mode

bit 1 **TRMT**: Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** 9th bit of Transmit Data
Can be Address/Data bit or a parity bit.

them as USART pins, we have to configure appropriate registers. RC6 is the Tx pin whereas RC7 is Rx pin. The Tx pin will transmit data out of microcontroller and Rx pin will receive data into the microcontroller. The pins will be connected reciprocally to the other device. That is the Tx pin of one device is connected to Rx of other and so on.

RCSTA register configures the receiving characteristics of hardware module whereas TXSTA register will configure the transmitting characteristics of the module.

As you can see various bits assigned various settings to control the behavior of transmission. For our purpose the most important bit to start with is bit 5. the TXEN bit. Setting this bit to 1 will enable serial transmission through Tx pin which is RC6.

```
TXSTA.5=1
```

The same thing can also be declared as:

```
HSERIAL_TXSTA %00100000
```

Baud Rate

Baud Rate is the speed at which data is transmitted. It is represented by a number indicating bits per second. In order to properly communicate it is very important that the communicating devices should have the same Baud rate.

The Baud rate is controlled independently by the Baud rate generator module in the chip. There is a complex calculation to determine exact values to be written in SPBRG register to produce the desired Baud Rate. This process has been simplified by BASIC compiler, using a declare to specify the Baud Rate.

```
HSERIAL_BAUD 9600
```

If this declare is not used the default Baud rate of 2400 is selected. Various commonly used rates are, 2400, 19200,57600, 115200.

Parity Bit

Parity bit is a sort of check to ensure the data sent and received are same. If parity is to be implemented both systems, sender and receiver must implement it. Parity bit is either set to Even or Odd, indicating the number of 1s in sent data. Parity is not commonly implemented.

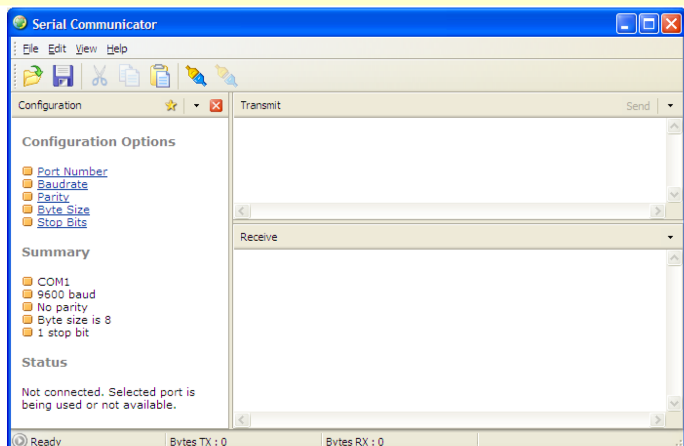
Stop Bit

As previously discussed a stop bit logical ‘1’ is sent to indicate end of a byte.

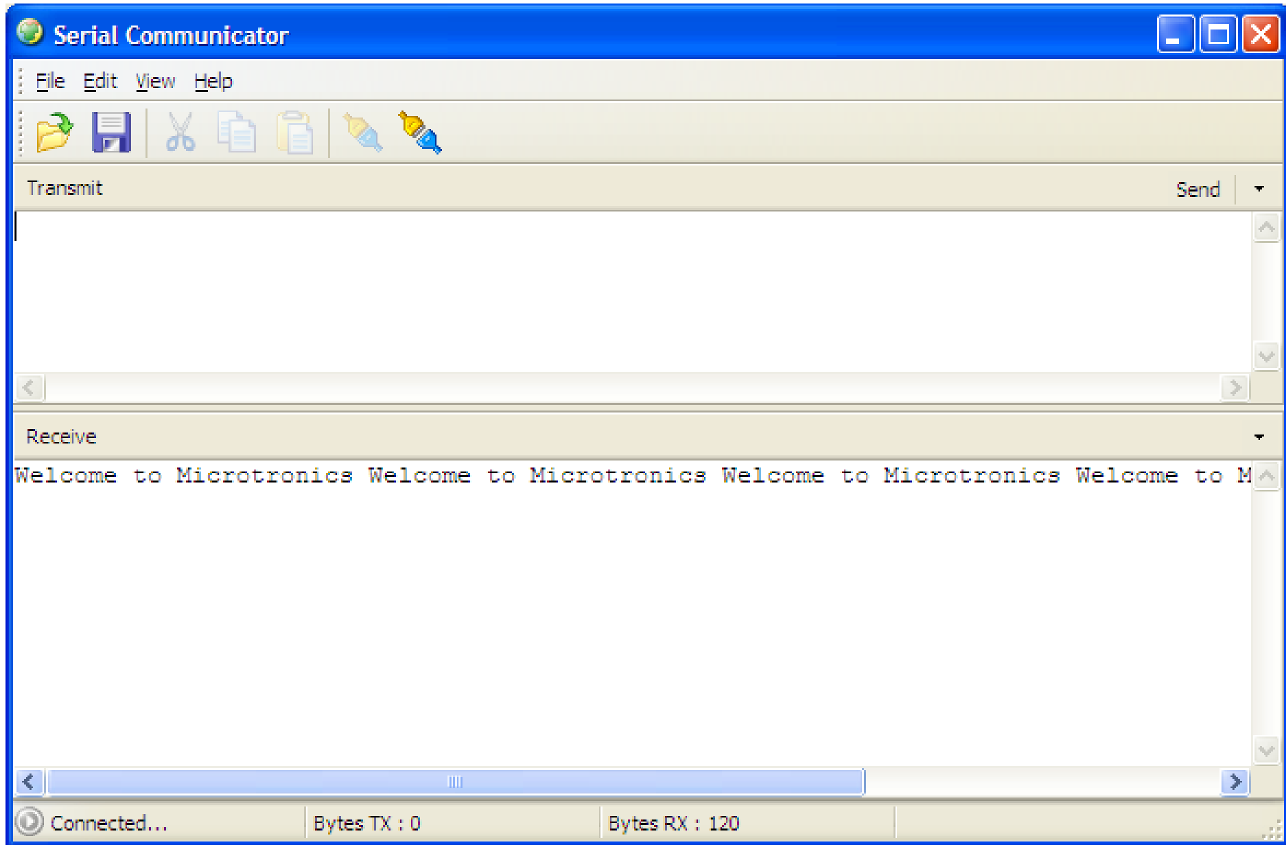
```
Device=18F452
XTAL=20
ALL_DIGITAL true
TXSTA.5=1  ' setting Transmit Enable Bit
HSERIAL_BAUD 9600 ' Setting Baud rate
loop:
HRSOut "Welcome to Microtronics "
DelayMS 1000
GoTo loop
```

The standard protocol is: 9600,N,8,1 this indicates a Baud rate of 9600, No parity, 8 data bits and 1 stop bit.

Now lets write a program to communicate with your PC over the serial port. On PC there must be a program to receive data from the serial port and respond accordingly. We shall use a terminal program, that can send and receive data. Windows has its own Hyper-terminal program in communications section. However Proton Basic provides a simple plug-in called terminal. You can use any one of them.



Lets First Write a program.



Now compile the program and burn it into your microcontroller. In order to communicate with your PC, connect the serial cable, which is connected to your programmer, to the serial connector on PIC Lab-II.

Now within Proton IDE, press F4, or click View, Plug-in, IDE Serial Communicator. This will popup a serial communication window. As shown here. First configure it, select PORT Number to match COM1 or COM2 where your serial cable is connected on PC. Second select Baud Rate, 9600 leave all other settings as such. Now press the Connect icon on communicator, or press F9. The transmit receive panel will open. Now turn your PIC Lab-II ON. This should show the "Welcome to Microtronics" message repeated after every second. If this succeeds, you have correctly made a serial communicating device, which is transmitting serial data to your PC.

Do not proceed till you get this thing. Make sure LEDs of PIC-Lab II are disabled by DIP SW1 OFF.

The key to this program are two declares, or settings, one is Baud rate setting and other is the TXSTA register setting. The other important command is HRSOut. We shall talk about this command little bit here.

HRSOut Command

HRSOut is the BASIC command used to send the data out from microcontroller to a receiving device. Since this communication is asynchronous it does not monitor if the data on other end has been received or not, neither does it verify if data has been correctly received. The **H** in this command indicates the Hardware. This command will use the internal hardware module present in your microcontroller to transmit data. Since the transmitting pin is pre-defined by the hardware, we do not need to mention on which I/O line data is to be transmitted or received.

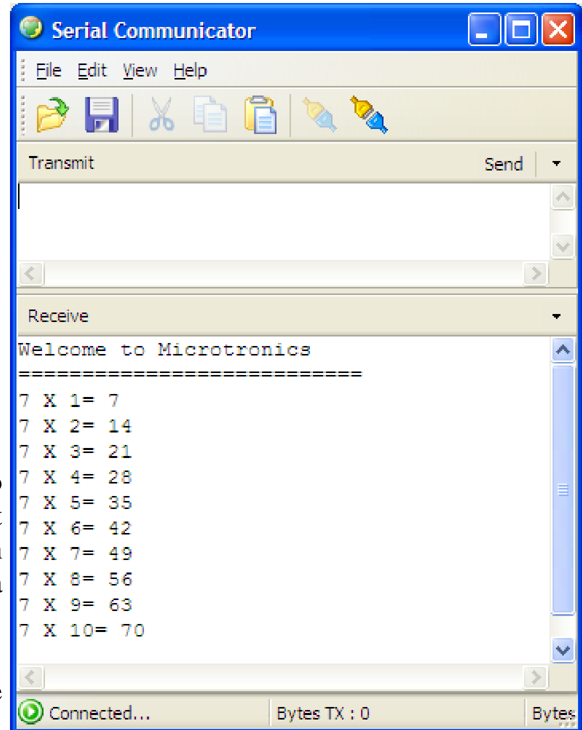
The HRSOut command has a complimentary HRSIn command used to receive data. There is also an RSOut and RSIn commands (without H). These commands implement serial transmission, using software only without involvement of hardware module. They can be configured to use any I/O lines. Thus although there is one hardware USART module in 18F452 and many other microcontrollers, yet we can communicate with many devices at the same time, using software commands.

The HRSOut command has modifiers similar to the PRINT command.

```
HRSOut 65
```

This command will send the decimal number 65, a single byte data. The terminal which can show only text,

```
Device=18F452
XTAL=20
ALL_DIGITAL true
TXSTA.5=1  ' setting Transmit Enable Bit
HSERIAL_BAUD 9600 ' Setting Baud rate
loop:
HRSOut "Welcome to Microtronics ",13
HRSOut "===========================",13
Dim x As Byte
For x=1 To 10
HRSOut "7 X ", Dec x, "= ", Dec x * 7 , 13
Next x
HRSOut "===================="
End
```



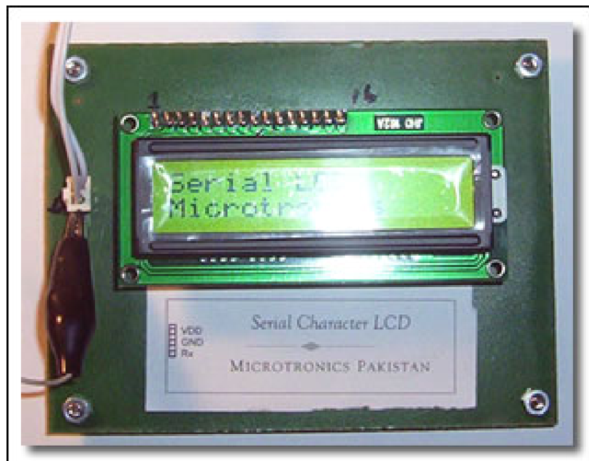
will show 65 as 'A' since 65 is the ASCII code for 'A'. To transmit 65 as a number HRSOut command must format it to send the code of 6 and 5 separately. Although you can do it by sending two ASCII values, yet HRSOut provides a simple method:

```
HRSOut Dec 65
```

The Dec before 65 indicates that the number 65 is to be sent and displayed as 6 and 5.

Now lets write another program that would loop around a variable to vary its value from 0 to 10 and calculate the 7 times table, transmit the data to Computer terminal for display.

Notice the ,13 in HRSOut command. Charcter 13 is new line ASCII code, so it will cause the line break after writing data, next data will be displayed on new line. So after sending the text data HRSOut will send a 13 to format text on terminal. The variable x has been formatted to display its value using Dec modifier.



Many devices an terminals are now available that accept serial data to display.

The benefit of using serial devices is that you can have many devices, attached with a microcontroller with pin conservation.

Microtronics Serial LCD

Here just as a passing reference I am going to introduce serial LCD from us. This device does not have level conversion, therefore you can use it on any pin directly. It has only 1 line input, and two power lines. Using the HRSOut or RSOut command you can send any data on LCD.

Many commercial devices are available that communicate only through serial data.

Like robotic arms, having motors, and sensors, are available, the controller can be any microcontroller, sending and receiving commands.

Serial Modems

A number of serial modems are available, that can accept simple AT (special text commands) commands to establish data connections with commercial gateways and internet. Almost all mobile phones, can be connected using their cable to serial port, and serial commands sent to control them.

Serial GPS Modules

Now even Global Positioning Modules, GPS are available that accept serial commands and return the GPS coordinates in real time the same way.

Thus mastering serial data communication has a long way to get along. Not only you can use serial devices, but you can make your own devices to communicate with others, without caring about hardware details. The LCD module shown above itself contains a microcontroller.

Receiving Serial Data

Well so far we have established a connection with a serial device, our PC, and successfully transmitted data from microcontroller to the PC. Now we are going to explore a little bit about receiving data from external

RCSTA: RECEIVE STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

bit 7 **SPEN:** Serial Port Enable bit
1 = Serial port enabled (configures RX/DT and TX/CK pins as serial port pins)
0 = Serial port disabled

bit 6 **RX9**: 9-bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN**: Single Receive Enable bit
Asynchronous mode:
Don't care
Synchronous mode - Master:
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.
Synchronous mode - Slave:
Don't care

bit 4 **CREN**: Continuous Receive Enable bit
Asynchronous mode:
1 = Enables receiver
0 = Disables receiver
Synchronous mode:
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

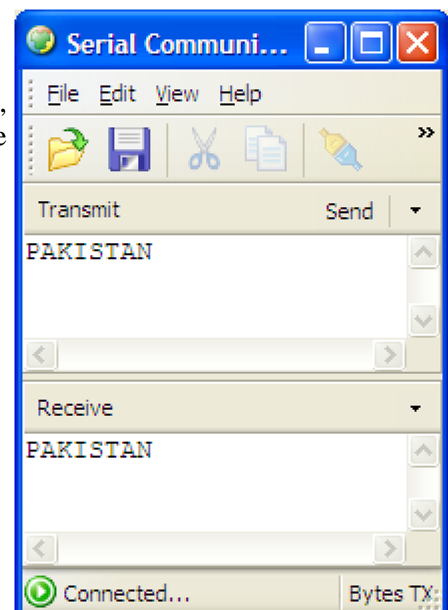
bit 3 **ADDEN**: Address Detect Enable bit
Asynchronous mode 9-bit (RX9 = 1):
1 = Enables address detection, enable interrupt and load of the receive buffer when RSR<8> is set
0 = Disables address detection, all bytes are received, and ninth bit can be used as parity bit

bit 2 **FERR**: Framing Error bit
1 = Framing error (can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

bit 1 **OERR**: Overrun Error bit
1 = Overrun error (can be cleared by clearing bit CREN)
0 = No overrun error

bit 0 **RX9D:** 9th bit of Received Data
This can be Address/Data bit or a parity bit, and

devices. The received data is usually used to control the device behavior or to update the various configurations as well. Most real world serial devices work in Full-Duplex mode, that is they send as well as receive data.



RCSTA Register

Just like TXSTA register which was used to control data transmission, there is a RCSTA register which would control the hardware module to receive data over hardware serial port.

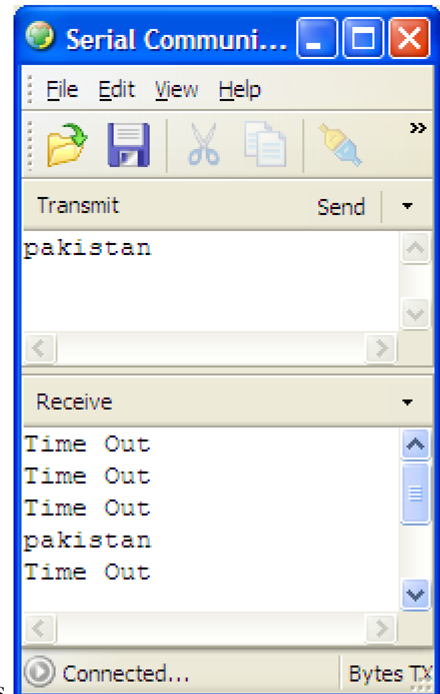
```
Device=18F452
XTAL=20
ALL_DIGITAL true
TXSTA.5=1  ' setting Transmit Enable Bit
RCSTA.7=1
RCSTA.4=1
HSERIAL_BAUD 9600 ' Setting Baud rate
Dim x As Byte
loop:
x=HRSin      'Get a single byte
HRSOut x     'Transmit the same byte back
GoTo loop
End
```

The most important bits in this register to begin with are bit 7 and bit 4. setting bit 7 enables the serial port module, setting bit 4 high enables continuous receive mode.

Write a serial command receiver and sender that should accept one byte and convert it to upper case and return the upper case byte back. Like if we write, Pakistan, it should return PAKISTAN

Now we are going to write a program that will accept a single byte from terminal and return the same byte back, without any processing. This is a sort of echo program, which is used to test the basic two way communication. The serial communicator sends transmit buffer one byte at a time when an Enter is pressed. So when we have pressed enter after writing PAKISTAN in transmit buffer, each byte is sent, starting with P, followed by A and so on. The HRSin command waits for an entire byte to be received, after a complete byte is received it proceeds on.

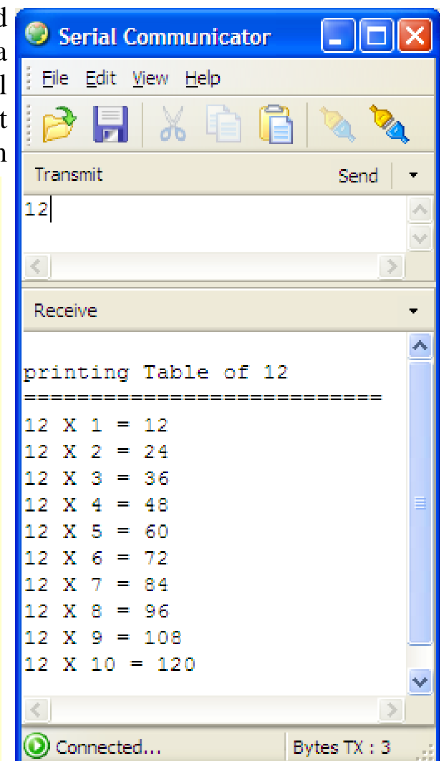
```
Device=18F452
XTAL=20
ALL_DIGITAL true
TXSTA.5=1  ' setting Transmit Enable Bit
RCSTA.7=1
RCSTA.4=1
HSERIAL_BAUD 9600 ' Setting Baud rate
Dim x As Byte
loop:
x=HRSin ,{2000, AA}      'Get a single byte waiting
for 2 seconds
HRSOut x     'Transmit the same byte back
GoTo loop

AA:
HRSOut "Time Out",13
GoTo loop
End
```



x=**HRSin**

This is the simplest method of getting an input from serial device. This command waits forever for a character to receive. Since this may have fatal results in case serial transmission is stopped, the HRSIn command will halt the entire program. A better version of this command accepts a parameter to represent the time in milliseconds for which HRSIn will wait, if within this time data is received it will progress, and if time out occurs, the program jumps to a label from where execution of program

```
Device=18F452
XTAL=20
ALL_DIGITAL true
TXSTA.5=1  ' setting Transmit Enable Bit
RCSTA.7=1
RCSTA.4=1
HSERIAL_BAUD 9600 ' Setting Baud rate
Dim x As Byte
Dim b As Byte
loop:
HRSin Dec x
HRSOut 13,"printing Table of ", Dec x,13
HRSOut "============================",13
For b=1 To 10
HRSOut Dec x, " X ", Dec b, " = ", Dec b * x , 13
Next b
HRSOut "=============="
GoTo loop
```



will continue.

HRSIn Modifiers

In the native form this command will receive only one byte at a time, what if we want to send 123 as a number, if we type 123 and press enter, this will be transmitted as ‘1’, ‘2’, ‘3’ three separate bytes. To solve this issue HRSIn has modifiers that instruct it how to expect data.

The Dec modifier in HRSIn command requires that the variable name is assigned within the HRSIn command. Instead of x=HRSin, we have used HRSIn Dec x.

The Dec modifier forces HRSIn command to seek only numeric characters, as soon as any non numeric character, enter or space is pressed HRSIn will stop collecting characters, and convert the existing characters into decimal form.

The **HRSIn** command can be configured to wait for a specified sequence of characters before it retrieves any additional input. For example, suppose a device attached to the PICmicro is known to send many different sequences of data, but the only data you wish to observe happens to appear right after the unique characters, "XYZ". A modifier named **WAIT** can be used for this purpose:

```
HRSIN WAIT( "XYZ" ) , SERDATA
```

The above code waits for the characters "X", "Y" and "Z" to be received, in that order, then it receives the next data byte and places it into variable SERDATA.

Using Software Based USART

As we have previously discussed we can use any digital I/O line to establish serial communication. Using a level converter with it is a matter of your own option, depending if the other device is using level conversion.

First we will use Microtronics serial LCD, to be attached to a general purpose I/O line and transmit serial data to it to show some text. This device does not require level conversion, therefore we can use it directly.

Note RSOUT_PIN declaration. This declaration indicates the software library as to which I/O line will be used for data transmission. The serial LCD is configured to work on Baud rate of 9600 and displays a short welcome message for 2 seconds, therefore a delay of 5 seconds has been placed before sending actual data.

Although you can connect the display board with PIC Lab II in any way you like, however there are num-

```
Device=18F452
XTAL=20
ALL_DIGITAL true
RSOUT_PIN PORTA.4
SERIAL_BAUD 9600
Output PORTA.4
DelayMS 5000
RSOut Cls
loop:
RSOut At 1,1,"Microtronics"
DelayMS 2000
GoTo loop
End
```

ber of connectors available for convenient connections. Notice a small TOCKI connector on left corner of board. This connector has three pins, Left most pin is connected to RA4, second is Ground and Third is VDD or 5V. This connector easily connects to the display, giving one line of data as well as supply.

Inter device Communication

In our previous example we have connected our PIC Lab-II board with an external device, a serial LCD. Now we are going to connect two PIC Lab II boards to communicate together.

In our first example we will connect the TOCKI connector, of two boards together using a 3+3 cable. So that GND is connected to GND, VDD to VDD and RA4 to RA4. Now one of our boards will act as transmitter to send some data and second board will receive the data and show it on LEDs.

First the transmitter board program. The program simply configures RA4 as output device and uses software library to send data. It sends numbers 0 to 255 every 200ms to the receiver board.

The receiver board receives data on its RA4 and displays the number received as binary on its LEDs. Since USART is not being used (RC6 and RC7) we can display data on LEDs.

```
'Transmitting device
Device=18F452
XTAL=20
ALL_DIGITAL true
RSOUT_PIN PORTA.4
SERIAL_BAUD 9600
Output PORTA.4
DelayMS 500
Dim x As Byte
loop:
For x=0 To 255
RSOut x
DelayMS 200
Next x
GoTo loop
```

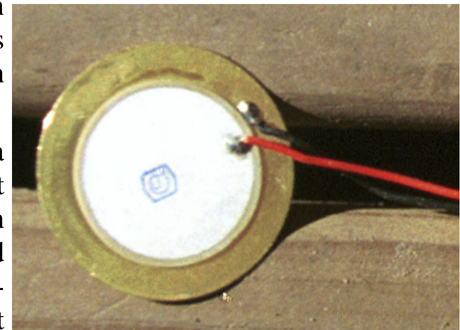
```
'Receiving Device
Device=18F452
XTAL=20
ALL_DIGITAL true
RSIN_PIN PORTA.4
SERIAL_BAUD 9600
Input PORTA.4
Output PORTC
PORTC=0
DelayMS 500
Dim x As Byte
loop:
x=RSIn
PORTC = x
GoTo loop
```

Chapter 11
Sound and Digital Signals

Sound is a complex data, it not only consists of a vibrating disc, but a number of other parameters affect the sound quality. Although dedicated chips are available for producing professional quality sound frequencies, yet PIC microcontroller can be used to produce simple sounds. The sounds produced by PIC microcontrollers are not meant for use in music production, but used to provide audible alerts, and signals like press of a button, completion of a process or to produce an alarm. The sound is an analog data, and PIC microcontroller pins are basically digital I/O lines. This means that a PIC pin can only be switched between 0 and 5V. Not in between. The sound produced therefore can be of square wave quality only.

PIZO Sounder

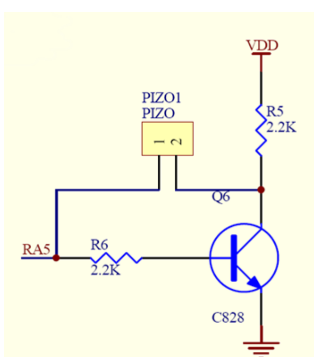
Pizo is a ceramic, having mechanical properties in response to electric current. The pizo crystal tends to expand when an electrical current is applied on its two ends. When current is removed it comes back to its original size. The reverse is also true, that is when it is mechanically deformed to produces an electrical current.



Unlike a speaker, the pizo sounder is light weight, does not have a coil and magnet and does not consume too much electrical power. It is therefore ideal device for use in electronic devices where very high quality sound is not required. The pizo has therefore two wires, and these wires must be given an oscillating signal, to make the diaphragm move too and fro. If current is constantly applied it does not produce sound. Thus some degree of programming is required to produce sound, by making an I/O line rapidly change its state between '0' and '1'. This however gives programmer a liberty to manipulate the diaphragm oscillation by changing the frequency and modifying the ratio between ON and OFF times.

In some applications only an audible beep is required and manipulation of sound frequency is not important. In these applications a device called Pizo Buzzer is used. The pizo buzzer contains a pizo sounder along with a small circuit to give a fixed frequency of say 1KHz sound, when power is applied. Still by changing the power on and off, you can manipulate the tones to some extent.

Connecting a pizo to the microcontroller is fairly simple. It can be connected directly to an I/O line and other part to either ground or +5V. It is however customary to place an intervening transistor, so that more powerful movements be produced, or if you are using a buzzer, the 5V or ground supply can be given through the transistor.



PIC Lab-II contains a transistor driver circuit connected to PORTA.5. the transistor is connected to RA5 via a 2.2K resistor. Remember RA5 is also analog input, available through PORTA header. An analog or any other digital signal on RA5 will also be transmitted to the transistor. The Pizo is not hard wired on board, but a connector is available, and pizo can be separately plugged onto this when required.

A pizo buzzer or sounder is supplied with PIC Lab-II. To experiment with it just

plug in the pizo and write programs to manipulate PORTA.5. Always make sure that you have made this pin digital before using as sound producing pin.

This program simply produces ON and OFF states on RA5 pin. The pin has already been declared as digital output. A delay of 1ms would produce a roughly 1KHz pulse train, resulting in audible sound. Try changing the delay periods, even using DelayUs (micro second delay) to produce various frequencies.

```
'sound
Device=18F452
XTAL = 20
ALL_DIGITAL true
Output PORTA.5
Symbol Pizo = PORTA.5
loop:
Low Pizo
DelayMS 1
High Pizo
DelayMS 1
GoTo loop
```

With the excellent I/O characteristics of the PICmicro, a speaker can be driven through a capacitor directly from the pin of the PICmicro. The value of the capacitor should be determined based on the frequencies of interest and the speaker load. Piezo speakers can be driven directly.

Proton Basic has simplified our task by providing a SOUND command. This commands accepts a port pin as a parameter, and the sound to be produced as a pair of parameters indicating the note and its duration.

The general format of this command is:

```
SOUND Pin, [ Note,Duration {,Note,Duration...} ]
```

You can repeat as many pairs of note and duration as you want to produce sequence of sounds.

The specified pin is automatically declared as output if not already done so. The note is a number from 0 to 255, it can be a constant or byte sized variable. Number 0 is silence, 1-127 are notes of increasing frequency, 128-255 are Noises called (white noise). Noise is sometimes useful, when used to produce a crash or blast in games etc.

```
'sound
Device=18F452
XTAL = 20
ALL_DIGITAL true
Symbol Pizo = PORTA.5
loop:
Sound Pizo, [100,20]
DelayMS 1000
GoTo loop
```

The note of number 1 is approximately 78 Hz and 127 is 10KHz.

```
'sound
' Star Trek The Next Generation...Theme and ship take-off
 Device 18F452
 XTAL = 20
 ALL_DIGITAL true
 Dim LOOP As Byte
 Symbol PIN = PORTA.5
THEME:
 Sound PIN, [50,60,70,20,85,120,83,40,70,20,50,20,70,20,90,120,90,20,98,160]
 DelayMS 500
 For LOOP = 128 To 255                         ' Ascending white noises
 Sound PIN, [LOOP,2]                           ' For warp drive sound
 Next
 Sound PIN, [43,80,63,20,77,20,71,80,51,20,90,20,85,140,77,20,80,20,85,20,_
                     90,20,80,20,85,60,90,60,92,60,87,_
                     60,96,70,0,10,96,10,0,10,96,10,0,_
                     10,96,30,0,10,92,30,0,10,87,30,0,_
                     10,96,40,0,20,63,10,0,10,63,10,0,_
                     10,63,10,0,10,63,20]
 DelayMS 10000
 GoTo THEME
```

The note and its duration (in milliseconds) is enclosed within square brackets. More notes can be mentioned within the same command.

Sound PIN,

Notice an underscore _ at the end of certain lines. Basic language expects the command to be completed in one line, although a long line can be written in the editor, but it will need to be scrolled to examine it. An underscore at the end of line tells the compiler that the command is not finished ant statements on next line should be considered as part of this line.

```
' Play 5 tunes

Device=18F452
XTAL=20
ALL_DIGITAL true
Symbol PIN  =PORTA.5

            Symbol  R    =      0
            Symbol  C    =      82
            Symbol  _DB  =      85
            Symbol  D    =      87
            Symbol  Eb   =      89
            Symbol  E    =      92
            Symbol  F    =      94
            Symbol  Gb   =      95
            Symbol  G    =      97
            Symbol  Ab1  =      99
            Symbol  A1   =      73
            Symbol  Bb1  =      76
            Symbol  BE1  =      79
            Symbol  C1   =      82
            Symbol  _DB1=       85
            Symbol  D1   =      87
            Symbol  Eb1  =      89
            Symbol  E1   =      92
            Symbol  F1   =      94
            Symbol  Gb1  =      95
            Symbol  G1   =      97
            Symbol      Ab2    =      99
            Symbol      A2     =      101
            Symbol      Bb2    =      102
            Symbol      BE2    =      104
            Symbol      C2     =      105
            Symbol      _DB2   =      106
            Symbol      D2     =      108
            Symbol      E2     =      110
            Symbol      F2     =      111
            Symbol      Gb2    =      112
            Symbol      G2     =      113
            Symbol      Bb3    =      115
            Symbol      Bm3    =      116
            Symbol      C3     =      117
            Symbol      D3     =      118

START:

Song1:      Sound PIN,
[G,80,D2,80,C2,20,BE2,20,A2,20,G2,80,D2,80,C2,20,BE2,20,A2,20,G2,80,D2,80,C2,20
,BE2,20,C2,20,A2,80]
            DelayMS 2000
Song2:      Sound PIN,
[F,80,R,2,F,70,R,2,F,10,R,2,F,80,Ab1,60,R,2,G,10,R,2,G,60,R,5,F,10,R,2,F,50,R,2
,E,20,R,1,F,40]
```

```
            DelayMS 2000
Song3:      Sound PIN,
[F2,40,R,2,C2,20,R,2,C2,20,R,5,D2,50,R,3,C2,30,R,40,E2,40,F2,50]
            DelayMS 2000
Song4:      Sound PIN,[_DB2,20,Gb2,20,Bb3,15,C3,30,R,5,Bb3,20,C3,75]
            DelayMS 2000
Song5:      Sound PIN,
[C2,30,R,10,C2,30,R,10,C2,80,R,3,C2,20,BE2,30,A2,20,BE2,30,C2,20,D2,30,R,5,C2,1
0,E2,30,R,15,E2,30,R,15,E2,80]
            DelayMS 2000
GoTo START
```

[50,60,70,20,85,120,83,40,70,20,50,20,70,20,90,120,90,20,98,160]

Produces a sequence of sounds giving an overall impression of melody.

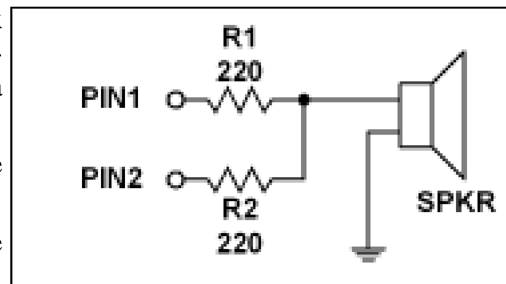
This program shows how you can make various kinds of sounds using the sound command. A more elegant example from Proton Basic samples is given here, this example will produce beautiful tones. The sample has been little modified to suite our PIC Lab-II hardware.

This code defines the notes of various musical symbols , so that in the sound statement you can easily mention the note instead of its number. As a task of exercise you can add more control over this.

- Allow the Push buttons on your board to select the song to be played.
- Include LCD display, to help make selection and then while playing the song its information is displayed on LCD.

Producing More Complex Sounds

Well sound command has been excellent tool to produce general purpose sounds. However real world sound contains a mix of many frequencies being played simultaneously. To give a better quality sound Proton Basic allows mix of sound frequencies from two pins. This feature is not implemented directly on PIC Lab-II however using I/O headers and a small breadboard you can easily implement this.



Pin1 and Pin2 are any two digital I/O lines. The speaker can be any speaker or Pizo Sounder.

To implement this algorithm of mixing two frequencies we have a new command in Proton Basic called, SOUND2.

The general syntax of sound2 command is:

SOUND2 *Pin1*, *Pin2*, [*Note1\Note2\Duration* {*,Note1,Note2\Duration...*}]

Pin1 and Pin2 are I/O lines to be used, the Notes are numbers indicating frequency to be produced. This number can range from 0 to 16000. indicating a frequency of 0 (silence to 16K) the duration is a number ranging from 0 to 65535 the numbers increment in 1ms. The triplet Note1 /Note2/Duration can be reated in the command to produce a sequence of notes. This produces more realistic sounds.

```
' Generate a 2500Hz tone and a 3500Hz tone for 1 second.
 ' The 2500Hz note is played from the first pin specified (PORTB.0),
 ' and the 3500Hz note is played from the second pin specified (PORTB.1).
 Device = 18F452
 XTAL = 20
 ALL_DIGITAL=true
 Symbol PIN1 = PORTB.0
 Symbol PIN2 = PORTB.1
 Loop:
 Sound2 PIN1 , PIN2 , [2500 \ 3500 \ 1000]
 DelayMS 1000
 GoTo Loop
```

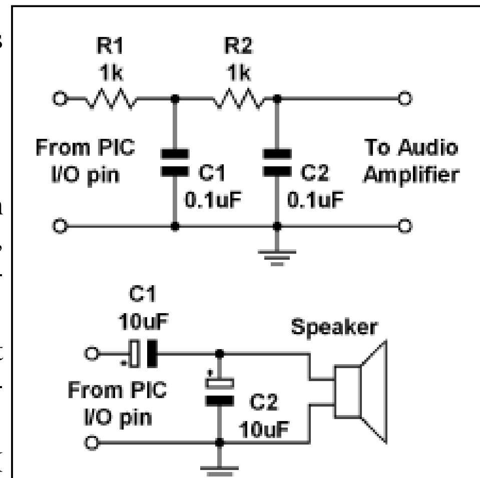
```
SOUND2 PIN1 , PIN2 , [2500 \ 3500 \ 1000 ,  2500 \ 3500 \ 2000 ]
```

The output can be fed through a capacitor to an amplifier or directly to a speaker. The wave quality is not SINE however.

SINE Wave Output

Professional sound amplifiers work on analog electronics where the sound produced from microphone is a sine wave. SINE wave production is a difficult task in digital electronics. The output of SINE wave does not reach its top all at once but slowly, usually following a SIN curve. However thanks to Proton BASIC and the PWM (talked later) characteristics of microcontrollers that this can be done.

The SIN wave is not only required for sounds, but think of making a DC to AC inverter, you can get SIN wave quality controlled frequency with this technique.



The command offered by Proton Basic for this purpose is FREQOUT.

The FREQOUT command has a general syntax of:

```
FREQOUT Pin , Period , Freq1 { , Freq2}
```

FreqOut generates one or two sine waves using a pulse-width modulation algorithm. **FreqOut** will work with a 4MHz crystal, however, it is best used with higher frequency crystals, and operates best with a 20MHz type (PIC Lab-II has 20MHz).

The raw output from **FreqOut** requires filtering, to eliminate most of the switching noise. The circuits shown below will filter the signal in order to play the tones through a speaker or audio amplifier.

The two circuits work by filtering out the high-frequency PWM used to generate the sine waves. **FreqOut** works over a very wide range of frequencies (0 to 32767KHz) so at the upper end of its range, the PWM filters will also filter out most of the desired frequency. You may need to reduce the values of the parallel capacitors shown in the circuit, or to create an active filter for your application.

```
' Generate a 2500Hz (2.5KHz) tone for 1 second (1000 ms) on bit 0 of PORTA.
FreqOut PORTA.0 , 1000 , 2500

' Play two tones at once for 1000ms. One at 2.5KHz, the other at 3KHz.
FreqOut PORTA.0 , 1000 , 2500 , 30000
```

DTMF Touch Tone Sequence

DTMF tones are a complete science in themselves. You should consult internet for more information on these tones. Briefly speaking these tones are actually special sequence of sounds, that are used to convey a message to the remote device. Digital telephones use these tones to dial a number. They send these DTMF tones over the copper wire to the exchange to indicate the desired number. The DTMF tones are also use to radio control remote devices. As such DTMF is not a property of microcontroller, if the algorithm is known any microcontroller can be used to generate these tones. Thanks again to proton Basic, which has a built-in command to do so.

DTMFOUT is the command to produce the tone. The syntax of DTMFOut command is:

```
DTMFOUT Pin , { OnTime } , { OffTime, } [ Tone {, Tone…} ]
```

The OnTime and OffTime are optional values to control the shape of pulse, if omitted then default values of 200ms for On and 50ms for Off are used. The tone, is a number ranging from 0 to 15. 0 to 11 are the standard tones for the telephone keypad, and 12 to 15 are the optional fourth column keys on commercial or test equipment keypads.

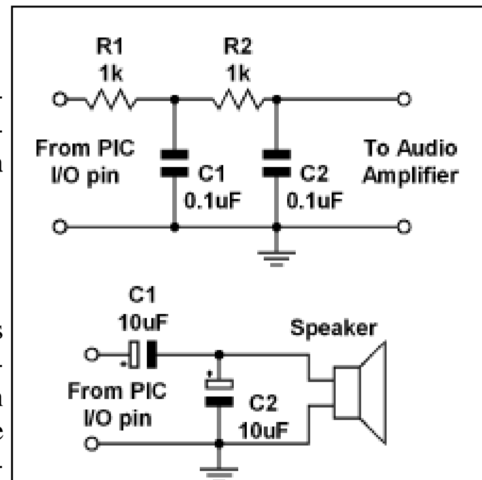
```
DTMFOut PORTA.0 , [ 0,4,2,4,2,7,0,4,5,3 ]
```

This command is going to call Microtronics Pakistan (0424270453) using PORTA.0. you will have to connect PORTA.0 of microcontroller to the telephone line (PTCL Land Line) using a simple circuit.

```
'Set the OnTime to 500ms and OffTime to 100ms
 DTMFOut PORTA.0 , 500 , 100 , [ 0,4,2,4,2,7,0,4,5,3]
```

Using the On time and Off time, this command will call Microtronics Pakistan, but dialing will be slow.

Caution: *Connecting anything to telephone lines is not recommended by PTCL. I do not condone unauthorized telephone connections, and will not be held responsible for any aforementioned authorised or unauthorized connections.*



Use the shown circuit to connect the I/O line to telephone line.

The PICmicro is a digital device, however, DTMF tones are analogue waveforms, consisting of a mixture of two sine waves at different audio frequencies. **So how can a digital device generate an analogue output?** The PICmicro creates and mixes two sine waves mathematically, then uses the resulting stream of numbers to control the duty cycle of an extremely fast pulse-width modulation (PWM) routine. Therefore, what's actually being produced from the I/O pin is a rapid stream of pulses. The purpose of the filtering arrangements illustrated above is to smooth out the high-frequency PWM, leaving behind only the lower frequency audio.

Make a complete dialer, using LCD, to get a number using push buttons. Then using one of the push buttons to dial the number.

You should keep this in mind if you wish to interface the PICmicro's DTMF output to radios and other equipment that could be adversely affected by the presence of high-frequency noise on the input. Make sure to filter the DTMF output scrupulously.

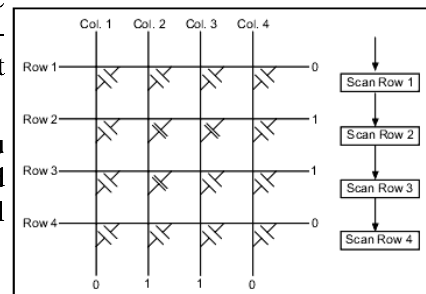
Chapter 12
Using Matrix Keypad

Keypad is a commonly used device to get user input. Although simple push switches can be used to get user input, as we have done so, this would require 1 I/O line per switch. The keypads use a slightly different methodology. Keypads are collection of push switches however arranged in the form of a matrix. So there are rows and columns of switches. The two connections of a switch are also connected in the matrix, so that the row has common connection and column has a common connection. Thus when a button is pressed a row and a column, where the button is pressed gets connected internally. The keypads are usually available as telephone type 3 x4 keypad. This one has three columns and 4 rows, or a 4 x 4 keypad having 4 rows and 4 columns.

The output available from keypad is arranged as rows and columns. Now to detect which key is being pressed is little tricky. Let us say when key 8 is pressed the Col2 and Row3 are connected together.

Connecting the keypad

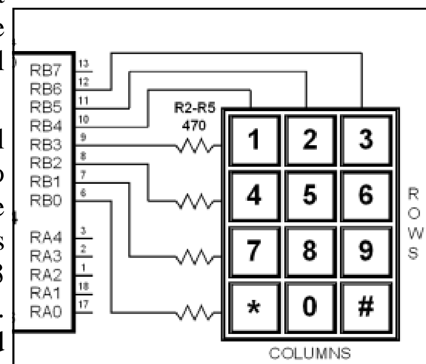
First you identify the various pins of keypad as to which are rows and columns. They are usually grouped together. In order to connect the keypad to microcontroller you need to pull the columns pins high with 10K pull-up resistors. The rows can be connected directly or preferably through 470 ohms current limiting resistors, as when a switch is pressed, the row pin and column pin will get short.



Since PORTB of PIC microcontroller has internal pull-up resistors, you do not need these if this PORT is used. If another PORT is to be used you will have to use Pull-Up resistors. PORTB is therefore the natural choice. All rows and columns must be connected to the same port.

Detecting Key Press

Now when the keypad has been connected it is important to detect which key is being pressed. The trick lies in scanning all the rows one by one. Since columns have pull-up resistors these pins are all at logical 1.



So the first step is to make the row 1 line low, logical 0. then to scan all the column lines for a logical 0. if all the column lines are high the no key in this row is being pressed. Lets say key 2 was being pressed, the column 2 pin of microcontroller would go low and other column pins would remain high. The same process is repeated for row2 and row3 and row4. every time one row is taken low and all columns are scanned. The key being pressed depends upon the column which gets low, and the row being scanned.

The process is simple, but requires quite a big code. This is specially so, when keypad is going to be used in a number of applications. Proton Basic has made it simple for us by providing a direct command that scans the keypad. Remember the scanning routine will give us a number of key being pressed, the number returned does not tally with the label on key. We have to translate in software the label and correspond it to the accompanying key code being returned.

The PIC Lab-II has PORTB header, which is most suitable for keypad, as PORTB has internal pull-up resistors. PIC Lab-II comes with a flexible 4x3 touch keypad. The keypad includes 15 ohm resistors with

rows to reduce the short circuit among row and column pins. Just connect the keypad connector on the PORTB connector starting from RB0. also connect the LCD to see the results returned by keypad.

Proton Basic provides an InKey command to scan the keypad. Before using the Inkey Command the compiler needs to be informed about the port on which keypad has been connected.

```
Device = 18F452
XTAL = 20
ALL_DIGITAL=true
KEYPAD_PORT PORTB
PORTB_PULLUPS true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Print Cls
Dim x As Byte
loop:
x=InKey
Print At 1,1,"Key Code:", DEC3 x
GoTo loop
```

The x=Inkey command will scan the keypad, and return a number in variable x. this command will not wait for a key to be pressed. If no key is being pressed it will return a value of 16. other values will depend how keypad has been attached. If properly connected following codes should be returned.

1 = 0	2= 001	3=002	4=004	5=006	6=006
7=008	8=009	9=010	* = 012	0=013	# = 014

Notice numbers 003, 007 and 015 are missing, these are for column 4 keys if you have 4 x 4 column keypad. If you do not get these numbers in this order, reverse the keypad connector, so that rows are connected to lower port bits and columns on upper port pins.

Mapping The KeyPad Labels

The keypad is basically a matrix of switches, it may not always be numeric keypad. There can be various symbols or some other labels, as per requirements of the project. Although you as a programmer know which key has been pressed by knowing its code, you can however your life easier by mapping various labels to the returned values. This can be implemented using If statement, the Proton Basic however provides a useful command called LookUp. The lookup command accepts a variable, and a set of labels, which are returned based upon index value given.

```
Device = 18F452
 XTAL = 20
 ALL_DIGITAL=true
 KEYPAD_PORT PORTB
 PORTB_PULLUPS true
 LCD_DTPIN PORTD.4
 LCD_RSPIN PORTD.3
 LCD_ENPIN PORTD.2
 Print Cls
 Dim x As Byte
 Dim a As Byte
 loop:
 x=InKey
 a= LookUp x, [1,2,3,255,4,5,6,255,7,8,9,255,"*",0,"#",255,255]
 Print At 1,1,"Key Code:", DEC3 x
 Print At 2,1,"Key Lbl :", DEC3 a
 GoTo loop
```

This program scans the keypad and reads the keypad value into variable x. the next line uses x as an index to locate a related value in the list. We have put 255 in places where a key does not exist, and the last 255 for number 16 if no key is pressed. The variable ‘a’ will now contain the matching numbers from the list. Note the * and # signs have been enclosed within inverted comas, this will return their ASCII Codes.

Reading Keypad to get a Number

Well so far we have practiced with keypad to read in individual keys. What if we want to read in a value,

and store it in a variable like an integer. Suppose we want to make a password protected device to open a lock when the right password has been entered. The first task would be to read the keypad keys and make a one number. Like if we press 6712 and then * the variable should contain number 6712 which can then be used in any other calculation, comparison or what not. The bare idea is to scan the keypad, and read the digits as 0-9 ignoring the idle state. When a number is read the number is added to the variable. When * is

```
Device = 18F452
XTAL = 20
ALL_DIGITAL=true
KEYPAD_PORT PORTB
PORTB_PULLUPS true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Print Cls
Dim x As Byte
Dim mynumber As Word
start:
Print Cls
mynumber=0
loop:
 x=InKey
 x= LookUp x, [1,2,3,255,4,5,6,255,7,8,9,255,"*",0,"#",255,255]
 If x="*" Then GoTo Exit_loop
 If x <> 255 Then
    DelayMS 500
    mynumber=mynumber *10 + x
   Print Cls, At 1,1, Dec mynumber
 EndIf
GoTo loop
Exit_loop:
Print Cls
Print At 1,1,"You Entered:"
Print At 2,1,Dec mynumber
DelayMS 5000
GoTo start
```

pressed the routine finishes.

The key point in this routine is a variable mynumber, which is word sized variable, so it can hold a maximum value of 65534. its initial value is 0, each time a key is pressed, its value is multiplied by 10, that shifts the present value left by 1 digit, and new digit is added in the units place.

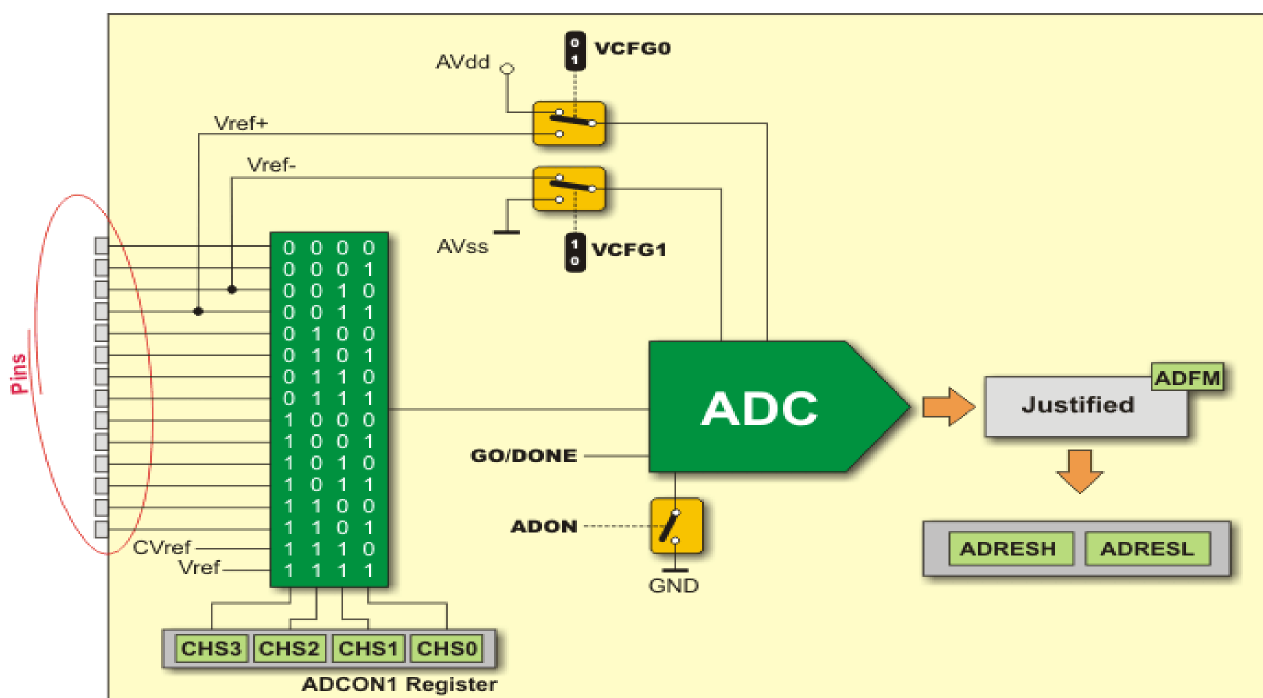
Chapter 13
Analog Module

It is rightly said that we live in Analog World, but we process our information in digital world. Most of the real world data is in Analog form, like temperature, pressure, humidity, altitude, distance, speed, force, voltage, light, radiation, direction, depth and hundreds of more parameters, they are all analog. In order to interact with these analog signals, we have to transform them some how into digital equivalents. The first step in this regards is an appropriate sensor that should be able to detect the particular modality, like temperature and convert the physical world entity into a corresponding electrical signal. The strength of signal in turn corresponds to the measured value.

Most of the sensors return the sensed data as voltage. The strength of physical parameter measured is reflected in the level of voltage returned. In order to use this analog data (voltage) in digital world, it has to be converted into digital equivalent. This process is called Analog to Digital Conversion or ADC. The ADC device has complex design of resistor ladders and networks that sequentially divide the input voltage into discreet levels and then return the value as digital number.

Since interaction with digital world is quite common for microcontrollers, PIC 18F452 has 8 channels of ADC input. The pins associated with analog inputs are also used for other purposes, in order to use them as analog certain registers have to be set. They enable microcontroller to recognize not only whether some pin is driven to logic zero or one (0 or +5V), but to precisely measure its voltage and convert it into numerical value, i.e. digital format. The whole procedure takes place in A/D converter module which has the following features:

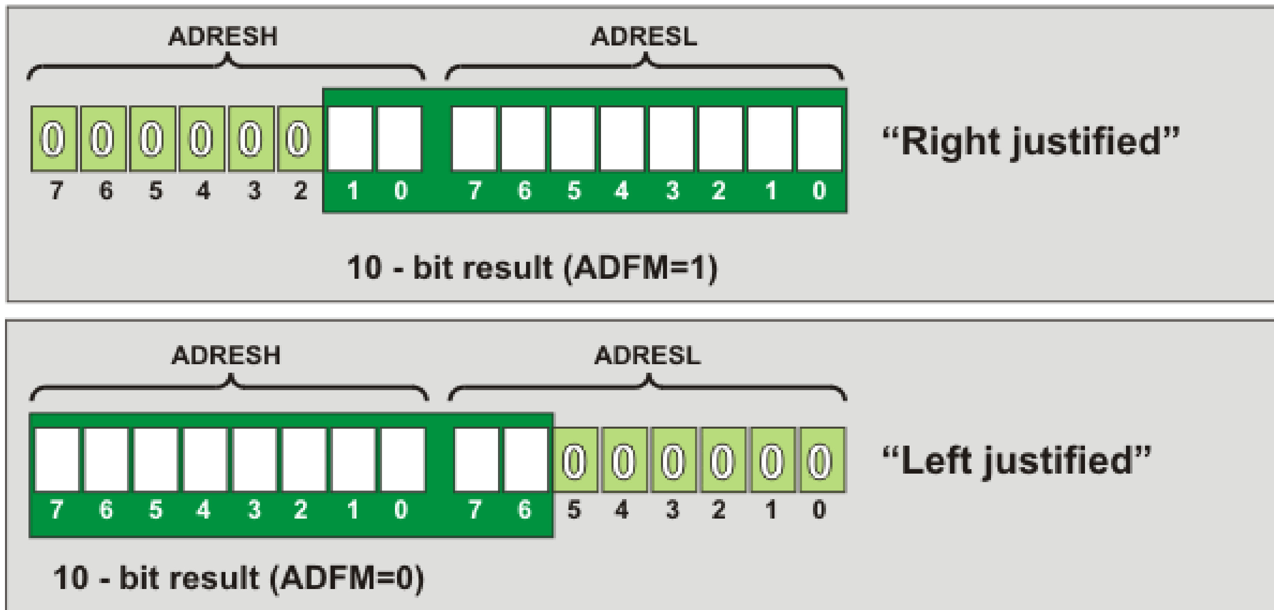
- The converter generates a 10-bit binary result using the method of successive approximation and stores the conversion results into the ADC registers (ADRESL and ADRESH).
- There are 8 separate analog inputs.
- The A/D converter allows conversion of an analog input signal to a 10-bit binary representation of that signal.
- By selecting voltage references Vref- and Vref+, the minimal resolution or quality of conversion may be adjusted to various needs.



Although Analog to digital conversion seems to be a difficult task, yet its really simple and easy when working with PIC microcontrollers. The figure shows an overall plan of ADC. In fact there is only one analog to digital converter. The 8 channels are multiplexed, into ADC module one by one. The selection and configuration of channels is determined by ADCON0 and ADCON1 registers whereas the output of ADC module, which is 10 bit number is given in two 8 bit registers ADRESH and ADRESL. The H and L indicate High Byte and Low byte respectively.

ADRESH and ADRESL Registers

Upon converting an analog value into a digital one, the result of 10-bit A/D conversion will be stored in these two registers. In order to deal with this value easier, it can appear in two formats- left justified and right justified. The ADFM bit of the ADCON1 register determines the format of conversion result (see figure). In case the A/D converter is not used, these registers may be used as general-purpose registers.



A/D Acquisition Requirements

For the ADC to meet its specified accuracy, it is necessary to provide certain time delay between selecting specific analog input and measurement itself. That time is called "acquisition time" and mainly depends on the source impedance. There is an equation used for accurate calculating this time which in the worst case amounts to approximately 20uS. Briefly, after selecting (or changing) the analog input and before starting conversion it is necessary to provide at least 20uS time delay to enable the ACD maximal conversion accuracy.

ADCON0 Register

The ADCON0 register selects two main things. First it selects which channel or pin to use to sample the analog signal and secondly the speed of conversion, also called TAD. The TAD depends on the source of clock signals.

ADCON0 REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON
bit 7							bit 0

The Chanel Select CHS0 - CHS2 are three bits which select the I/O pin to sample. The ADCS0 and ADCS1 bits determine the TAD. The ADON bit powers up the converting module, GO/DONE bit when set to 1 starts conversion. It remains 1 till conversion is going on, when conversion is complete and data has been transferred to ADRES registers this bit is automatically cleared indicating completion.

In order to avoid damage to the digital circuitry, the analog pins must be set to analog settings before getting analog data.

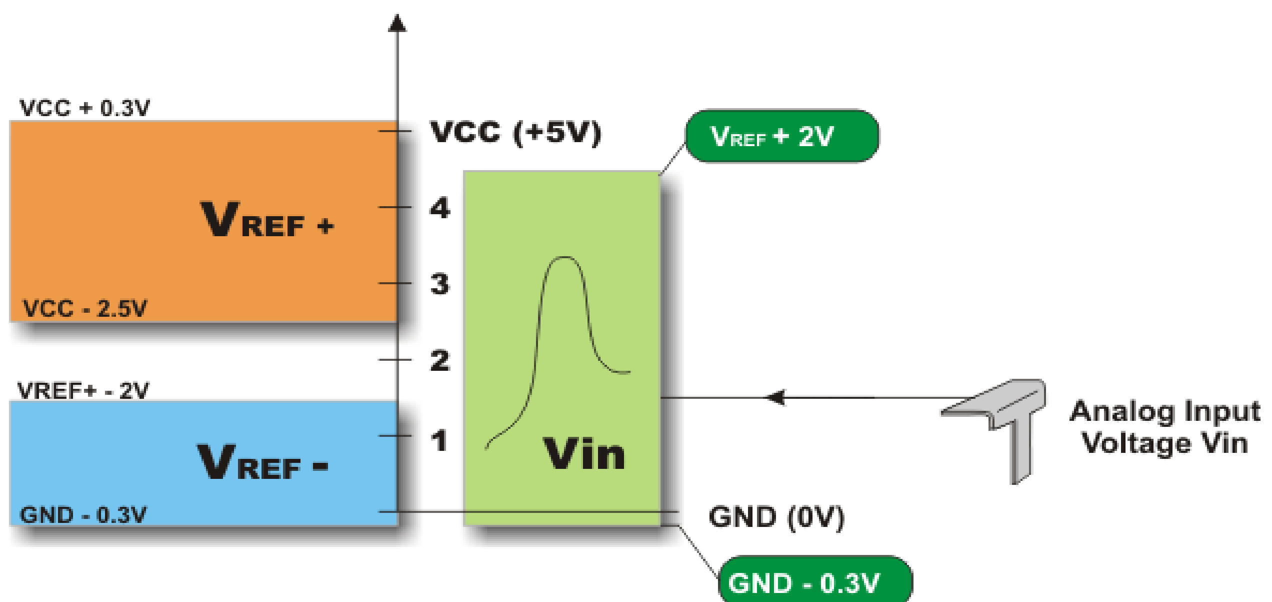
·2: ADCON1 REGISTER

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

The pins to be used as analog are set by bits PCFG0 - PCFG3 see microcontroller data sheet for details. The ADFM bit sets the output format as already said.

Reference Volts

The ADC of 18F452 has 10 bit resolution. This means it can have a value from 0 to 1023. 0 indicating lowest measurable voltage and 1023 maximum measurable voltage. The output is only a number and has to be converted into actual measure by come calculation. Normally it is assumed that the source voltage which is to be measured will range from 0 to 5V. Thus we can say the lower limit is 0V and highest limit is 5V. These two limits are called reference volts. The low reference is called VREF– and high reference is called VREF+ . Thus in standard practice if nothing is set, the VREF– or lower limit is 0 and VREF+ or upper limit is 5V. This means that the output will be 0 when the pin has 0V and the output will be 1023 (all 10 bits set) when input is 5V.



Therefore a scale of 0-5V has 1023 steps. In other words we say that a value of 1023 indicates 5V and each step indicates 5/1023=0.00488V. This means that ADC will measure in increments of 0.00488V. An output of 0 will indicate 0 input volts and an output of 1 will indicate 0.00488V an output of 2 will indicate 0.00976V and so ON. Until a value of 1023 is reached which would indicate 1023 * 0.00488=4.999 (5V, we truncated the 5/1023 result).

Now consider a device which has a maximum output voltage of 2.5V and minimum 0V. In this case we mean that on reaching 2.5V our output should reach 1023. thus each step in output would indicate a 2.5/1023=0.00244V this gives more precision if we adjust our VREF+ to appropriate level. There are two pins on microcontroller marked as VREF+ and VREF –. The settings on PCFG0-PCFG3 bits determines if VREF to be used or not. If the appropriate bits are set and VREF is selected, then an external source of precise volts must be applied to the VREF+ and VREF– pins these volts will then determine the resolution and step of volts measured and the output in ADRES registers.

A look at the data sheet of PIC18F452 shows that PORTA and PORTE lines can be used as analog input pins. The PIC microcontroller analog input pins can tolerate a maximum voltage of 5V. Thus in case your

input has more than 5V, it should be scaled down with a suitable resistor.

Well by now we have discussed the theory of analog to digital conversion in pretty detail. Now is the time to come into action. Although all I/O lines are available as headers on board, we have placed special purpose headers as well for certain jobs. Unplug the LCD and below you will find two headers labeled as AN0 and AN1. these headers also contain GND as well as 5V power supply, and a connection to RA0 and RA1 respectively.

The objective of supplying power supply with these inputs is that most analog devices work on 5V supply, and produce their output as a voltage ranging between 0 and 5V. Thus it is convenient to use this header so that the external device may not need its own power supply. However in case the device has its own supply, you need only to connect the input pin and GND.

One of the simplest method of experimenting, is to attach a variable resistor or potentiometer to the analog input. The two ends of potentiometer will be connected to 5V and GND and the center tape connected to analog input pin. Moving the slider will change the input volts, and the Analog system would be used to sample this input and display data on LCD.

> Note: Attach Analog data cable / Potentiometer after you have programmed the analog software into the microcontroller, so that the pin is declared as analog. Otherwise a previous program might have declared the pin digital, and if POT is fully on left or right giving full 5V or GND and pin in opposite state, will cause a short circuit.

```
' ADC test
Device = 18F452
XTAL=20
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
ADIN_RES    10                      ' Set the resolution to 10
ADIN_TAD    FRC                     ' Choose the RC osc for ADC samples
ADIN_STIME  100                     ' Allow 100us for charge time
ADCON1 = %10000010                  ' Set PORTA analog and right justify result
Input PORTA.0
Dim raw As Word
Dim v As Float
Print Cls
Loop:
raw=ADIn 0
Print At 1,1,"Raw:", DEC4 raw
v=(5/1023)* raw
Print At 2,1, DEC3 v
DelayMS 2000
GoTo Loop
```

This program is reading the state of a potentiometer connected to AN0 header. Three declares have been used to set the analog input. ADIN_RES 10 indicates 10 bit resolution, ADIN_TAD FRC indicates that an internal RC oscillator will be used. ADIN_STIME 100 indicates 100ms sample time.

Then ADCON1 register is set to right justify the return value and configure PORTA.0 as analog.

Notice we did not mention ALL_Digital True in this program. PORTA.0 is made an input pin, the ADIn command reads channel 0 of analog module, which is PORTA.0. the returned value is 10bit number stored in a word sized vari-

able, raw.

The raw variable will contain a value from 0 to 1023. 1023 would mean full 5V. Thus the returned value is converted into volts by (5/1023) * Raw value the result is stored in a float type variable and both values are displayed on LCD. Changing the Potentiometer will change the raw as well as actual volts measured on LCD. You can check the volts present on PORTA.0 or by measuring the volts on center pin of POT and comparing it with the value shown on LCD. Note connecting a voltmeter will slightly affect the reading as it also draws some volts from the circuit.

LM-35 Temperature Sensor

LM-35 is a precision temperature sensor. This is a small 3 pin IC, in TO-92 package. Remember this is an IC. Its center pin is output and other two are power. PIC-Lab-II comes with LM-35 sensor along with connector cable for easy integration. The LM-35 measures temperature in centigrade. Output of LM35 is analog, uniformly linear over the entire range. It rises by 10.0mV / centigrade. This IC does not require external calibration or trimming. Various variants of this commonly used sensor IC are available, which differ in the range of measured temperature.

```
' LM35 temperature
Device = 18F452
XTAL=20
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
ADIN_RES    10                  ' Set the resolution to 10
ADIN_TAD    FRC                 ' Choose the RC osc for ADC samples
ADIN_STIME  100                 ' Allow 100us for charge time
ADCON1 = %10000010              ' Set PORTA analog and right justify result
Input PORTA.0
Dim raw As Word
Dim v As Float
Print Cls
Loop:
raw=ADIn 0
Print At 1,1,"Raw:", DEC4 raw
v=(5/1023)* raw
v=v*1000
Print At 2,1, DEC1 v ,"mv"
v=v/10                          '1 degree centigrade for every 10mV
Print At 2,9,"Tem:", Dec v ,"C"
DelayMS 2000
GoTo Loop
```

For instance if the temperature was 25 degrees Centigrade then the LM-35 would output a voltage of 250mV so setting the DVM to a 2 volt range would allow you to see the current temperature directly (over the whole of the LM35 temperature range). The LM35 measures from -55º to +150ºC. The LM34 measures from -50º to +300ºF

Note: You need to check the exact part number to get the full range as you can buy LM-35 that only cover the range 0º to +100ºC; similarly for the LM-34.

The simplest method of obtaining temperature, is to first measure the volts, the same way as we previously did, then convert the volts into millivolts by multiplying with 1000. Now we know that LM35 produces 10mv for every 1 degree centigrade temperature. So dividing the millivolts by 10 would give us the temperature in Centigrade.

That works fine, just hold the LM35 in your hand and you will see a rise in temperature. You would have noticed that there is lot of variation in temperature, even when environment is same. The reason is that LM35 is very sensitive, so do not rely on 1 reading as we did in the above program, rather record about 10 readings and then take their average. This will stabilize the temperature variation being shown.

Make a temperature controller, that should accept a set temperature, then show the temperature, as soon as temperature is achieved LED0 should go ON.

Well this was just an overview, of how to read and manipulate analog data, a number of sensors are available, that that give analog data about environment, you can use them and analog processor to display and control environmental parameters, like turning ON compressor / heater / Fan etc.

Other related Commands

The analog data can also be obtained technically by a change in resistance, or capacitance. A typical method is to measure the charging time of a capacitor through a resistor. If Capacitor is fixed resistance can be calculated and if resistor is fixed capacitance can be calculated. Such devices include thermistors, which change their resistance with temperature, and humidity sensors which change their capacitance with humidity.

To deal with these situations Proton Basic provides RCIn and POT commands. We recommend reading Proton Basic manual for details.

Chapter 14
On-Chip EEPROM

Microcontroller applications need to store their acquired data, somewhere for later use. This is not always so, but many real world applications need to do so. Although memory variables are one place to keep the data for later use. However as previously said, memory variables are created at run-time in a special area of memory called RAM. This is volatile memory, it tends to clear whenever microcontroller is reset, or power is taken off. Data that needs to permanently stored can be placed within the program memory, this data remains alive as program memory, can only be erased when re-programmed. However the disadvantage is that this data can not be modified during program execution.

In order to address this problem, another type of memory has been introduced, called EEPROM. This memory is electrically erasable, and re-programmable. Once data is stored, like flash memory, it remains alive even when power is taken off. EEPROM memory is commonly used to store various configuration / calibration data, dynamically updated lookup tables, and to remember last time active settings. For example you have made a device to control the speed of a fan. The seed once set, using various buttons, is stored in EEPROM. Next time when power is turned on, the last set speed is read and used to control the fan speed. This avoids the user from setting the values again and again each time system re-boots. In other words EEPROM, also called data EEPROM is used to store and update important device parameters.

Internally there are four special function registers used to control the reading and writing contents to the EEPROM. These are called:

- EECON1
- EECON2
- EEDATA
- EEADR

Proton Basic has made it however extremely simple to write and read data from this memory. Not every microcontroller has EEPROM on chip, while others has smaller number of memory locations and others have really nice chunk of bytes available. PIC18F452, which is used in PIC Lab-II has 256 bytes of EEPROM on chip. Although it looks to be a small amount of memory, yet it is more than enough to store most of the device related configuration information. If more storage is required external EEPROM chips are available (discussed later).

The memory in EEPROM is addressed as bytes. The address of first byte in EEPROM is 0 and increments by 1 successively. While storing and reading data to and from EEPROM, one must be careful about this addressing, as if you are storing a 16 bit data, like a word sized variable, it will occupy 2 bytes of memory, and next storage should be at address 2 bytes away.

Writing data to EEPROM

EWrite is the Proton Basic command to store data in EEPROM. EWrite command will store data during program execution, and is therefore used mainly for updating the memory. Writing to EEPROM is somewhat slower procedure and also consumes its life cycles. Therefore very frequent writes to EEPROM should be avoided if possible. Reading from EEPROM is however exceptionally good, and does not affect its life.

Another method of writing data within EEPROM is at the time of programming. A default set of device parameters are stored at the time of chip burning, and then periodically updated as per requirements. This is done using EData command.

The following program uses EData to store a default number in EEPROM memory, and then the program

accesses this location to display the stored data. New data is however changed using the push switches, and updated in EEPROM, so that next time device starts the updated data is read.

```
'EEProm
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.1
Symbol SW5 = PORTB.0
Dim x As Byte
x=ERead 0            'Read from Address
0 in EEPROM
Print Cls
Print At 1,1,"EEPROM 0:", Dec x
End

EData 100
```

Notice the Edata command at the end of program. If you are programming the microcontroller using ICPROG, when program is loaded into it, notice that apart from program buffer, the EEPROM buffer contains 64 (= 100 decimal) hexadecimal value written. This data is written at the time of programming, now in our program, we read the EEPROM contents using ERead command. This command expects the address to read. Since x is a byte sized variable ERead will automatically read only one byte from the specified location. This program when run will display 100, which is read from EEPROM.

```
'EEProm
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.1
Symbol SW5 = PORTB.0
Dim x As Byte
Start:
x=ERead 0   'from Address 0 in EEPROM
Print Cls
Loop:
Print At 1,1,"EEPROM 0:", DEC3 x
If SW3=0 Then
    x=x+1
    DelayMS 200
EndIf
If SW4=0 Then
    x=x-1
    DelayMS 200
EndIf
If SW5=0 Then
    EWrite 0, [x]
    DelayMS 200
    Print Cls
    Print "EEPROM Updated"
    DelayMS 2000
    GoTo Start
EndIf
GoTo Loop
End

EData 100
```

This program by default contains 100 in byte 0 of EEPROM. Run the program and 100 is displayed. Now press SW3, or SW4 to increase or decrease the value. When a new value is selected press SW5 this will write the new value into EEPROM, byte 0, effectively updating the previous value of 100. Now power off the board and restart it, this time the newly selected value is displayed.

As previously said if the data stored is not byte sized, it will occupy more bytes, so be careful about the address while reading it back. EEPROM is also used to act as data logger device, where sequential data record is stored in EEPROM to be retrieved later.

Lets develop a small application that records the environment temperature and after every minute stores the current temperature into EEPROM. The data can later be retrieved and viewed using Switches.

```
'EEProm
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
ADIN_RES     10                           ' Set the resolution to 10
ADIN_TAD     FRC                          ' Choose the RC osc for ADC samples
ADIN_STIME   100                    ' Allow 100us for charge time
```

```
ADCON1 = %10000010                         ' Set PORTA analog and right justify
result
Input PORTA.0
Symbol SW3 = PORTE.0
Symbol SW4 = PORTE.1
Symbol SW5 = PORTB.0
Dim x As Byte
Dim raw As Word
Dim c As Float
Dim n As Byte
n=0
Print Cls
Print At 1,1,"Temp Logger"
Loop:
If SW5=0 Then
    DelayMS 200
    GoSub Show
EndIf
raw=ADIn 0
c=(5/1023)*raw
c=c*1000
c=c/10
Print At 2,1,DEC2 c," C"
EWrite n,[raw]
Print At 2,8,"Log:", DEC3 n
n=n+2
If n>256 Then n=0
DelayMS 5000
GoTo Loop

Show:
Print Cls
Print At 1,1,"Review:"
DelayMS 1000
Dim k As Byte
k=0
AA:
If SW3=0 Then
    DelayMS 200
    k=k-2
EndIf
If SW4=0 Then
    DelayMS 200
    k=k+2
EndIf
If SW5=0 Then
    DelayMS 200
    GoTo rr
EndIf
raw=ERead k
c=(5/1023)*raw
c=c*1000
c=c/10
Print At 2,1,DEC2 c," C"
Print At 1,8,"Log:", DEC3 k
GoTo AA
rr:
Print Cls
Print At 1,1,"Temp Logger"
Return
```

This program, although not very advanced, but shows many important procedures. Whenever the processor starts it starts logging the temperature, into EEPROM. Instead of storing the floating point number it stores the raw data obtained from ADC. Mean while the program scans for switches, if SW5 is pressed it halts data logging and branches to a subroutine to show/review data, now using SW3 or SW4 you can view the stored data, converted into temp. again when SW5 is pressed, review is cancelled and program enters into data logging again starting over where it halted.

There is lot to improve, you can think yourself, like a procedure to clear the log, know exactly how many logs are there etc.

Data loggers are a big market in microcontroller applications. For example, consider a shipment of frozen food to some country, the temperature data logger will keep on recording the temperature every hour. On arrival the company can inspect the logged temperatures to know if temperature increased beyond certain level, or if it remained within safe limits. Similarly a medical equipment recording blood pressure of a patient can store the recorded values, so that a review can be made about how blood pressure fluctuated.

Write a temperature logger, and display the review as graphic data on GLCD

Chapter 15

On-Chip CCP

Capture | Compare | PWM

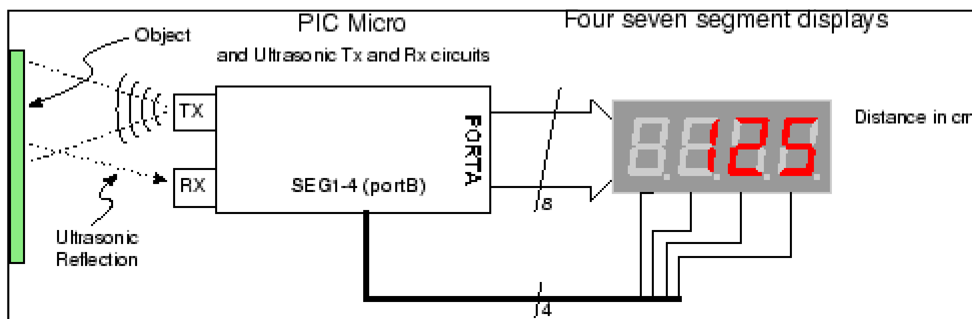
This chapter will discuss, CCP which stands for Capture, Compare and Pulse width modulation is one of the most complicated modules in PIC microcontrollers. I will go through this module only briefly, as not overburden the beginner. In fact this module does three functions, or it has three modes. There are two such modules present in PIC18F452.

In **Capture Mode**, the peripheral allows timing of duration of an event. This circuit gives insight into the current state of some register which constantly changes its value. In this case, it is the timer TMR1 register. Thus with this mode we can measure for how long a pin remained in logic 1 or 0.

The **Compare Mode** compares values contained in two registers at some point. One of them is the timer TMR1 register. This circuit also allows the user to trigger an external event when a predetermined amount of time has expired. Thus if you set a compare register to some value, when Timer1 reaches that value, a capture event takes place and an interrupt signal is fired indicating that a predefined time period has been elapsed.

PWM - *Pulse Width Modulation* can generate signals of varying frequency and duty cycle. The ratio between ON and OFF states of the pulse determines the amount of energy transferred to device. This method is useful in controlling the speed of motors, or brightness of LEDs etc.

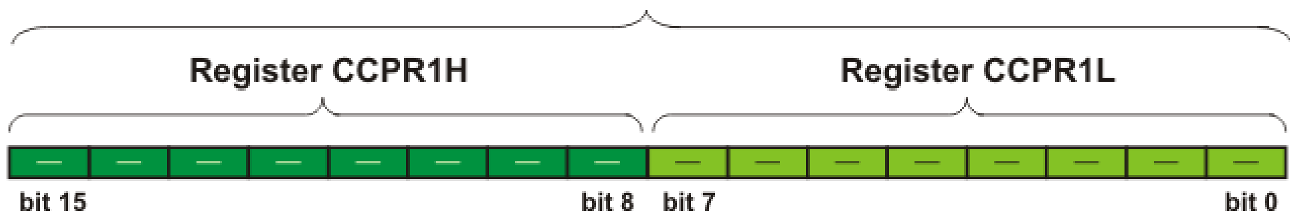
Now consider an example of ultrasonic range finder. The project consists of a few driving transistors, and standard 40KHz ultrasonic transducers. The microcontroller uses its CCP module in capture mode. So the value of timer1 is noted and an impulse is given on Tx transducer, when the impulse is returned the CCP module stores the new value of Timer1 in CCP register, subtracting the previous value from current value



gives you the time to echo. From this you can easily calculate distance. (see projects section).

If we talk about CCP1, it is the same as CCP2. The central point in CCP module is CCPR register. This is a 16 bit register and consists of a H and L parts. This register contains the values captured or compared with

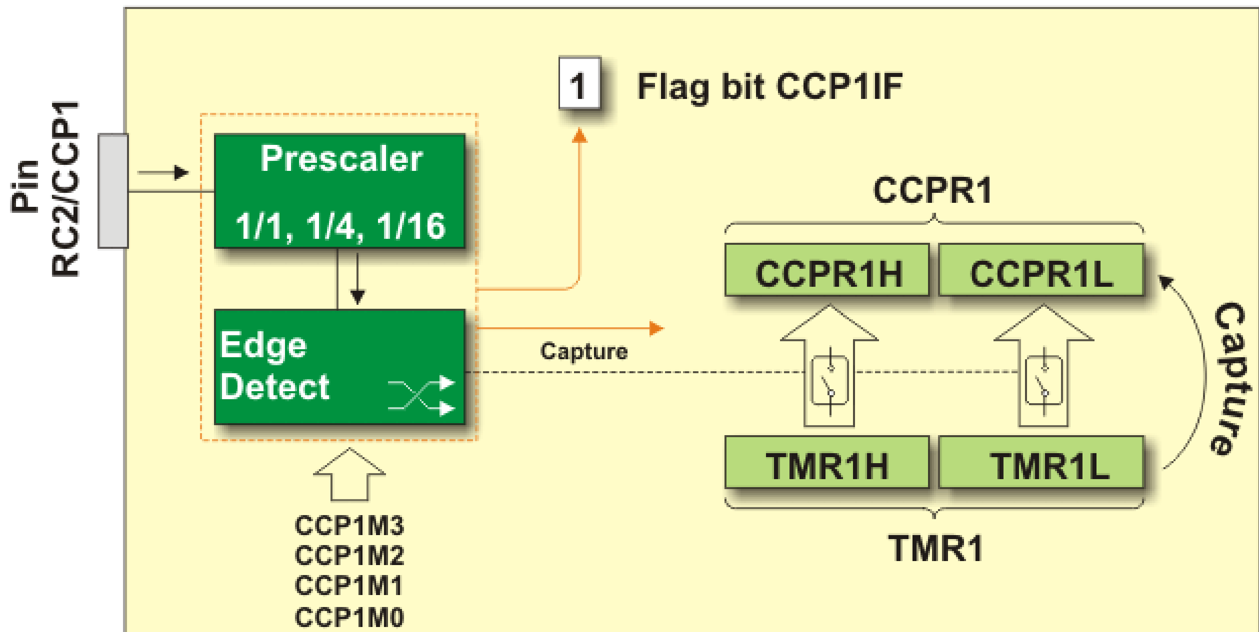
Module CCPR1 Registers



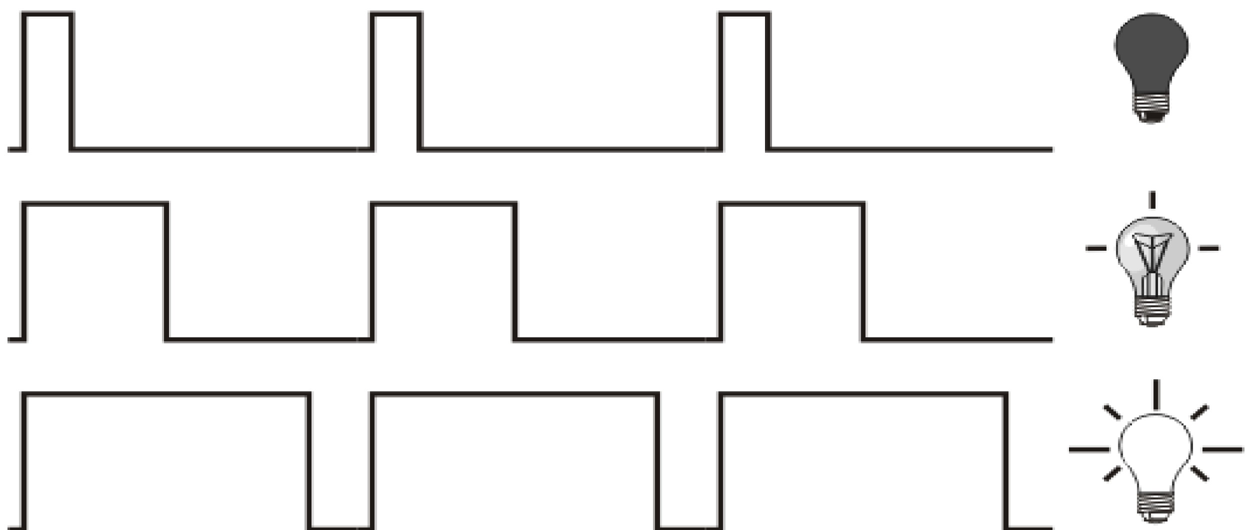
Timer1. Remember when Capture is initiated, Timer1, is not initialized. And at the time of capture, whatever the value of Timer1 register is, is copied into this register.

Pulse Width Modulation

Pulse width modulation is a technique where digital data is used to control the energy transfer to a device.



Whenever a digital signal is high it is powering the target device, like a transistor, or LED. When it is Low it is not powering the device. If the line is constantly kept high, full energy (100%) is being transferred to the device and when it is constantly Low, there is No energy transfer. In between if the line is On for some time and Off for sometime, the energy delivered depends upon the ON time / Off time ratio as well as the frequency of pulses. As you can see in this figure when On time is small and Off time long, Bulb hardly gets any time to turn ON, As the ON time is increased and OFF time decreased it gets brighter. This is called the Duty Cycle. So the duty cycle is ratio between ON and OFF. A 50% duty cycle is equal ON and Equal OFF time per cycle.

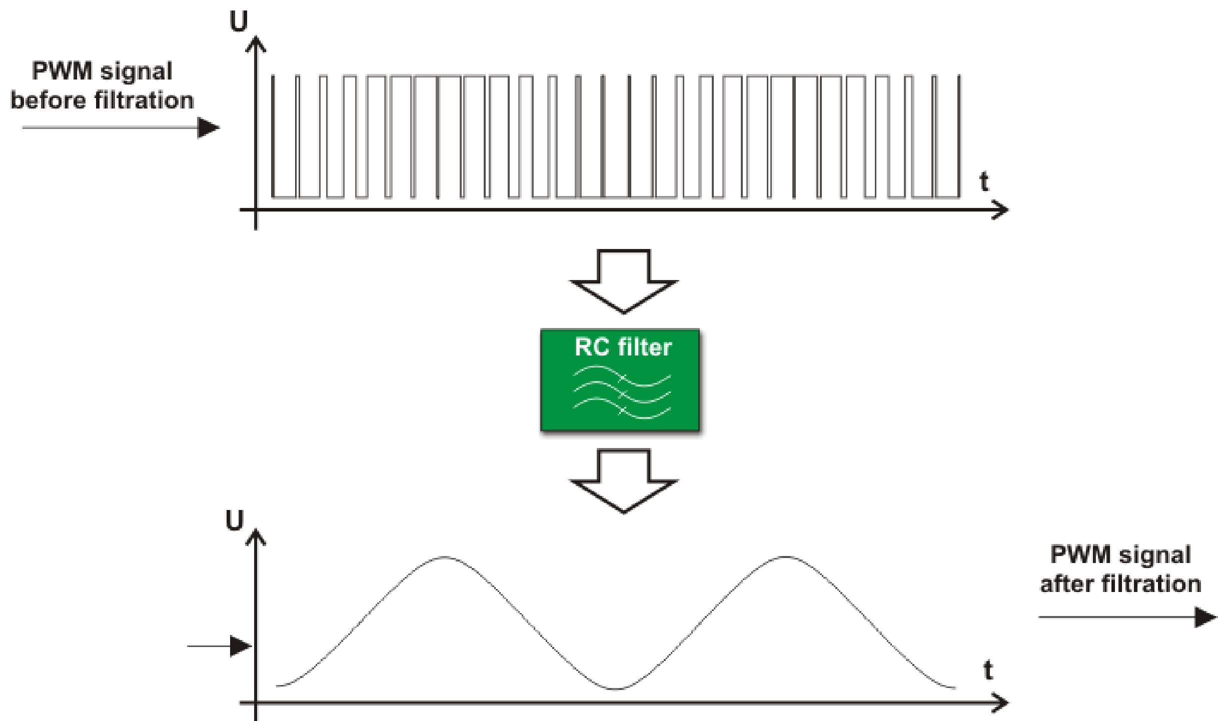


Another common usage of PWM is creation of various kinds of waveforms, like sine wave. If you recall the sounds chapter, various types of sound waves are formed by internally using PWM.

In order to produce a waveform from the digital circuit, we have to include some sort of filter. This filter can be as simple as an RC-Filter, which charges the capacitor, and gradually discharges. The rate and speed of charging is influenced by the width of pulse.

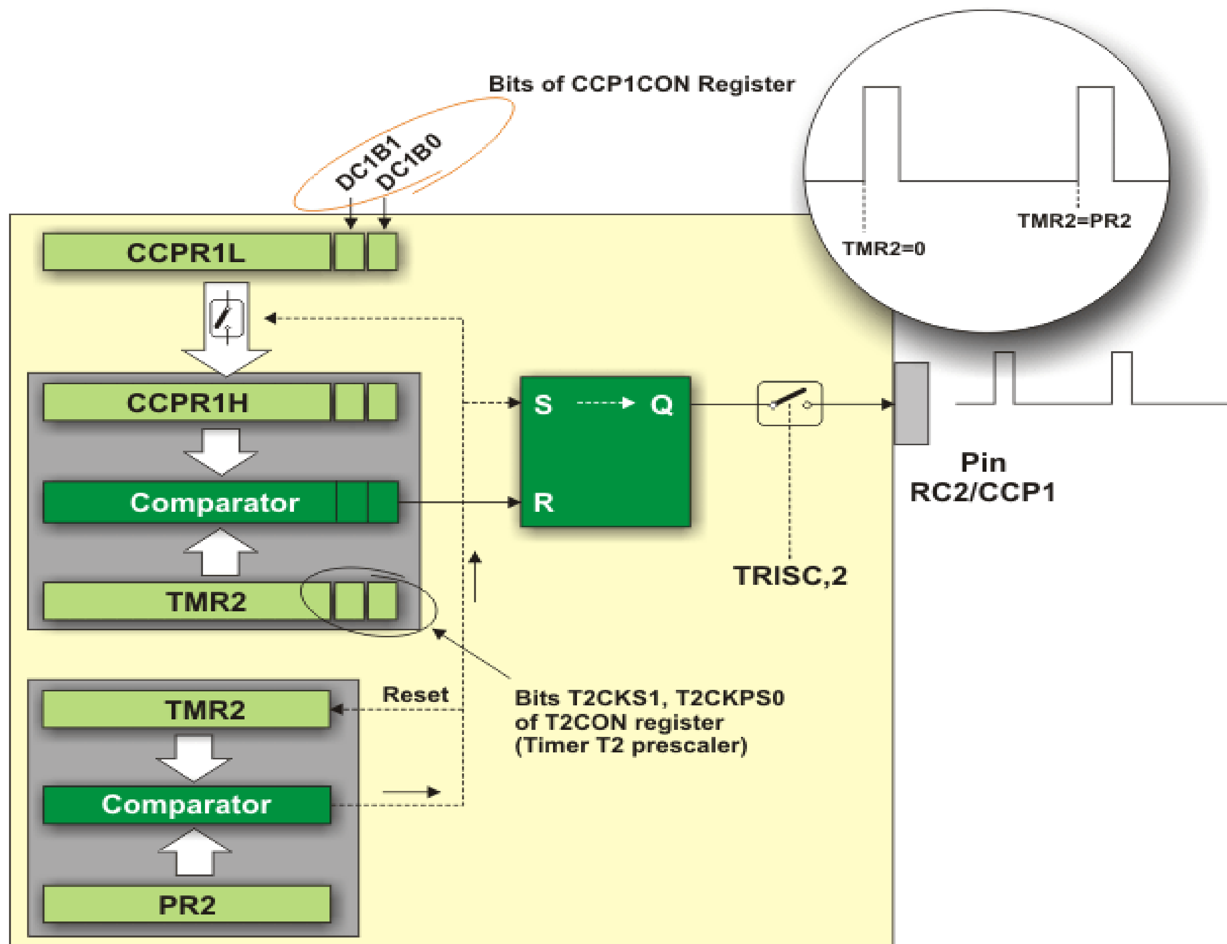
Notice when pulses are wide, the waveform reaches peak, and when the pulses are narrow, with smaller duty cycle the waveform falls down. Devices which operate in this way are often used in practice as switching regulators which control the operation of motors (speed, acceleration, deceleration etc.).

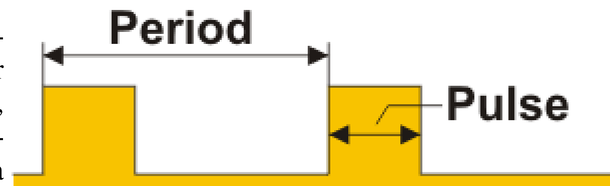
The figure above shows block diagram of the CCP1 module setup in PWM mode. In order to generate a



pulse of arbitrary form on its output pin, it is necessary to determine only two values- pulse frequency and duration.

Although a lot needs to be discussed about these various modules, and various registers that set these pa-





rameters, I think it will be rather confusing for a beginner. So now lets come to the business, and see how our Proton Basic is going to help. Just like other commands, as we have seen, all the register level details are managed by the Proton Basic itself, and we are left with a neat easy to use code. Nevertheless a sound understanding of the things behind the scene makes a real difference.

As you know Pulse width Modulation as such is a technique, if you can produce On/OFF wave on any pin, it can be used as PWM output. In this example we are going to produce a PWM output on pin PORTC.0 which is also connected to the LED on PIC Lab-II board. So that we can see the effect. You can make a waveform simply by changing the ON and OFF times. In this example we have used a command offered by

```
Device=18F452
XTAL=20
ALL_DIGITAL true
Symbol LED PORTC.0
Symbol SW3 PORTE.0
Symbol SW4 PORTE.1
Input SW3
Input SW4
Output LED
Dim x As Byte
x=100
loop:
If SW3=0 Then x=x-10:DelayMS 200
If SW4=0 Then x=x+10:DelayMS 200
PWM LED,x,1000
GoTo loop
```

Proton Basic called PWM, accepts a pin as parameter and duty cycle as second parameter, the number of pulses to be sent as last parameter. Duty is a variable or a constant which specifies the analog level required. It ranges from 0-255. 255 produces full 5V.

PWM emits a burst of 1s and 0s whose ratio is proportional to the *duty* value you specify. If *duty* is 0, then the pin is continuously low (0); if *duty* is 255, then the pin is continuously high. For values in between, the proportion is *duty*/255. For example, if *duty* is 100, the ratio of 1s to 0s is 100/255 = 0.392, approximately 39 percent. When such a burst is used to charge a capacitor arranged, the voltage across the capacitor is equal to:-

(duty/ 255) * 5.

So if *duty* is 100, the capacitor voltage is

(100/255) * 5 = 1.96 volts.

This voltage will drop as the capacitor discharges through whatever load it is driving. The rate of discharge is proportional to the current drawn by the load; more current = faster discharge. You can reduce this effect in software by refreshing the capacitor's charge with frequent use of the **PWM** command. You can also buffer the output using an op-amp to greatly reduce the need for frequent **PWM** cycles.

So we have used a standard I/O line for PWM, that is fairly good. However to keep the PWM going on the instruction must be executed continuously.

The hardware PWM module eliminates this need and continuously gives PWM pulses on the specific PWM pin, while the program continues doing something else. This is really a sort of multitasking. The output of CCP1 and CCP2 modules are hard-wired to specific pins and they may vary among PICs, so always read the datasheet. On PIC18F452 CCP1 is on RC2 pin, and fortunately we have LED on that pin as well, so we can test Hardware PWM right on board.

Notice that after declaring the HPWM statement, the processor is busy in an endless loop to display some data on LCD, while the CCP1 module is producing PWM pulses on the specified channel. Since CCP modules pins vary among processors, it is advisable to declare the CCP pin in program.

```
Device=18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
```

```
LCD_ENPIN PORTD.2
Symbol LED PORTC.2
Symbol SW3 PORTE.0
Symbol SW4 PORTE.1
Input SW3
Input SW4
Output PORTC
CCP1_PIN PORTC.2
Dim x As Byte
PORTC=0
HPWM 1,50,1000
Cls
Print At 2,1,"PWM ON"
loop:
For x=0 To 255
Print At 1,1,DEC3,x
DelayMS 200
Next x
GoTo loop
```

Chapter 16
Pulse

Many devices need a variety of pulses to operate. Some need continuous pulses at a certain frequency, while others need just a single pulse. Some applications need to time the incoming pulse. All these applications are discussed in this section.

PulsOut:

This command is used to send a pulse of specific duration on the specified pin. The syntax of this command is:

```
PULSOUT Pin , Period, { Initial State }
```

The pin is any digital I/O line, this line will automatically be declared as output. Period is time duration for which a pulse is to remain high or low whatever the case may be. The initial state is optional. If initial state is 0, a pulse of logical 1 is generated for specified duration and then the initial state of 0 is restored. If your device requires a logical 0 as pulse then the initial state may be set as logical 1.

We use this method to issue reset pulse, or advance the counter by 1, as you will see in our projects involving shift registers. The period of pulse is dependent upon clock frequency. If using 4MHz, the period increments time as 10us and if its 20MHz the increment is 2uS.

Counter:

This command counts the number of pulses arriving on a specific pin in a specified time. Its general syntax is:

```
Variable = COUNTER Pin , Period
```

The counter uses clock declaration as a time base, therefore period is set as milliseconds. Counter checks the state of the pin in concise loop, and counts the rising edges of pulses, that means a change from low to high. With a 4MHz Oscillator the pin is checked every 20uS and with 20MHz it is checked every 4uS. Thus the highest frequency that can be counted is 25KHz for 4MHz crystal and 125KHz for 20MHz crystal.

PulsIn:

This command measures the time duration of a single pulse arising at the specified pin. The specified pin is automatically made as Input.

```
Variable = PULSIN Pin , State
```

The state indicates the edge, when to start counting. Pulsing command uses a fast clock counter, it starts counting when first change takes place and stops counting when the change again takes place. The pulsing command waits for a max of 0.655 seconds for a change, if no change is detected it returns 0. the value returned depends upon the type of variable, and clock frequency. A byte variable can hold max of 255, while word type variable can hold 65535 units. Each unit measures 10us for 4MHz clock and 2uS for 20MHz clock.

Sending out serial data

Although we have learnt how to send asynchronous serial data in chapter on USART, some devices require synchronized data transfer. Thus they have a clock line and a data line. The data line sends data on every clock pulse. The receiving device also synchronizes its reception with clock signals.

SHOUT

The SHOUT command (Shift Out) shifts the contents of a Byte or word out of a single pin, one bit at a

time, synchronized with the second clock pin. The SHOUT command is commonly used to send data out into the shift registers, which accept bits and shift them forwards. While shifting data out we can mention which part of byte to be shifted first. LSBFIRST means shifting out the least significant bit first, similarly MSBFIRST starts from the highest bit first. We can also mention if whole byte or word is to be shifted or just a part of it, like 6 bits to be shifted.

We shall see details of this command in our projects on 8x32 led matrix, which uses shift registers.

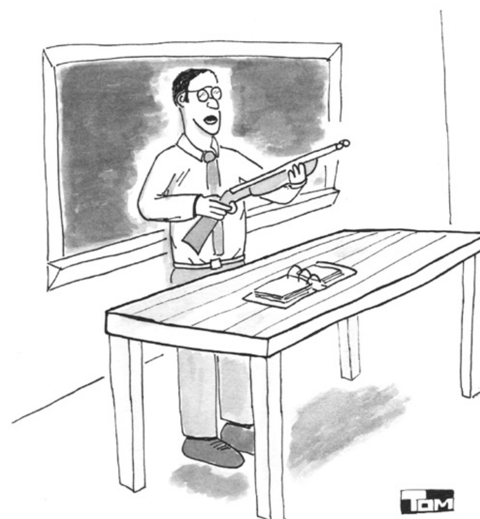
```
SHOUT DTA , CLK , MSBFIRST , [ 250 ]
```

Where DTA and CLK are the data and clock pins.

Exactly opposite to this is SHIN command, that accepts synchronized data.

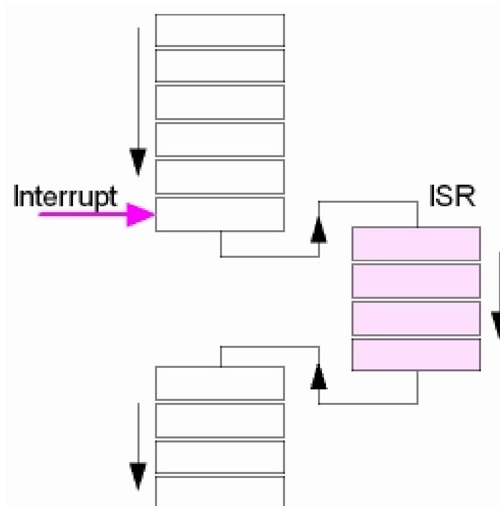
Chapter 17
Interrupts

The subject of interrupts is probably going to be the longest and most difficult to go through. There is no easy way of explaining interrupts, but hopefully by the end of this section you will be able to implement interrupts into your own programs. We have split the section into two parts. This is to help break the subject up, and to give you, a break. So what is an interrupt? Well, as the name suggests, an interrupt is a process or a signal that stops a microprocessor/microcontroller from what it is doing so that something else can happen. Let me give you an every day example. Suppose you are sitting at home, chatting to someone. Suddenly the telephone rings. You stop chatting, and pick up the telephone to speak to the caller. When you have finished your telephone conversation, you go back to chatting to the person before the telephone rang. You can think of the main routine as you chatting to someone, the telephone ringing causes you to interrupt your chatting, and the interrupt routine is the process of talking on the telephone. When the telephone conversation has ended, you then go back to your main routine of chatting. This example is exactly how an interrupt causes a processor to act. The main program is running, performing some function in a circuit, but when an interrupt occurs the main program halts while another routine is carried out. When this routine finishes, the processor goes back to the main routine again.



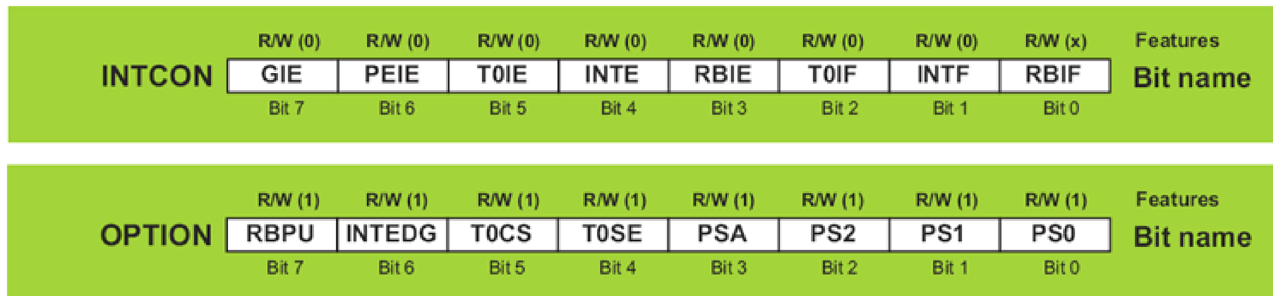
The PIC has 4 sources of interrupt. They can be split into two groups. Two are sources of interrupts that can be applied externally to the PIC, while the other two are internal processes. We are going to explain the two external ones here. The other two will be explained in timers and storing data.

If you look at the pin-out of the PIC, you will see that pin 6 shows it is RB0/INT. Now, RB0 is obviously Port B bit 0. The INT symbolizes that it can also be configures as an external interrupt pin. Also, Port B bits 4 to 7 can also be used for interrupts. Before we can use the INT or other Port B pins, we need to do two things. First we need to tell the PIC that we are going to use interrupts. Secondly, we need to specify which port B pin we will be using as an interrupt and not as an I/O pin.



Inside the PIC there is a register called INTCON, and is at address 0Bh. Within this register there are 8 bits that can be enabled or disabled. Bit 7 of INTCON is called GIE. This is the Global Interrupt Enable. Setting this to 1 tells the PIC that we are going to use an interrupt. Bit 4 of INTCON is called INTE, which means INTerrupt Enable. Setting this bit to 1 tells the PIC that RB0 will be an interrupt pin. Setting bit 3, called RBIE, tells the PIc that we will be using Port B bits 4 to 7. Now the PIC knows when this pin goes high or low, it will need to stop what it's doing and get on with an interrupt routine. Now, we need to tell the PIC whether the interrupt is going to be on the rising edge (0V to +5V) or the falling edge (+5V to 0V) transition of the signal. In other words, do we want the PIC to interrupt when the signal goes from low to high, or from high to low. By default, this is set up to be on the rising edge. The edge 'triggering' is set up in

another register called the OPTION register, at address 81h. The bit we are interested in is bit 6, which is called INTEDG. Setting this to 1 will cause the PIC to interrupt on the rising edge (default state) and setting it to 0 will cause the PIC to interrupt on the falling edge. If you want the PIC to trigger on the rising edge, then you don't need to do anything to this bit.



Ok, so now we have told the PIC which pin is going to be the interrupt, and on which edge to trigger, what happens in the program and the PIC when the interrupt occurs? Two things happen. First, a 'flag' is set. This tells the internal processor of the PIC that an interrupt has occurred. Secondly, the program counter which points to a particular address within the PIC. Let's quickly look at each of these separately.

Interrupt Flag

In our INTCON register, bit 1 is the interrupt flag, called INTF. Now, when any interrupt occurs, this flag will be set to 1. While there isn't an interrupt, the flag is set to 0. And that is all it does. Now you are probably thinking 'what is the point?' Well, while this flag is set to 1, the PIC cannot, and will not, respond to any other interrupt. So, let's say that we cause an interrupt. The flag will be set to 1, and the PIC will go to our routine for processing the interrupt. If this flag wasn't set to 1, and the PIC was allowed to keep responding to the interrupt, then continually pulsing the pin will keep the PIC going back to the start of our interrupt routine, and never finishing it. Going back to my example of the telephone, it's like picking up the telephone, and just as soon as you start to speak it starts ringing again because someone else want to talk to you. It is far better to finish one conversation, then pick up the phone again to talk to the second person.

There is a slight drawback to this flag. Although the PIC automatically sets this flag to 1, it doesn't set it back to 0! That task has to be done by the programmer – i.e. you. This is easily done, as We are sure you can guess, and has to be done after the PIC has executed the interrupt routine.

Memory Location | Interrupt Routine

When you first power up the PIC, or if there is a reset, the Program Counter points to address 0000h, which is right at the start of the program memory. However, when there is an interrupt, the Program Counter will point to address 0004h. So, when we are writing our program that is going to have interrupts, we first of all have to tell the PIC to jump over address 0004h, and keep the interrupt routine which starts at address 0004h separate from the rest of the program. This is very easy to do.

First, we start our program with a command called ORG. This command means Origin, or start. We follow it with an address. Because the PIC will start at address 0000h, we type ORG 0000h. Next we need to skip over address 0004h. We do this by placing a GOTO instruction, followed by a label which points to our main program. We then follow this GOTO command with another ORG, this time with the address 0004h. It is after this command that we enter our interrupt routine. Now, we could either type in our interrupt routine directly following the second ORG command, or we can place a GOTO statement which points to the interrupt routine. It really is a matter of choice on your part. To tell the PIC that it has come to the end of the interrupt routine we need to place the command RTFIE at the end of the routine. This command means return from the interrupt routine. When the PIC see this, the Program Counter points to the last location the PIC was at before the interrupt happened.

This is how we set an interrupt system in Assembly. However in Proton Basic it is simply a procedure. Since interrupt routines have to be fast and release the processor from interrupt as soon as possible, many programmers prefer to manage the interrupts in assembly.

There are two things you should be aware of when using interrupts. The first is that if you are using the

same register in your main program and the interrupt routine, bear in mind that the contents of the register will probably change when the interrupt occurs. For example, let's you are using the w register to send data to Port A in the main program, and you are also using the w register in the interrupt routine to move data from one location to another. If you are not careful, the w register will contain the last value it had when it was in the interrupt routine, and when you come back from the interrupt this data will be sent to Port A instead of the value you had before the interrupt happened. The way round this is to temporarily store the contents of the w register before you use it again in the interrupt routine. The second is that there is a delay between when one interrupt occurs and when the next one can occur. As you know, the PIC has an external clock, which can either be a crystal or it can be a resistor-capacitor combination. Whatever the frequency of this clock, the PIC divides it by 4 and then uses this for it's internal timing. For example if you have a 4MHz crystal connected to your PIC, then the PIC will carry out the instructions at 1MHz. This internal timing is called an Instruction Cycle. Now, the data sheet states (admittedly in very small print) that you must allow 3 to 4 instruction cycles between interrupts. My advice is to allow 4 cycles. The reason for the delay is the PIC needs time to jump to the interrupt address, set the flag, and come back out of the interrupt routine. So, bear this in mind if you are using another circuit to trigger an interrupt for the PIC.

Now, a point to remember is that if you use bits 4 to 7 of Port B as an interrupt. You cannot select individual pins on Port B to serve as an interrupt. So, if you enable these pins, then they are all available. So, for example, you can't just have bits 4 and 5 – bits 6 and 7 will be enabled as well. So what is the point of having four bits to act as an interrupt? Well, you could have a circuit connected to the PIC, and if any one of four lines go high, then this could be a condition that you need the PIC to act on quickly. One example of this would be a house alarm, where four sensors are connected to Port B bits 4 to 7. Any sensor can trigger the PIC to sound an alarm, and the alarm sounding routine is the interrupt routine. This saves examining the ports all the time and allows the PIC to get on with other things.

We covered quite a bit of ground , and so we think it is time that we wrote our first program. The program we are going to write will count the number of times we turn a switch on, and then display the number. The program will count from 0 to 9, displayed on 4 LEDs in binary form, and the input or interrupt will be on RB0. in PIC Lab-II SW5 is connected to RB0. although it is active low, we are going to use interrupt on rising edge, thus the interrupt will take place when key is released. The processor will be held in an endless loop, from which it can not come out. Under normal circumstances if the processor is busy in some loop, it can not scan the input buttons, however using interrupt it will attend the button press.

```
Device=18F452
XTAL=20
ALL_DIGITAL true
Symbol GIE INTCON.7
Symbol INT0IE INTCON.4
Symbol INT0F INTCON.1
Dim x As Byte
Output PORTC
on_interrupt GoTo Jingle
GoTo Start
Jingle:
x=x+1
PORTC=x
INT0F=0
Context Restore

Start:
x=0
PORTC=0
INT0IE = 1
GIE=1
aa:
GoTo aa
```

Notice the interrupt routine starting at label Jingle, we have defined the symbols, for easy understanding for enabling general interrupt system GIE, enabling RB0 interrupt also called INT0 and IN0 flag INT0F then we have defined variables and direction of PORTC. Next we have issued the on_interrupt command that tells the compiler where to branch whenever an interrupt takes place. Since the code at jingle is to be executed on interrupt we jump over it to start label. Here we have initialized our variables and ports, and enable INT0, and the enable GIE. Then we put the processor into an endless loop. Without interrupt system, this program should not respond to key press. Bust since interrupt is enabled, when SW5 (RB0) is pressed and released (trigger on rising edge) the program will jump to interrupt, here we do something, and on finishing, reset the interrupt flag to 0 and context restore command restores the stack, and other registers used for jump, back to the state in which they were before interrupt.

Now pressing the SW5, will change the LEDs on PORTC.

Interrupts can also take place on internal events, like a timer / counter reaching its maximum value, data received on USART port, ADC conversion completed and others. These interrupts are called hardware interrupts as they occur on peripheral devices present within the PIC hardware. We shall talk about timers and timer interrupts in chapter on timers.

Chapter 18
Timers & Interrupts

There are three timers on board in PIC microcontroller. These timers can also be used as counters when external pulses are applied. The timers are programmable, and sometimes share with other peripheral devices. These are named as TMR0, TMR1 and TMR2 there are few other timers, not to be discussed here like Watchdog Timer and Brown-Out timers. These timers are useful in measuring the time delays in various events as well as counting and timing external events.

Pre-Scalar

Sometimes the number of pulses being presented to the timers / counters can be so enormous that the register associated with counting the events would get full before our event is finished. In that case, instead of measuring every pulse we configure a pre-scalar for the timer. The pre-scalar divides the pulses by a factor of say 4, 8 and so on. Thus if a pre-scalar of 4 is being used the timer will increment by one on every fourth pulse. Thus when the event is finished actual pulses will be the timer register value multiplied by 4.

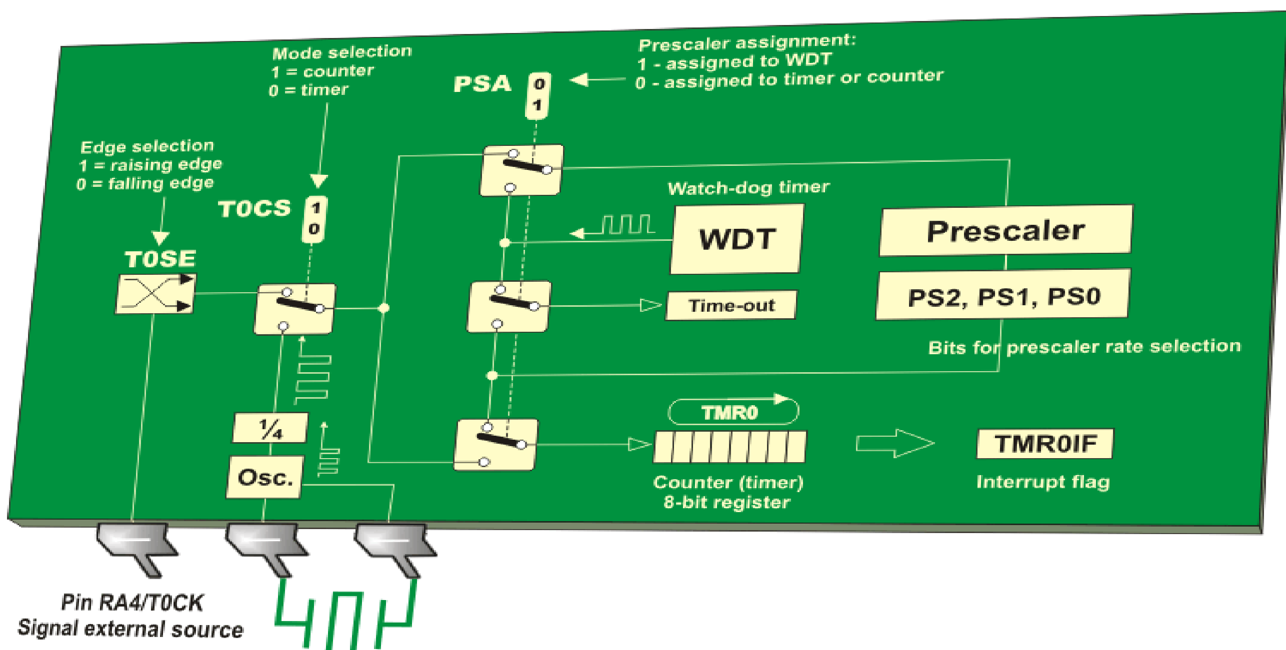
Timer TMR0

The timer TMR0 has a wide range of applications in practice. Only few programs do not use it in some way. Even simple, it is very convenient and easy to use for writing program or subroutine for generating pulses of arbitrary duration, time measurement or counting external pulses (events) almost with no limitations.

The timer TMR0 module is an 8-bit timer/counter with the following features:

- 8-bit timer/counter register
- 8-bit prescaler (shared with Watchdog timer)
- Programmable internal or external clock source
- Interrupt on overflow
- Programmable external clock edge selection

Figure below represents the timer TMR0 schematic with all bits which determine its operation. These bits



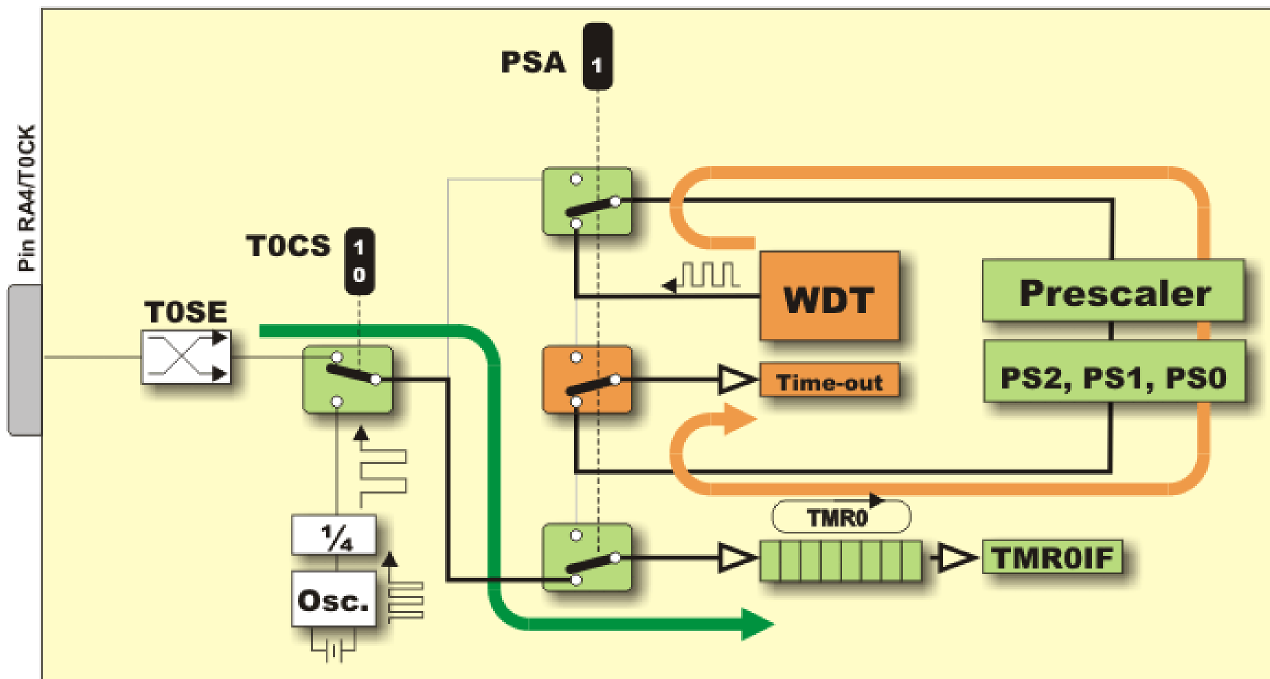
are stored in the T0CON register.

T0CON: TIMER0 CONTROL REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

bit 7 **TMR0ON:** Timer0 On/Off Control bit
1 = Enables Timer0
0 = Stops Timer0
bit 6 **T08BIT**: Timer0 8-bit/16-bit Control bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter
bit 5 **T0CS**: Timer0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKO)
bit 4 **T0SE**: Timer0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin
bit 3 **PSA**: Timer0 Prescaler Assignment bit
1 = TImer0 prescaler is NOT assigned. Timer0 clock input bypasses prescaler.
0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.
bit 2-0 **T0PS2:T0PS0**: Timer0 Prescaler Select bits
111 = 1:256 prescale value
110 = 1:128 prescale value
101 = 1:64 prescale value
100 = 1:32 prescale value
011 = 1:16 prescale value
010 = 1:8 prescale value
001 = 1:4 prescale value
000 = 1:2 prescale value

PS2	PS1	PS0	TMR0	WDT
0	0	0	1:2	1:1
0	0	1	1:4	1:2
0	1	0	1:8	1:4
0	1	1	1:16	1:8
1	0	0	1:32	1:16
1	0	1	1:64	1:32
1	1	0	1:128	1:64
1	1	1	1:256	1:128



In addition to above mentioned, this is also useful to know:

- When the prescaler is assigned to the timer/counter, any write to the TMR0 register will clear the prescaler.
- When the prescaler is assigned to watch-dog timer, a CLRWDT instruction will clear both the prescaler and WDT.
- When writing to the TMR0 register used as a timer, will not cause the pulse counting to start immediately, but with two instruction cycles delay. In accordance to that, it is necessary to adjust the value written to the TMR0 register.
- When the microcontroller is setup in *sleep* mode, the oscillator is turned off. Overflow cannot occur since there are no pulses to count. That is why the TMR0 overflow interrupt cannot wake up the processor from Sleep mode.
- When used as external clock counter without prescaler, a minimal pulse length or a pause between two pulses must be 2 Tosc + 20 nS. Tosc is oscillator signal period.
- When used as external clock counter with prescaler, a minimal pulse length or a pause between two pulses is 10nS.
- 8-bit prescaler register is not available to the user, which means that it cannot be directly read or written.

In order to use TMR0 properly, it is necessary:

To select mode:

- Timer mode is selected by the T0CS bit of the OPTION_REG register, (T0CS: 0=timer, 1=counter).
- When used, the prescaler should be assigned to the timer/counter by clearing the PSA bit of the OPTION_REG register. The prescaler rate is set by using the PS2-PS0 bits of the same register.
- When using interrupt, the GIE and TMR0IE bits of the INTCON register should be set.

To measure time:

- Reset the TMR0 register or write some well-known value to it.
- Elapsed time (in microseconds when using quartz 4MHz) is measured by reading the TMR0 register.
- The flag bit TMR0IF of the INTCON register is automatically set every time the TMR0 regis-

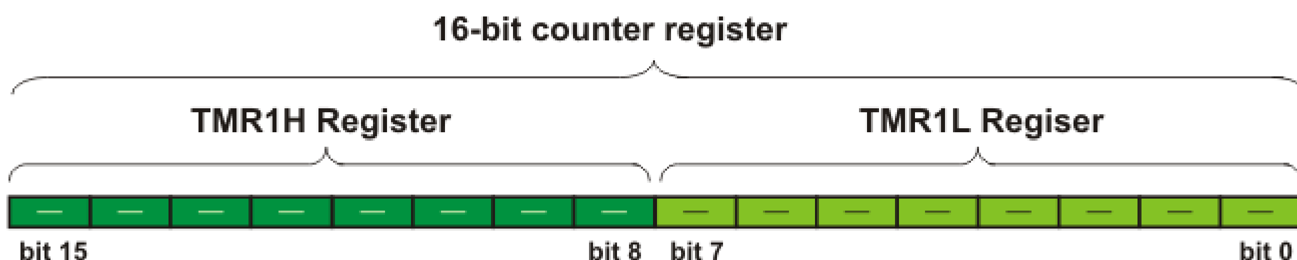
ter overflows. If enabled, an interrupt occurs.

To count pulses:

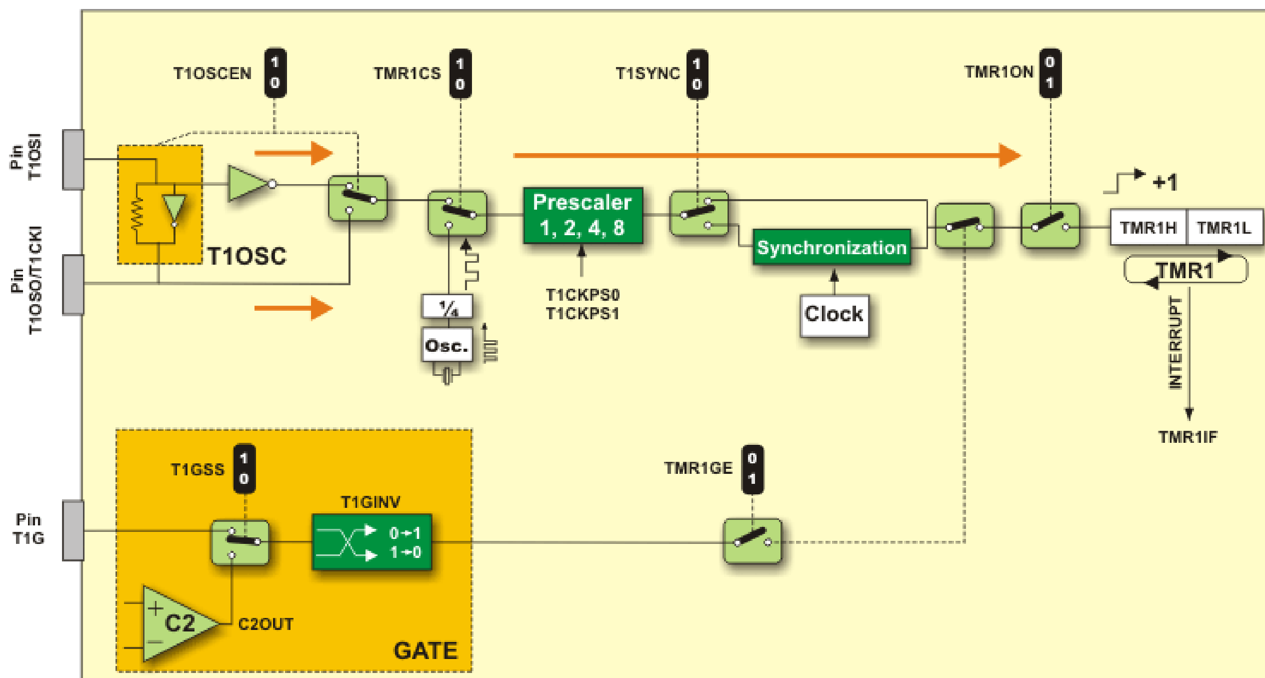
- The polarity of pulses are to be counted is selected on the RA4 pin are selected by the TOSE bit of the OPTION register (T0SE: 0=positive, 1=negative pulses).
- Number of pulses may be read from the TMR0 register. The prescaler and interrupt are used in the same way as in timer mode.

Timer 1 | TMR1 Module

Timer TMR1 module is a 16-bit timer/counter, which means that it consists of two 8 bit registers (TMR1L and TMR1H). Because of that, it can count up 65535 pulses in a single cycle, i.e. before the counting starts from zero.



Similar to the timer TMR0, these registers can be read or written at any moment. In case overflow occurs, an interrupt is generated. The timer TMR1 module may operate in one of two basic modes- as a timer or a counter. However, unlike the timer TMR0, each of these modules has additional functions. Bits of the T1CON register are in control of the operation of the timer TMR1.

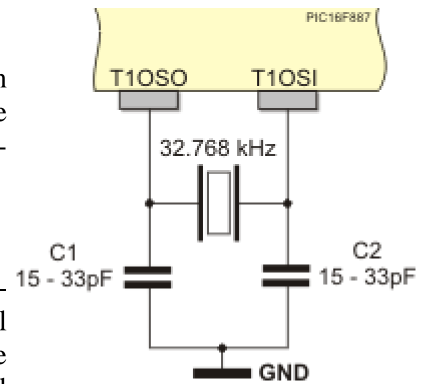


Timer TMR1 Oscillator

RC0/T1OSO and RC1/T1OSI pins are used to register pulses coming from peripheral electronics, but also have additional function. As seen in figure, they are simultaneously configured as both input (pin RC1) and output (pin RC0) of the additional LP quartz oscillator (low power).

This additional circuit is primarily designed for operating at low frequencies (up to 200 KHz), more precisely, for using 32,768 KHz quartz crystal. Such crystal is used in quartz watches because it is easy to ob-

tain one-second-long pulses by simple dividing this frequency.



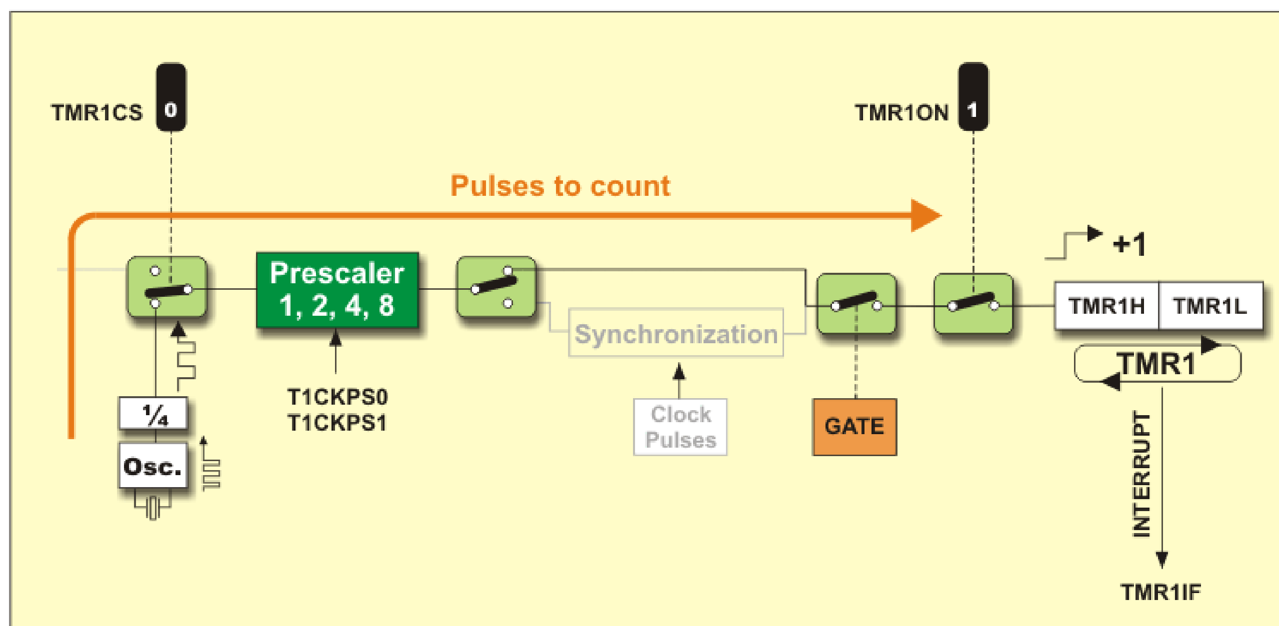
Since this oscillator does not depend on internal clock, it can operate even in *sleep* mode. It is enabled by setting the T1OSCEN control bit of the T1CON register. The user must provide a software time delay (a few milliseconds) to ensure proper oscillator start-up.

Timer TMR1 Gate

Timer 1 gate source is software configurable to be the T1G pin or the output of comparator C2. This gate allows the timer to directly time external events using the logic state on the T1G pin or analog events using the comparator C2 output. Refer to figure above. In order to time a signal duration it is sufficient to enable such gate and count pulses having passed through it.

TMR1 in timer mode

In order to select this mode, it is necessary to clear the TMR1CS bit. After that, the 16-bit register will be incremented on every pulse coming from the internal oscillator. In case 4MHz quartz crystal is in use, it will be incremented every microsecond.

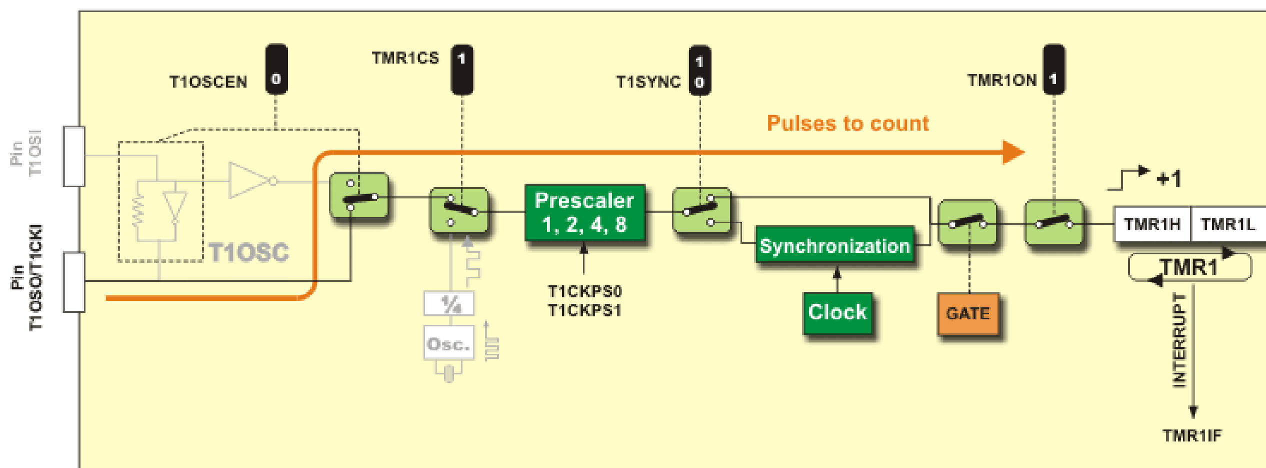


In this mode, the T1SYNC bit does not affect the timer because it counts internal clock pulses. Since the whole electronics uses these pulses, there is no need for synchronization. The microcontroller's clock oscillator does not run during sleep mode so the timer register overflow cannot cause any interrupt if internal clock is used.

TMR1 in counter mode

Timer TMR1 starts to operate as a counter by setting the TMR1CS bit. It means that the timer TMR1 is incremented on the rising edge of the external clock input T1CKI. Besides, if control bit T1SYNC of the T1CON register is cleared, the external clock inputs will be synchronized on their way to the TMR1 register. In other words, the timer TMR1 is synchronized to the microcontroller system clock and called a synchronous counter therefore.

When the microcontroller ,operating in this way, is set in *sleep* mode, the TMR1H and TMR1L timer registers are not incremented even though clock pulses appear on input pins. Simply, since the microcontroller system clock does not run in this mode, there are no clock inputs to use for synchronization. However, the prescaler will continue to run if there are clock pulses on the pins since it is just a simple frequency divider.



Timer TMR1 T1CON Register

T1CON register is 8 bit register controlling the functionality of TMR1.

1: T1CON: TIMER1 CONTROL REGISTER

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

bit 7 **RD16:** 16-bit Read/Write Mode Enable bit
- 1 = Enables register Read/Write of Timer1 in one 16-bit operation
- 0 = Enables register Read/Write of Timer1 in two 8-bit operations

bit 6 **Unimplemented:** Read as '0'

bit 5-4 **T1CKPS1:T1CKPS0**: Timer1 Input Clock Prescale Select bits
- 11 = 1:8 Prescale value
- 10 = 1:4 Prescale value
- 01 = 1:2 Prescale value
- 00 = 1:1 Prescale value

bit 3 **T1OSCEN:** Timer1 Oscillator Enable bit
- 1 = Timer1 Oscillator is enabled
- 0 = Timer1 Oscillator is shut-off

The oscillator inverter and feedback resistor are turned off to eliminate power drain.

bit 2 **T1SYNC:** Timer1 External Clock Input Synchronization Select bit

When TMR1CS = 1:
- 1 = Do not synchronize external clock input
- 0 = Synchronize external clock input

When TMR1CS = 0:

This bit is ignored. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS:** Timer1 Clock Source Select bit
- 1 = External clock from pin RC0/T1OSO/T13CKI (on the rising edge)
- 0 = Internal clock (FOSC/4)

bit 0 **TMR1ON:** Timer1 On bit
- 1 = Enables Timer1
- 0 = Stops Timer1

Well enough theory has been said about timers, we will make some good projects, in projects section using these timers. However here we will be giving some basic examples, as how to use them and how to manage their interrupts.

Our first example will use TMR0, to display the TMR0 register on LCD.

This example first defines the bits of T0CON register as meaningful names, so that it is not confusing to use bit numbers in the rest of program. Whenever you are going to address various registers in your program and their bits, it is a good practice to assign them symbolic names, and use those names in your program. This has two advantages, first the code becomes more reader friendly, and you have to look into the data

sheet only a few times, secondly you have referred to a bit several times in your program, and later you realize its not bit 0, its bit 5. you will not have to make changes at all references, but only to change the new value in symbol definition.

```
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Symbol T0E T0CON.7
Symbol T0CS T0CON.5
Symbol T0PSA T0CON.3
Symbol T0PS2 T0CON.2
Symbol T0PS1 T0CON.1
Symbol T0PS0 T0CON.0
T0CS=0  'internal clock 20MHz/4
T0PSA=0 'Enable pre scaling
T0PS0=0 'Prescaler 000 = 1:2
T0PS1=0
T0PS2=0
T0E=1   'Enable TMR0
Print Cls
Print At 1,1,"TMR0 PSA 000"
loop:
Print At 2,1, Dec   TMR0L
GoTo loop
```

After defining the symbols, we have assigned various values to the T0CON register a prescaler of 000 means 1:2 ratio, this means that the TMR0 will receive clock signals at half of internal clock. So the TMR0L register will get 0 very quickly, indeed you will not be able to see any value. To slow down the procedure, enable a pre scalar of 111, this will cause every 256th pulse to increment the counter, however this still will be too fast to be captured on LCD.

Anyway the objective of this example was to show how you can use the TMR0 and its value stored in TMR0L output register.

> In PIC18F452 TMR0 is also 16 bit Timer, consisting of TMR0H and TMR0L registers. This can be configured to operate as 16 bit or 8 bit timer by setting appropriate bits in T0CON register.

Try T0E=0 and you will notice that timer stops counting the internal events.

Now what is the use of all this. This timer can be used to measure the event duration. Although Proton Basic has PulsIN command, but lets have a look at this feature. As our PIC lab-II is running at 40MHz the internal frequency is F/4 = 5MHz. Therefore the timer module is receiving 5 x 10^6 pulses per second or the counter is incremented by 1 in 2us thus it can measure the incoming pulse of this small duration. The trick lies, in noting the value of register in a variable, and noting the value again at end of event, the difference in two will give you a time of event in multiples of 2us.

What if the event duration is long, you can increase the pre-scalar, and then multiply the counted values with prescaler dividend to get the actual number of clocks. Even more prolonged, event ! The counter after reaching its maximum value of 255 will reset to 0. So in that case just reading the TMR0L register will be erroneous. Either a 16 bit timer mode be used, or consider another technique. Whenever the TMR0 will overflow it will generate an interrupt signal, if the interrupt on TMR0 is enabled the interrupt routine will be fired, in this routine you count the number of overflows. At the end of event the overflows are multiplied by 256 and the present value of TMR0L is added to give the total number of counts. This is additionally adjusted for pre-scalar to give the timing of event.

Now lets say we want an LED to blink at a rate of 1 per second using TMR0 interrupt. Now we know that the clock signals will be 5000000 per second, if we use a pre scalar of 1:256 this will become, 19532 per second the interrupt will fire on every overflow of 256, so 76 interrupts will take place per second. We shall count the interrupts in a variable, and if the variable has reached 76, will toggle the LED, and reset the variable to 0

```
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Symbol T0E T0CON.7
Symbol T0CS T0CON.5
Symbol T0PSA T0CON.3
```

```
Symbol T0PS2 T0CON.2
Symbol T0PS1 T0CON.1
Symbol T0PS0 T0CON.0

Symbol GIE INTCON.7
Symbol TMR0IE INTCON.5
Symbol TMR0IF INTCON.2

Symbol LED PORTC.0
T0CS=0  'internal clock 20MHz/4
T0PSA=0 'Enable pre scaling
T0PS0=1 'Prescaler 000 = 1:256
T0PS1=1
T0PS2=1
T0E=1   'Enable TMR0
Dim x As Byte   ' to count interrupts
Output PORTC
PORTC=0
on_interrupt GoTo jingle
GoTo start
jingle:
x=x+1
If x=76 Then
        Toggle LED
        x=0
EndIf
TMR0IF=0
Context Restore

start:
TMR0IE=1    'enable TMR0 Interrupt
GIE=1
loop:
GoTo loop
```

Notice in this program there is an endless loop, in which the processor is busy all the times, the timer 0 is counting internal pulses and causing an interrupt to take place 76 times a second. Thus our processor is performing two tasks at a time, independent of each other, this is called **Multi-Tasking.** All higher processors like Pentium in your PC, does the same thing to handle multiple events at the same time. You can think of various things to do with this technique. So using timers with interrupts are the basis for making multi-tasking programs and operating systems.

```
jingle:
x=x+1
If x=38 Then
    Toggle LED1
EndIf
If x=76 Then
        Toggle LED0
        x=0
EndIf
TMR0IF=0
Context Restore
```

We shown here only a slightly modified interrupt routine, here two LEDs have been defined, and the toggling rate of each is different. Led 0 is toggling at rate of 1/second and led 1 at 2/second. So we have three tasks running, one your main loop, and other two in background.

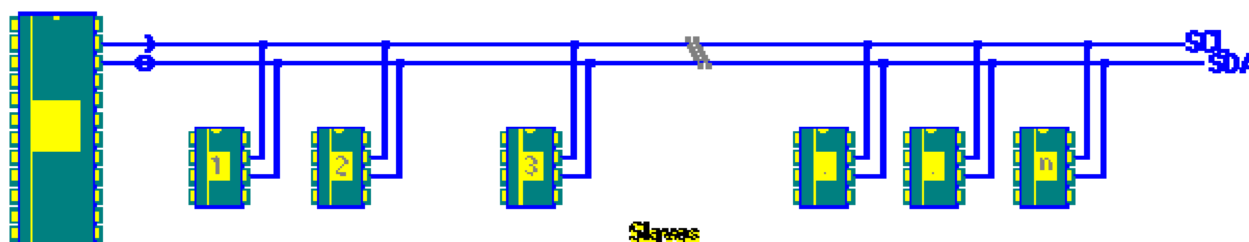
Chapter 19

I^2C Communication

Modern electronics is based upon modular technology. This means that more and more devices are being made just like objects, in order to make a new device which should have certain features you do not have to make everything yourself, just like integrated circuits, all you need to know is their function and pins. Similarly complete devices, or even devices packed within an IC are now available for general purpose usage. When many such devices are present on a board or project they frequently need to communicate and transfer data among each other. You may think that USART can be used as communication, well fair enough, but this will tie up at least 1 I/O line per device.

To address this problem and to ensure reliable communication Philips® came up with a solution which they call Inter - Integrated Circuit Communication. The system has a predefined protocol, which is a set of rules that every device will follow. The communication takes place over 2-wires, however up to 7 different devices can be connected to the same two wire– Bus, called I2C Bus. In this section we will learn and explore how the I2C based devices work, and how to make circuits based upon these devices.

The I2C design consists of a Master device that is controlling the entire communication and a set of Slave devices which are responding to the needs of master. In common scenario Master device is your microcontroller and slave devices are various like EEPROM, RTC, Ultrasonic Ranger, Temperature sensor etc. The I2C Bus consists of two lines, called SDA and SCL. SDA is for data and SCL is fir synchronized clock sig-



nals. By design these devices have open collectors for these two lines therefore two 10K pull up resistors must be placed on these lines. The device that initiates the data transfer process is called Master device. The Bus can contain multiple Masters, but we will consider one Master and many slave system. Now how will the slave know that a particular message is for it? The society which holds the I2C rights issues a unique address for each device type. No two devices on the same Bus should have the same address. The Master uses this address to inform the device, that next instructions are for this device.

Standard communication speed is 100 Kbits/s but certain slower devices communicate at 10Kbits/s some recent devices are using 400 Kbits/s speed. Higher speed devices are being made but they are not in the range of microcontrollers.

I2C Communication protocol

The communication is started by Master, by what is arbitrarily called a START condition. This is a sequence of taking SDA and SCL lines High and low in a particular order. The start is followed by transmis-

sion of device address, which is 7 bit number followed by a write bit. The device with that address sends an

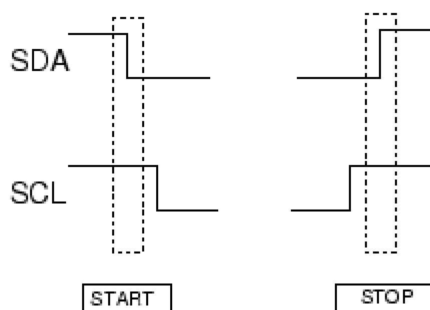
Acknowledgment that yes, its present and it is ready. This is followed by one data byte at a time, followed by Acknowledgment from device on each byte, finally when Master wants to close the communication it sends a STOP sequence. Following which the device is released, and Bus is also released so that a new communication with another device be setup.

Device addresses

Each device you use on the I2C bus must have a unique address. For some devices e.g. serial memory you can set the lower address bits using input pins on the device others have a fixed internal address setting e.g. a real time clock DS1307. You can put several memory devices on the same IC bus by using a different address for each.

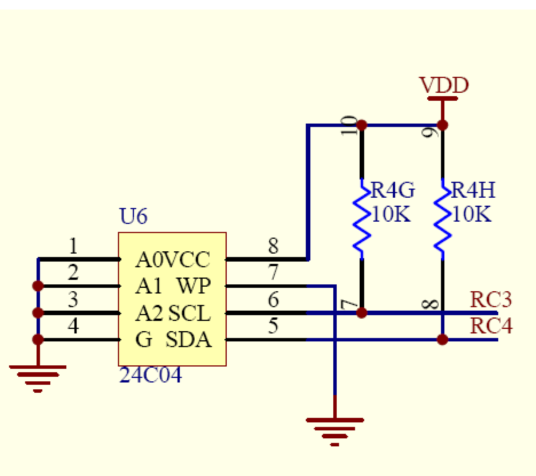
Note: The maximum number of devices is limited by the number of available addresses and by the total bus capacitance (maximum 400pF).

Thus in order to use a commercially available slave device you must know its slave address.



We are going to use an I2C device commonly used in electronics projects. This is I2C EEPROM. PIC Lab-II has on board I2C EEPROM socket, which comes with 24C08. This 8 pin chip contains 8K EEPROM. A wide range of devices are available, you can place another if required.

As you have seen I2C communication is basically an art of making the various lines high and low, this can be implemented via software on any two lines. Since this is so commonly used protocol PIC microcontrollers have a built-in hardware module which takes care of the entire process, and our requirement is dramatically reduced as far as software coding is concerned.



If you look at the pins of 18F452 the RC3 is SCL and RC4 is SDA. The board contains two pull up resistors on these lines and connected directly to I2C EEPROM. Moreover the same Bus is also available as header for expansion to other devices. However do consider that expansion with long cables will result in increased capacitance and failure of bus.

The internal address if I2C EEPROM chips 24cXXX is %1010xxx next 3 bits are the chip address set on your hardware. The 24Cxxx chips have three lines for configuring this address thus you can place up to 7 EEPROM devices provided their chip addresses are set different. In the circuit shown PIC Lab-II has set this address to 000. thus the complete 7 bit address of this device would be %1010000, the write command is 0 and read command is 1 as last bit of the address. So To write data we send %10100000 the last 0 is the bit to inform a write, and to read data we issue %10100001.

Now we shall write a program to write some data into the external EEPROM, using I2C communication. Please make sure you understand, that there is also internal EEPROM within the microcontroller. External EEPROM is required if large amount of data is to be kept.

Proton Basic therefore has two sets of commands to deal with I2C communication. One set uses software to do the entire job, and gives you the liberty to use any two digital I/O lines. The second set uses hardware module, thus produces less code, however requires you to use only designated pins for SDA and SDL.

We will first use the software method, even though we will be using the same pins as for hardware, but the hardware module will not be actually used.

In this mode or method we have to inform the compiler as to which of the I/O lines will be used for I2C Communication.

The program shown here first defines the SDA and SCL pins, these can be any pins, you want to connect

```
'I2C EEPROM Write / Read
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
SDA_PIN PORTC.4
SCL_PIN PORTC.3
Dim x As Byte
x=100
BStart
BusOut %10100000,0,[x]
BStop
Print Cls
Print At 1,1,"Write OK"
DelayMS 2000
Print Cls
X=0
BStart
x=BusIn %10100001,0
BStop
Print At 1,1,"X=", Dec x
End
```

your device. Then it has a variable x, and stores a value of 100 the Bstart command initiates the Master to issue a Start condition, the BusOut command sends the address of device, followed by the memory address where data is to be written and then the data x. the address byte has 1010 as device address and 000 as the chip address, last 0 is the write instruction. Then the Bstop breaks the connection. Next section again opens the connection and reads the data back and display it on screen.

If you are using some other device like Real time clock (DS1307), temperature sensor (DS1624) etc read their data sheet first to know which addresses contain the registers for particular commands.

Infra red Remote Controls

PIC Lab-II features an on-board infra-red remote control sensor. This is a general purpose I-R sensor, however unlike commonly used IR sensors, it detects only IR signals which are modulated at 38KHz. This eliminates the interference from surrounding Infra red signals. All IR remote controls use 38KHz modulated signals to transmit their data. However they all vary in specifications and protocols of sending data. Sony remote controls have relatively better documentation available to understand their data. The data is serial and digital, but does not comply with USART or I2C protocols etc. Actually what is done, the width of pulses is measured with Pulsin command, and then each pulse length is encoded as 0 or 1, the decoded bit is then written into the least significant bit of a variable, and entire byte or word is shifted to left. In this way the data transmitted by remote controls is accepted. You will have to do some research to decode these data. As far as sensing a remote control input is concerned its simple.

```
Device=18F452
XTAL=20
ALL_DIGITAL true
Symbol IR = PORTA.3
ALL_DIGITAL true
Symbol LED PORTC.0
Input IR
Output PORTC
PORTC=0
loop:
If IR=0 Then
    Toggle LED
    DelayMS 200
EndIf
GoTo loop
```

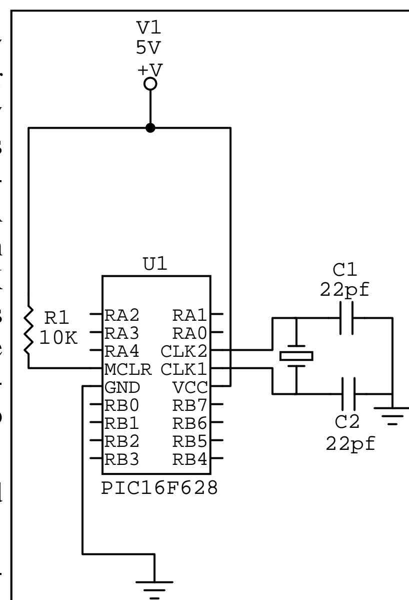
Use any IR remote, pressing any button will toggle led 0. Make sure Dip Switch 3 for IR sensor is On.

You can make your own custom remote control, by using any microcontroller, and an IR Led on output. Send serial data, modulated at 38KHz.

Appendix 1
Basic Electronics

The PIC Hardware

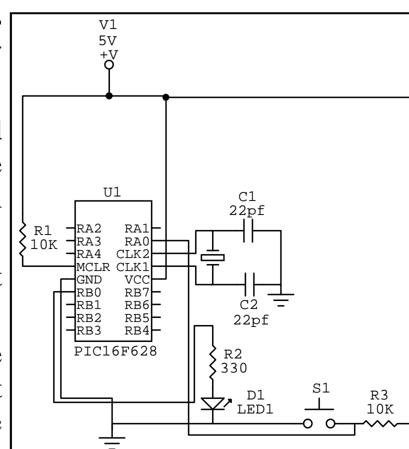
Well so far you have gained an insight about the various features of 1PIC microcontroller. Now is the time to understand how to use it in a project. In order to experiment with PIC Lab you don't need to do that, but it is an advantage to know, how the microcontroller based circuit is made. After all you are going to design your devices, using these microcontrollers and if you don't know how to put the thing into your circuit, of what use will be all this exercise. Certainly you can not use the PIC Lab motherboard in your every application.!



Basic circuit drawing is shown here. PIC microcontroller needs only four components to start functioning. A crystal oscillator of your choice is connected between CLK1 and CLK2. these pins will vary among various PICs, so always consult data sheet to locate pins. It is customary to talk in terms of pin names, rather than numbers in microcontrollers. The Vlk pins are grounded using 22pf capacitors. A 10K resistor is connected to Vcc at MCLR. Connecting a push switch to ground will provide a convenient Reset circuit. (don't remove 10K pull up) so that when switch is open MCLR gets Vcc. That's it. This is the basic circuit, and rest of the pins are all I/O you are free to use them, in whatever way you like. This circuit will run whatever program has been loaded into it. Since you will have to take the IC to programmer, try putting it in a socket.

Now lets see if we can make a blinking LED connected to RB0, and an input switch connected to RA0.

Although the PIC pins can both source and sink the load. It is customary to use them as source, so that a '1' on pin drives the load. These pins can give sufficient current to handle the load of an LED, still it is better to protect the pin from overloading by limiting the current flow using a current limiting resistor, usually a 330 Ω or 680 Ω .



A switch can be connected to vcc, giving '1' on the pin or connected to ground giving '0' when pushed. We prefer the second form, and the pin is connected using a 10K resistor to vcc to give logical '1' when switch is open and give logical '0' when switch is pressed.

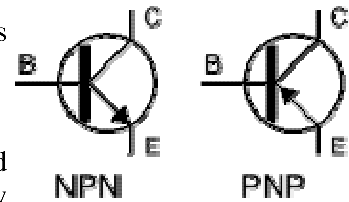
PIC Lab uses switches in this form, so when a switch is pressed, it will deliver a logical '0' to the program.

For driving low current circuits, like other digital devices, pins can be used directly. To drive a transistor, it is customary to use a current limiting resistor in series, like 2.2K with the base of transistor. The transistor can then be used to drive heavy loads, like switching a relay On.

Using Transistors (Basic Electronics)

There are two types of standard transistors, **NPN** and **PNP**, with different circuit symbols. The letters refer to the layers of semiconductor material used to make the transistor. Most transistors used today are NPN because this is the easiest type to make from silicon. We are going to talk about NPN transistors and if you are new to electronics it is best to start by learning how to use these first.

The leads are labeled **base** (B), **collector** (C) and **emitter** (E). These terms refer to the internal operation of

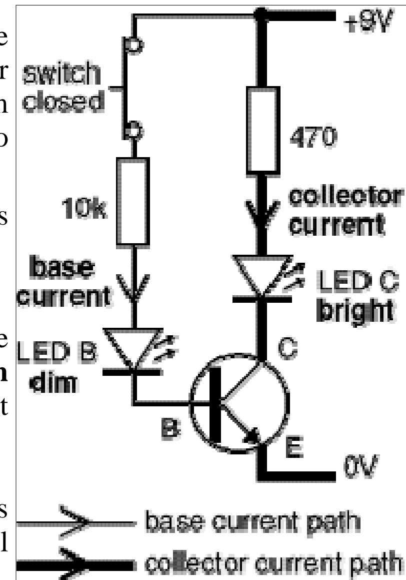


a transistor but they are not much help in understanding how a transistor is used, so just treat them as labels!

Transistor currents

The diagram shows the two current paths through a transistor. You can build this circuit with two standard 5mm red LEDs and any general purpose low power NPN transistor (BC108, BC182 or BC548 for example).

The small **base current** controls the larger **collector current**.



When the switch is closed a small current flows into the base (B) of the transistor. It is just enough to make LED B glow dimly. The transistor amplifies this small current to allow a larger current to flow through from its collector (C) to its emitter (E). This collector current is large enough to make LED C light brightly.

When the switch is open no base current flows, so the transistor switches off the collector current. Both LEDs are off.

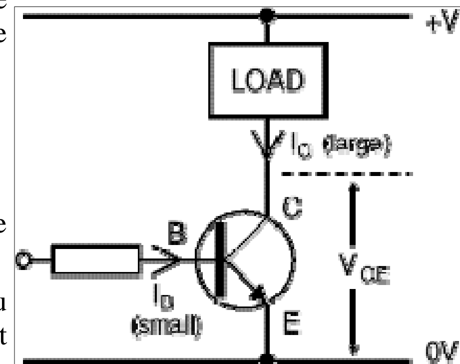
A transistor amplifies current and can be used as a switch.

This arrangement where the emitter (E) is in the controlling circuit (base current) and in the controlled circuit (collector current) is called **common emitter mode**. It is the most widely used arrangement for transistors so it is the one to learn first.

Thus if base of transistor is given a small current via a resistance in series and connected to microcontroller pin, a logical '1' on microcontroller will turn the transistor on, and a logical '0' will turn it off.

Using a transistor as a switch

When a transistor is used as a switch it must be either **OFF** or **fully ON**. In the fully ON state the voltage V_{CE} across the transistor is almost zero and the transistor is said to be **saturated** because it cannot pass any more collector current Ic. The output device switched by the transistor is usually called the 'load'.



The power developed in a switching transistor is very small:

In the **OFF** state: power = Ic × V_{CE}, but Ic = 0, so the power is zero.

In the **full ON** state: power = Ic × V_{CE}, but V_{CE} = 0 (almost), so the power is very small.

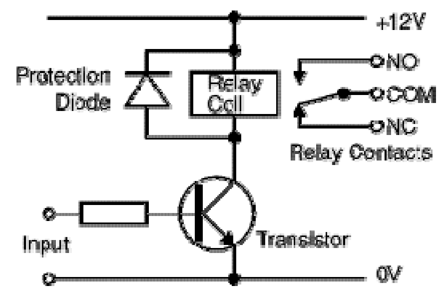
This means that the transistor should not become hot in use and you do not need to consider its maximum power rating. The important ratings in switching circuits are the **maximum collector current Ic (max)** and the **minimum current gain h_{FE}(min)**. The transistor's voltage ratings may be ignored unless you are using a supply voltage of more than about 15V. For information about the operation of a transistor please see the functional model above.

Protection diode

If the load is a **motor**, **relay** or **solenoid** (or any other device with a coil) a diode must be connected across the load to protect the transistor (and chip) from damage when the load is switched off. The diagram shows how this is connected 'backwards' so that it will normally NOT conduct. Conduction only occurs when the load is switched off, at this moment current tries to continue flowing through the coil and it is harmlessly diverted through the diode. Without the diode no current could flow and the coil would produce a damaging high voltage 'spike' in its attempt to keep the current flowing.

When to use a Relay

Transistors cannot switch AC or high voltages (such as mains electricity) and they are not usually a good choice for switching large currents (> 5A). In these cases a relay will be needed, but note that a low power



transistor may still be needed to switch the current for the relay's coil!

Advantages of relays:

Relays can switch **AC and DC**, transistors can only switch DC.

Relays can switch **high voltages**, transistors cannot.

Relays are a better choice for switching **large currents** (> 5A).

Relays can switch **many contacts** at once.

Disadvantages of relays:

Relays are **bulkier** than transistors for switching small currents.

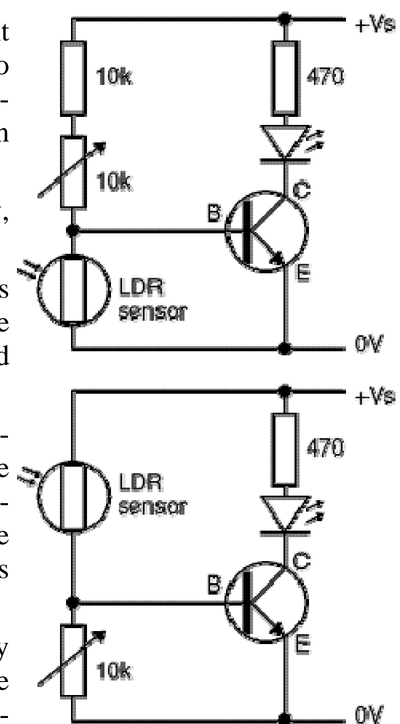
Relays **cannot switch rapidly**, transistors can switch many times per second.

Relays **use more power** due to the current flowing through their coil.

Relays **require more current than many chips can provide**, so a low power transistor may be needed to switch the current for the relay's coil.

Using a transistor switch with sensors

The circuit diagram shows an LDR (light sensor) connected so that the LED lights when the LDR is in darkness. The variable resistor adjusts the brightness at which the transistor switches on and off. Any general purpose low power transistor can be used in this circuit.



The 10k fixed resistor protects the transistor from excessive base current (which will destroy it) when the variable resistor is reduced to zero. To make this circuit switch at a suitable brightness you may need to experiment with different values for the fixed resistor, but it must not be less than 1k.

If the transistor is switching a load with a coil, such as a motor or relay, remember to add a protection diode across the load.

The switching action can be inverted, so the LED lights when the LDR is brightly lit, by swapping the LDR and variable resistor. In this case the fixed resistor can be omitted because the LDR resistance cannot be reduced to zero.

Note that the switching action of this circuit is not particularly good because there will be an intermediate brightness when the transistor will be **partly on** (not saturated). In this state the transistor is in danger of overheating unless it is switching a small current. There is no problem with the small LED current, but the larger current for a lamp, motor or relay is likely to cause overheating.

Other sensors, such as a thermistor, can be used with this circuit, but they may require a different variable resistor. You can calculate an approximate value for the variable resistor (Rv) by using a multimeter to find the minimum and maximum values of the sensor's resistance (Rmin and Rmax):

Variable resistor, Rv = square root of (Rmin × Rmax)

For example an LDR: Rmin = 100, Rmax = 1M, so Rv = square root of (100 × 1M) = 10k.

You can make a much better switching circuit with sensors connected to a suitable IC (chip). The switching action will be much sharper with no partly on state.

LED lights when the LDR is **dark**

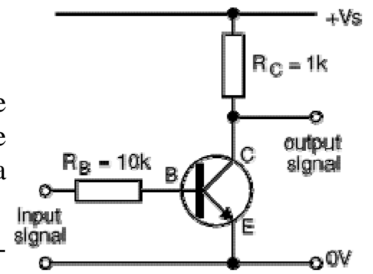
LED lights when the LDR is **bright**

A transistor inverter (NOT gate)

Inverters (NOT gates) are available on logic chips but if you only require one inverter it is usually better to use this circuit. The output signal (voltage) is the inverse of the input signal:

When the input is high (+Vs) the output is low (0V).

When the input is low (0V) the output is high (+Vs).



Any general purpose low power NPN transistor can be used. For general use R_B = 10k and R_C = 1k, then the inverter output can be connected to a device with an input impedance (resistance) of at least 10k such as a logic chip or a 555 timer (trigger and reset inputs).

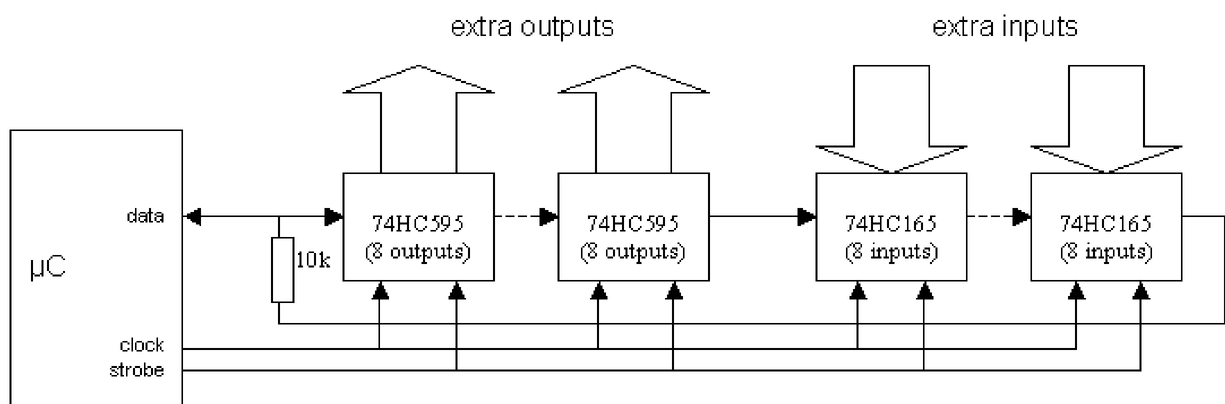
If you are connecting the inverter to a CMOS logic chip input (very high impedance) you can increase R_B to 100k and R_C to 10k, this will reduce the current used by the inverter.

Appendix 2

Expanding Microcontroller I/O Lines

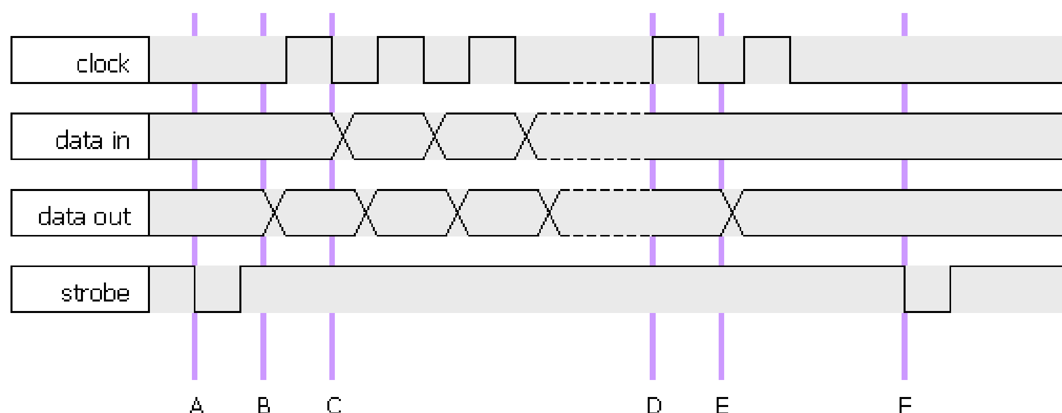
The most precious resource on a microcontroller is its I/O lines. Today's I/O line hungry applications require more and more lines from microcontrollers. For this reason many microcontrollers with more and more I/O lines are available. However remaining confined to your existing microcontroller, you can expand its I/O lines, by using Serial-In parallel Out, or parallel In serial Out Shift registers.

Shift registers like 74HC595 require three I/O lines, one for serial data, one for clock signals to shift data and one for latching the data from internal registers to output lines of shift registers. 74HC595 is an 8 bit shift register, which means you get 8 lines (for Output) by sacrificing three lines of microcontroller, a gain of 5 lines. However the most beautiful thing is that they can be chained together, in definitely, so that you can shift out 16, 24 or even 32 bits of data, just by using three I/O lines.



2.2. How to drive it

The circuit description covers use of the 74HC595. Use of the 74HC4094 (which may be cheaper and easier to find) is covered later in this document. First the strobe line is dipped low and back high again. This latches all the inputs into the 74HC165 input shift registers. Then the clocking begins. With each clock pulse, the data line is set to an output, and the appropriate data bit is presented on the line to be clocked into the output shift register. On the same clock pulse, the input shift register presents its next data bit to the mi-



crocontroller data pin. The data pin is set to an input, and this data bit read.

The number of clock pulses required is the larger of the number of inputs and the number of outputs. After this number of clock pulses, all the required output states have been shifted into position in the 74HC597 output shift registers, and another dipping of the "strobe" line is performed to set these states on the shift register output pins.

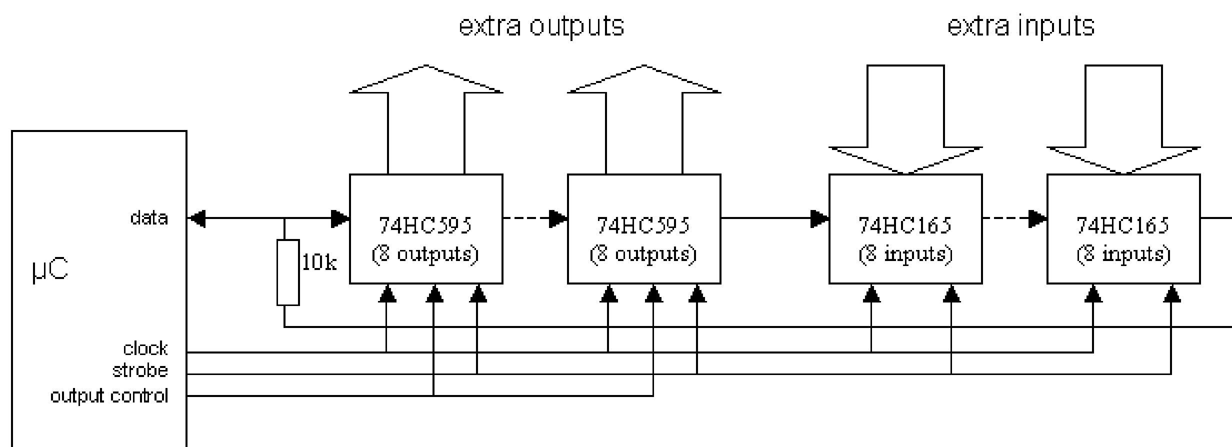
The points in this diagram are:

- A: STROBE is pulsed low to latch inputs.
- B: DATA line is set to an output, and appropriate data placed on it. CLOCK is pulsed high to shift this data bit into the chain, and get next bit from the chain.
- C: DATA line is set to an input, and the data bit is read from the chain.
- D: In this example there are more outputs than inputs, so from this point, the rest of the outputs are clocked out.
- E: This is the rest of the outputs being clocked out.
- F: Finally, the STROBE line is pulsed again to latch the outputs onto the output shift register output pins.

You can make the chain any length you want, with as many output stages or input stages as required. It may be all input stages, or it may be all output stages too. There are constants in the code where the dimensions of the shift register are defined, and they control how the software drives it.

2.3. Protected outputs

The outputs will be in an undefined state when the circuit is powered up. This may be dangerous if the lines that drive the output must not be turned on unless special circumstances are observed. Therefore, the output shift registers chosen have three-state outputs. This means that their outputs can be turned off (not high or low but effectively open-circuit). They will then need to be pulled high or low as required by the circuit that the output is connected to. Using this three-state option requires a further output line from the microcontroller as shown below:



For practical project on shift registers see our 8 x 32 Matrix LED project, which uses 4 shift registers.

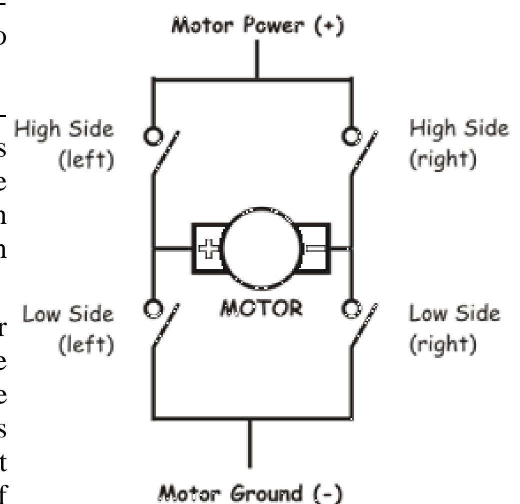
Appendix 3
H-Bridge and DC Motors

Introduction

A number of web sites talk about H-bridges, they are a topic of great discussion in robotics clubs and they are the bane of many robotics hobbyists. I periodically chime in on discussions about them, and while not an expert by a long shot I've built a few over the years. Further, they were one of my personal stumbling blocks when I was first getting into robotics. This section is devoted to the theory and practice of building H-bridges for controlling brushed DC motors (the most common kind you will find in hobby robotics ...) .

Basic Theory

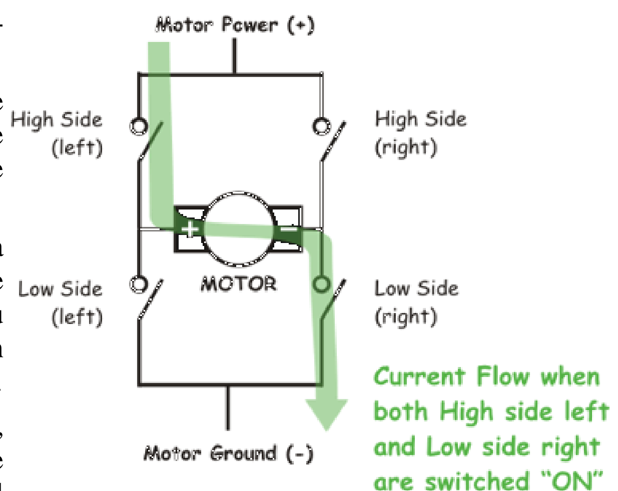
Let's start with the name, H-bridge. Sometimes called a "full bridge" the H-bridge is so named because it has four switching elements at the "corners" of the H and the motor forms the cross bar. The basic bridge is shown in the figure to the right.



Of course the letter H doesn't have the top and bottom joined together, but hopefully the picture is clear. The key fact to note is that there are, in theory, four switching elements within the bridge. These four elements are often called, high side left, high side right, low side right, and low side left (when traversing in clockwise order).

The switches are turned on in pairs, either high left and lower right, or lower left and high right, but never both switches on the same "side" of the bridge. If both switches on one side of a bridge are turned on it creates a short circuit between the battery plus and battery minus terminals. This phenomena is called shoot through in the Switch-Mode Power Supply (SMPS) literature. If the bridge is sufficiently powerful it will absorb that load and your batteries will simply drain quickly. Usually however the switches in question melt.

To power the motor, you turn on two switches that are diagonally opposed. In the picture to the right, imagine that the high side left and low side right switches are turned on. The current flow is shown in green.



The current flows and the motor begins to turn in a "positive" direction. What happens if you turn on the high side right and low side left switches? You guessed it, current flows the other direction through the motor and the motor turns in the opposite direction.

Pretty simple stuff right? Actually it is just that simple, the tricky part comes in when you decide what to use for switches. Anything that can carry a current will work, from four SPST switches, one DPDT switch, relays, transistors, to enhancement mode power MOSFETs.

One more topic in the basic theory section, quadrants. If each switch can be controlled independently then you can do some interesting things with the bridge, some folks call such a bridge a "four quadrant device" (4QD get it?). If you built it out of a single DPDT relay, you can really only control forward or reverse. You can build a small truth table that tells you for each of the switch's states, what the bridge will do. As each switch has one of two states, and there are four switches, there are 16 possible states. However,

since any state that turns both switches on one side on is "bad" (smoke issues forth), there are in fact only four useful states (the four quadrants) where the transistors are turned on.

H-Bridge Driver Chips

A few driver chips are available, which contain the H-bridge based upon heavy duty switching transistors. One such chip is L298 which contains 2 drivers for two DC motors.

Appendix 4

Stepper Motors

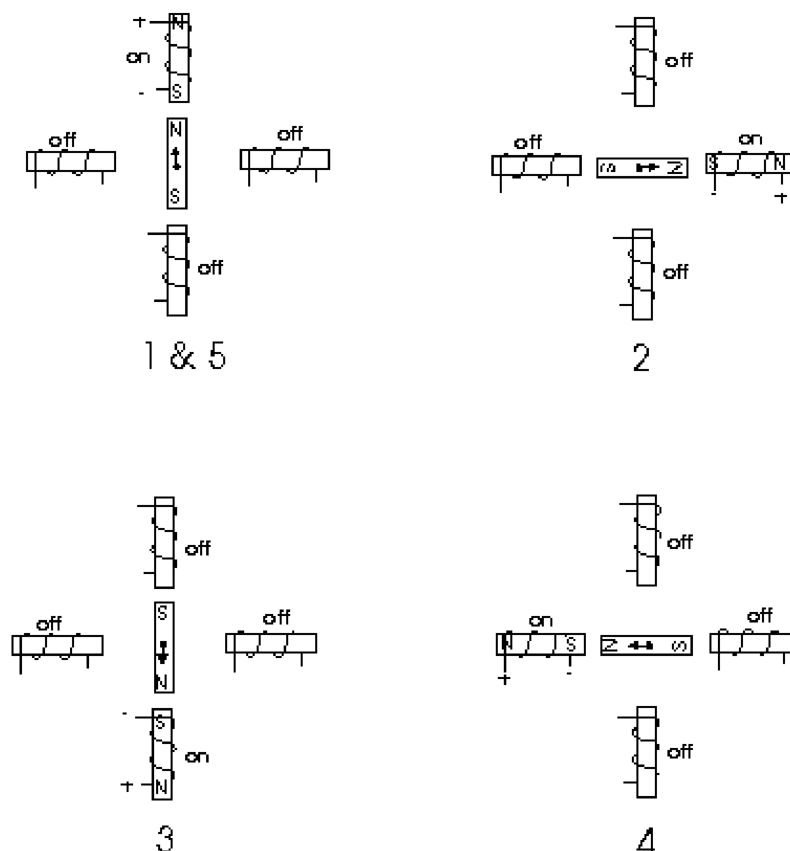
A **stepper motor** is a brushless, synchronous electric motor that can divide a full rotation into a large number of steps. The motor's position can be controlled precisely, without any feedback mechanism.

Fundamentals of operation

Stepper motors operate much differently from normal DC motors, which rotate when voltage is applied to their terminals. Stepper motors, on the other hand, effectively have multiple "toothed" electromagnets arranged around a central gear-shaped piece of iron. The electromagnets are energized by an external control circuit, such as a microcontroller. To make the motor shaft turn, first one electromagnet is given power, which makes the gear's teeth magnetically attracted to the electromagnet's teeth. When the gear's teeth are thus aligned to the first electromagnet, they are slightly offset from the next electromagnet. So when the next electromagnet is turned on and the first is turned off, the gear rotates slightly to align with the next one, and from there the process is repeated. Each of those slight rotations is called a "step." In that way, the motor can be turned a precise angle.

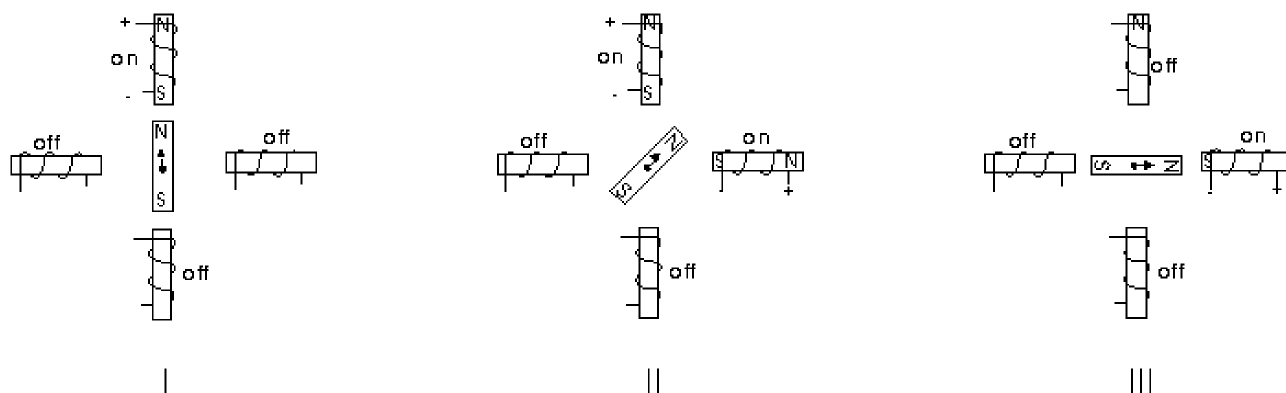
How Stepper Motors Work

Stepper motors consist of a permanent magnet rotating shaft, called the rotor, and electromagnets on the stationary portion that surrounds the motor, called the stator. Fig illustrates one complete rotation of a stepper motor. At position 1, we can see that the rotor is beginning at the upper electromagnet, which is currently active (has voltage applied to it). To move the rotor clockwise (CW), the upper electromagnet is deactivated and the right electromagnet is activated, causing the rotor to move 90 degrees CW, aligning itself with the active magnet. This process is repeated in the same manner at the south and west electromagnets until we once again reach the starting position.



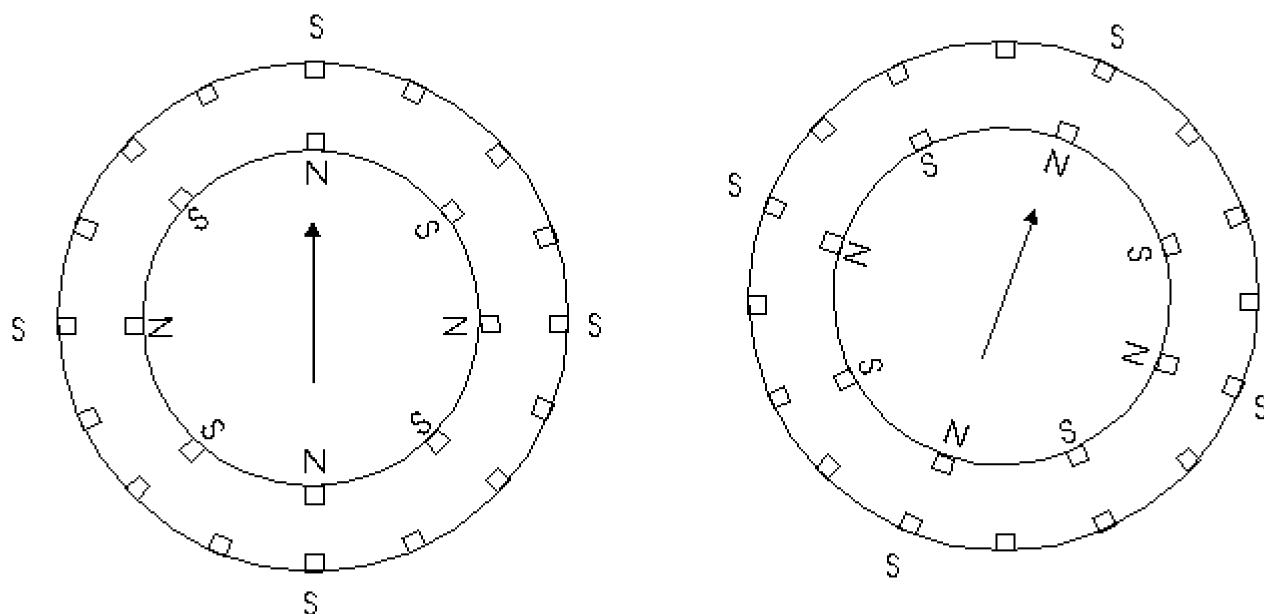
In the above example, we used a motor with a resolution of 90 degrees or demonstration purposes. In reality, this would not be a very practical motor for most applications. The average stepper motor's resolution -- the amount of degrees rotated per pulse -- is much higher than this. For example, a motor with a resolution of 5 degrees would move its rotor 5 degrees per step, thereby requiring 72 pulses (steps) to complete a full 360 degree rotation.

You may double the resolution of some motors by a process known as "half-stepping". Instead of switching the next electromagnet in the rotation on one at a time, with half stepping you turn on both electromagnets, causing an equal attraction between, thereby doubling the resolution. As you can see in Figure 2, in the first position only the upper electromagnet is active, and the rotor is drawn completely to it. In position 2, both the top and right electromagnets are active, causing the rotor to position itself between the two active poles. Finally, in position 3, the top magnet is deactivated and the rotor is drawn all the way right. This process can then be repeated for the entire rotation.



There are several types of stepper motors. 4-wire stepper motors contain only two electromagnets, however the operation is more complicated than those with three or four magnets, because the driving circuit must be able to reverse the current after each step. For our purposes, we will be using a 6-wire motor.

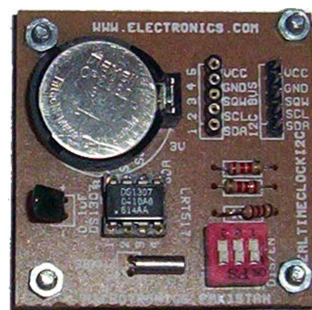
Unlike our example motors which rotated 90 degrees per step, real-world motors employ a series of mini-poles on the stator and rotor to increase resolution. Although this may seem to add more complexity to the process of driving the motors, the operation is identical to the simple 90 degree motor we used in our example. An example of a multipole motor can be seen in Figure 3. In position 1, the north pole of the rotor's perminant magnet is aligned with the south pole of the stator's electromagnet. Note that multiple positions are alligned at once. In position 2, the upper electromagnet is deactivated and the next one to its immediate left is activated, causing the rotor to rotate a precise amount of degrees. In this example, after eight steps the sequence repeats.



Appendix 5

Real Time Clock

Some applications need to keep track of time. Although timing can be achieved by PIC microcontroller, yet when microcontroller is busy in some task, it can not precisely update the system time. Moreover time will be implemented using memory variables, when power is turned off, these variables reset. Most applications that require an on board clock / calendar implement it using a standalone real time chip. There are numerous chips out there which can be used with microcontroller to keep the clock/calendar function. We are going to introduce here a popular chip, DS1307. although you can make the circuit yourself, Microtronics has a smart board, for this chip, that implements it as a stand alone device to be used with any microcontroller.



DS1307 uses I2C communication, and you can use this board on I2C bus, or on any other I/O lines, implementing I2C communication using software routines. We will implement this on PORTB, as the connector on board contains an additional pin called SQW, which is strictly speaking not part of I2C and therefore to connect the board on I2C bus on PIC Lab-II a slightly modified connector has to be made.

GENERAL DESCRIPTION

The DS1307 serial real-time clock (RTC) is a low-power, full binary-coded decimal (BCD) clock/calendar plus 56 bytes of NV SRAM. Address and data are transferred serially through an I2C, bidirectional bus. The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The end of the month date is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24- hour or 12-hour format with AM/PM indicator. The DS1307 has a built-in power-sense circuit that detects power failures and automatically switches to the backup supply. Timekeeping operation continues while the part operates from the backup supply.

DETAILED DESCRIPTION

The DS1307 is a low-power clock/calendar with 56 bytes of battery-backed SRAM. The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The date at the end of the month is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The DS1307 operates as a slave device on the I2C bus. Access is obtained by implementing a START condition and providing a device identification code followed by a register address. Subsequent registers can be accessed sequentially until a STOP condition is executed. When V_{CC} falls below 1.25 x V_{BAT}, the device terminates an access in progress and resets the device address counter. Inputs to the device will not be recognized at this time to prevent erroneous data from being written to the device from an out-of tolerance system. When V_{CC} falls below V_{BAT}, the device switches into a low-current battery-backup mode. Upon power -up, the device switches from battery to V_{CC} when V_{CC} is greater than V_{BAT} +0.2V and recognizes inputs when V_{CC} is greater than 1.25 x V_{BAT}. The block diagram in Figure 1 shows the main elements of the serial RTC.

RTC AND RAM ADDRESS MAP

Table 2 shows the address map for the DS1307 RTC and RAM registers. The RTC registers are located in address locations 00h to 07h. The RAM registers are located in address locations 08h to 3Fh. During a multi byte access, when the address pointer reaches 3Fh, the end of RAM space, it wraps around to location 00h, the beginning of the clock space.

CLOCK AND CALENDAR

The time and calendar information is obtained by reading the appropriate register bytes. Table 2 shows the RTC registers. The time and calendar are set or initialized by writing the appropriate register bytes. The contents of the time and calendar registers are in the BCD format. The day-of-week register increments at

midnight. Values that correspond to the day of week are user-defined but must be sequential (i.e., if 1 equals Sunday, then 2 equals Monday, and so on.) Illogical time and date entries result in undefined operation. Bit 7 of Register 0 is the clock halt (CH) bit. When this bit is set to 1, the oscillator is disabled. When cleared to 0, the oscillator is enabled.

Note that the initial power-on state of all registers is not defined. Therefore, it is important to enable the oscillator (CH bit = 0) during initial configuration.

The DS1307 can be run in either 12-hour or 24-hour mode. Bit 6 of the hours register is defined as the 12-hour or 24-hour mode-select bit. When high, the 12-hour mode is selected. In the 12-hour mode, bit 5 is the AM/PM bit with logic high being PM. In the 24-hour mode, bit 5 is the second 10-hour bit (20 to 23 hours). The hours value must be re-entered whenever the 12/24-hour mode bit is changed. When reading or writing the time and date registers, secondary (user) buffers are used to prevent errors when the internal registers update. When reading the time and date registers, the user buffers are synchronized to the internal registers on any I2C START. The time information is read from these secondary registers while the clock continues to run. This eliminates the need to re-read the registers in case the internal registers update during a read. The divider chain is reset whenever the seconds register is written. Write transfers occur on the I2C acknowledge from the DS1307. Once the divider chain is reset, to avoid rollover issues, the remaining time and date registers must be written within one second.

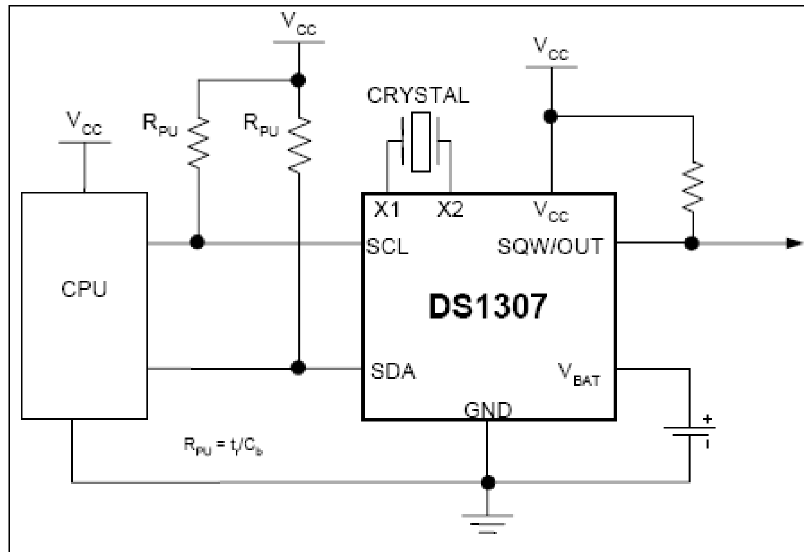
Table 2. Timekeeper Registers

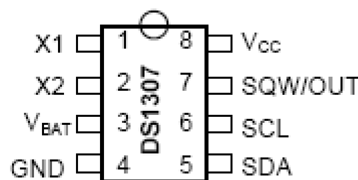
ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00H	CH	10 Seconds			Seconds				Seconds	00–59
01H	0	10 Minutes			Minutes				Minutes	00–59
02H	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/ AM							
03H	0	0	0	0	0	DAY			Day	01–07
04H	0	0	10 Date		Date				Date	01–31
05H	0	0	0	10 Month	Month				Month	01–12
06H	10 Year				Year				Year	00–99
07H	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08H-3FH									RAM 56 x 8	00H–FFH

The DS1307 may operate in the following two modes:

1. **Slave Receiver Mode (Write Mode):** Serial data and clock are received through SDA and SCL. After each byte is received an acknowledge bit is transmitted. START and STOP conditions are recognized as the beginning and end of a serial transfer. Hardware performs address recognition after reception of the slave address and direction bit (see Figure 4). The slave address byte is the first byte received after the master generates the START condition. The slave address byte contains the 7-bit DS1307 address, which is 1101000, followed by the direction bit (R/W), which for a write is 0. After receiving and decoding the slave address byte, the DS1307 outputs an acknowledge on SDA. After the DS1307 acknowledges the slave address + write bit, the master transmits a word address to the DS1307. This sets the register pointer on the DS1307, with the DS1307 acknowledging the transfer. The master can then transmit zero or more bytes of data with the DS1307 acknowledging each byte received. The register pointer automatically increments after each data byte are written. The master will generate a STOP condition to terminate the data write.
2. **Slave Transmitter Mode (Read Mode):** The first byte is received and handled as in the slave receiver mode. However, in this mode, the direction bit will indicate that the transfer direction is reversed. The DS1307 transmits serial data on SDA while the serial clock is input on SCL. START and STOP conditions are recognized as the beginning and end of a serial transfer. The slave address byte is the first byte received after the START condition is generated by the master. The slave address byte contains the 7-bit DS1307 address, which is 1101000, followed by the direction bit (R/W), which is 1 for a read. After

receiving and decoding the slave address the DS1307 outputs an acknowledge on SDA. The DS1307 then begins to transmit data starting with the register address pointed to by the register pointer. If the register pointer is not written to before the initiation of a read mode the first address that is read is the last one stored in the register pointer. The register pointer automatically increments after each byte are read. The DS1307 must receive a Not Acknowledge to end a read.





Writing applications is fairly simple, however keep in mind that the data obtained from registers is in BCD format and not as binary number. Like seconds, say if its 49 seconds, bits 0 to 3 will contain 9 and bits 4,5,6 will contain 4 so you have to extract the numbers from the byte read.

Secondly and most importantly, when the chip is used for the very first time, always make sure that CH bit of register 0 is cleared, so that clock starts. After that always make sure whenever setting seconds, not to set this bit as logical 1 as this will halt the clock.

Project 1

Frequency Counter

Frequency is defined as number occurrences of an event in a specified time. Usually expressed as events per second. A frequency counter is a device that continuously monitors the occurrence of some event. This event can be an electrical signal, or a real world event, like number of cars passing through a gate. In any case the frequency counter counts the number of events in a given time scale and displays or records the data. Frequency counter consists of two parts, a digital part based upon microcontroller, which we will be working on in this project, and a suitable input part which captures the real world event, and converts it into appropriate microcontroller readable pulse. For example if you want to measure the radio-frequency you need a suitable adapter, that will capture the radio signal, and convert it into digital signal.

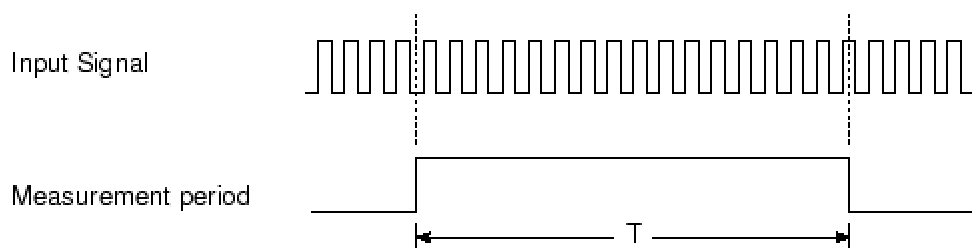
Since frequency of any event is usually variable, it is mandatory to sample the events frequently and update the display continuously. From this perspective there are two types of frequency counters, one which take the sample when required and display the result, they do not take another sample unless told to do so. The other type will continuously take samples of the input line and update the display. The time duration after which samples will be taken dictate the resolution of counter. Real time counters, do it almost continuously.

A simple frequency counter measures frequency by counting the number of edges of an input signal over a defined period of time (T).

A more complex method is reciprocal counting (we shall talk about it later).

Frequency is defined as (Number of events) / (time in seconds) and measured in Hz.

To make calculations trivial using a 1 second gate time (T) gives a direct reading of frequency from the edge counter.



Making a frequency counter for frequencies up to 65.535kHz is easy as the counters in a PIC chip can count up to 65535 without overflowing. Up to 65.535kHz all you do is wait for 1 second while the count accumulates, read the value and display it. It will be the frequency in Hertz. Above 65.536kHz you have to monitor the overflow value while at the same time making an accurate delay time (T).

Note: Using a 1 second measurement period results in the frequency counter count value being a direct measurement of frequency requiring no further processing. It also means that the measurement is resolved to 1Hz. (Increasing T to 10s resolves to 0.1Hz while using T=0.1s gives a resolution of 10Hz).

Crystal oscillator

For the following projects the crystal oscillator is used as the time-base. In these projects measurement of T (set at one second) is made by executing a delay that takes a set number of machine cycles.

Using a 20MHz oscillator gives a machine cycle of 5MHz (a period of 0.2us) which makes calculating and setting time delays fairly easy since most PIC instructions execute in one machine cycle. The accuracy of the frequency counter depends on the accuracy of the crystal driving the microcontroller. All crystals have some factor of error, which is expressed as PPM, parts per million. Thus a 4MHz crystal may actually be oscillating between 3.998 to 4.002 MHz, which can slightly change our time-base and therefore the counts. For our project we accept this small change extremely small and ignore it. However the frequency of oscillator can be changed by changing the capacitance, or by calibrating an internal correction variable to compensate for the change. Calibration is usually done by giving a signal of known frequency.

Timer 0 Algorithm

TMR0 (timer Zero) is ideal for frequency measurement (counting edges) as it can function as a 16 bit counter taking its input directly from a port pin (RA4,T0CKI). PIC-Lab-II has this pin separately taken out as T0CKI header, which also contains 5V power supply and ground for the probe, or for a test circuit. This the easiest way of measuring frequency using a PIC micro - you can use Timer 1 as well which can get its input from RC0.

By default TMR0 in 18F452 is an 8 bit counter, however it can be configured to work as 16 bit counter by setting T0CON bit 6 to 0. (see data sheet). In PIC 16F877 and like devices, it is 8 bit timer. So when designing your actual project either select TMR1, or appropriately adjust software if using TMR0. We will use TMR0 in 16bit mode in this project.

So if we begin with TMR0=0 every clock pulse on the appropriate pin will increase the count. This count cn reach a maximum value of 65535 (all 1s in 16 bit register). Now when the counter has reached its maximum, next pulse will reset it to 0, however it will generate an interrupt and set an overflow flag high. If we count the number of overflows, this gives us the number of 65536 cycles. Multiplying overflows with 65536 will give the number of pulses during that period of time, plus we have to add any more accumulated in TMR0 after last overflow. This will give us a total number of pulses arriving at the input pin in a given time period –T

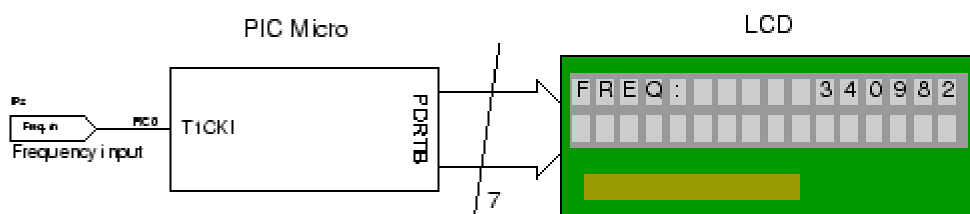
The trick to making the frequency counter algorithm work is that the overflow flag must be polled within the delay routine but it must be polled ensuring that the polling routine takes a constant time (So that the delay period can be calculated exactly).

For the frequency counter interrupts are not used at all in either measurement of the input signal edges or measurement of the time period T. This is because using an interrupt as part of the measurement process would interrupt the time measurement part of the code. The number of interrupts would be dependent on the input signal frequency and so the time measurement would be inaccurate. If the time period (T) measurement was made using a different method e.g. Using a 1s externally generated time period T then interrupts could safely be used.

Displaying Data

Data can be displayed in a number of ways. Obviously LCD is the simplest and easiest to implement, however not cost effective if making a commercial device. 7-segment displays are another method, however using them to display data requires little bit of more software coding.

For simplicity we shall use an LCD to display our data.



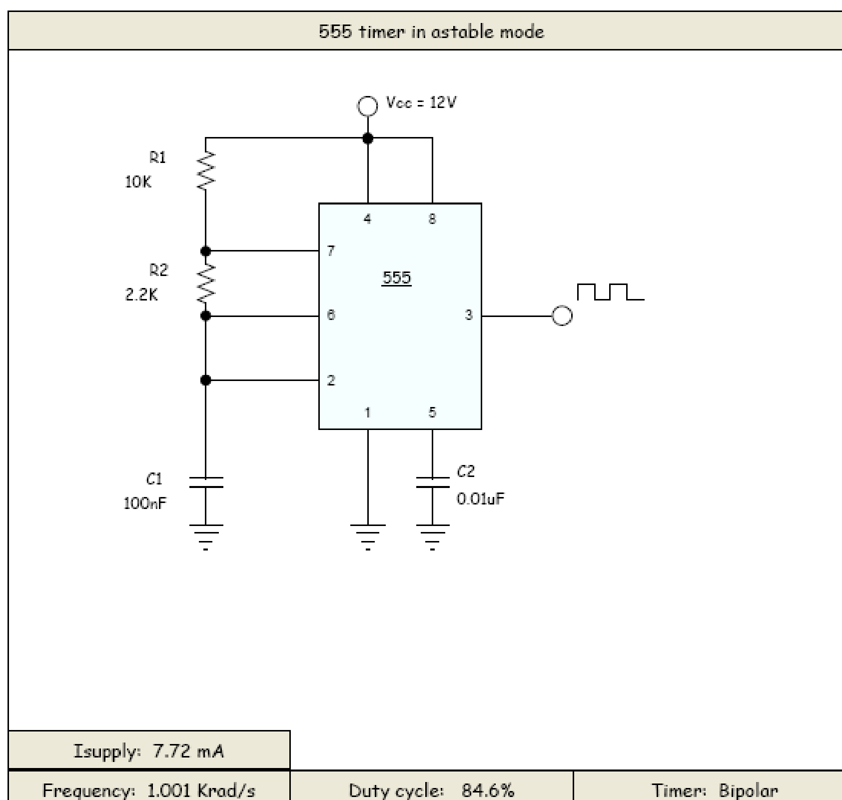
This project uses TMR0 of PIC18F452 in 16 bit mode to count the input edges appearing from an external circuit. The pulses are measured precisely for 1 second and then the number of overflows multiplied by 65536 plus the counts in TMR0 will give us the total number of pulses in 1 second. Since we are measuring it exactly for 1 second there is no need for further conversion, and the count is the frequency in Hz. However for display we can divide it by 1000 or 100000 to display as KHz or MHz, whichever is appropriate.

By keeping the time-base at 1 second there is another advantage. The minimum number of counts detected can be 1. So we can measure from as low as 1Hz to as high as 50MHz. (Upper limit is the limit of PIC input pin).

In order to learn the things we will begin with a simple procedure and then improve the project gradually.

Frequency Source:

In order to measure an input frequency you must have an external source of frequency. The external source must give its output as TTL level, like 0 and 5V but not more than that. If you want to measure analog frequency source then an additional circuitry to convert it into appropriate TTL level using gates and Schmitt triggers will be necessary. Foe demonstration purpose we will use 555 timer in astable mode to produce clock pulses. Its output will be given to the T0CKI pin of PIC.



The above figure shows how you can make a simple oscillator using 555 timer IC. You can replace R1 with a variable resistor to change the frequency. You can omit C2 if you want. So all you need is an IC, two resistors and a capacitor. In present configuration this circuit will generate almost 1000 cycles per second or 1KHz. Instead of using 12V as VCC, you can use 5V from your PIC-Lab-II. So T0CKI header can be directly connected to this board, powering it as well as measuring the frequency.

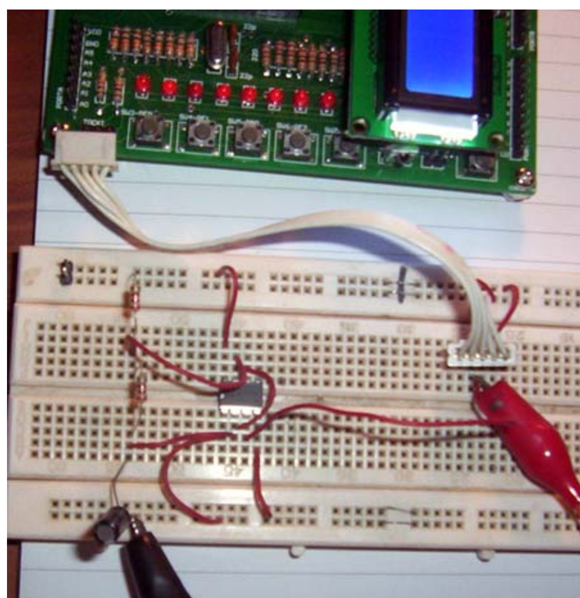


Figure on right shows the construction of 555 timer based oscillator on bread-board. The output of timer circuit is connected to PIC-Lab-II T0CKI header (RA4).

In our first and preliminary program, we have used R1,R2=2.2K and C1=1uf this is circuit should give a calculated frequency of 217 Hz, however when actually tested on an oscilloscope, because of small variation in resistors and capacitance, the measured frequency way 206.6 Hz .

Since the frequency is below 255, we can simply use the TMR0 in either 8 bit mode, or if used in 16 bits mode

measure the counts, accumulated in TMR0L register. In PIC18F452 TMR0 output is placed in TMR0H and TMR0L registers, which are two 8 bit registers. In our simplest frequency counter, we configure TMR0 as 16 bit timer, and do not use any pre-scalar, thus every pulse will increment the counter. Before measuring the sample we set TMR0 output registers to 0, and wait for 1 second, after 1 second we store the value of TMR0 registers into a 16 bit variable and display it. Since this variable can hold a value up to 65535, and the value corresponds exactly with the pulses, the count exactly gives us the frequency in Hz.

```
' frequency counter
Device = 18F452
XTAL=20
ALL_DIGITAL true
LCD_DTPIN PORTD.4
LCD_RSPIN PORTD.3
LCD_ENPIN PORTD.2
Symbol TMR0_ON T0CON.7       ' 1=Enable timer 0=disable
Symbol TMR0_8bit T0CON.6     ' 1= 8 Bit, 0=16 Bit
Symbol TMR0_CS T0CON.5       ' Clock Source 1=RA4 pin, 0=internal oscillator
Symbol TMR0_SE T0CON.4       ' Signal Edge, Rising or falling
Symbol TMR0_PSA  T0CON.3     ' Enable Prescalar, 1=OFF 0=ON
Symbol TMR0_PS2 T0CON.2      ' Prescalar settings if PSA enabled
Symbol TMR0_PS1 T0CON.1
Symbol TMR0_PS0 T0CON.0

TMR0_CS =1  'Count pulses on RA4
TMR0_8bit = 0 ' Use 16 bit counter
TMR0_PSA = 1 ' do not use prescaler count every pulse

Dim x As Word
Dim y As Word
Dim z As Word
TMR0_ON =1
loop:
x=0
y.LowByte=TMR0L
y.HighByte=TMR0H
DelayMS 1000
x.LowByte = TMR0L
x.HighByte=TMR0H
z=x-y
Print At 1,1, "Frequency:", At 2,1, DEC6 z, " Hz"
GoTo loop
```

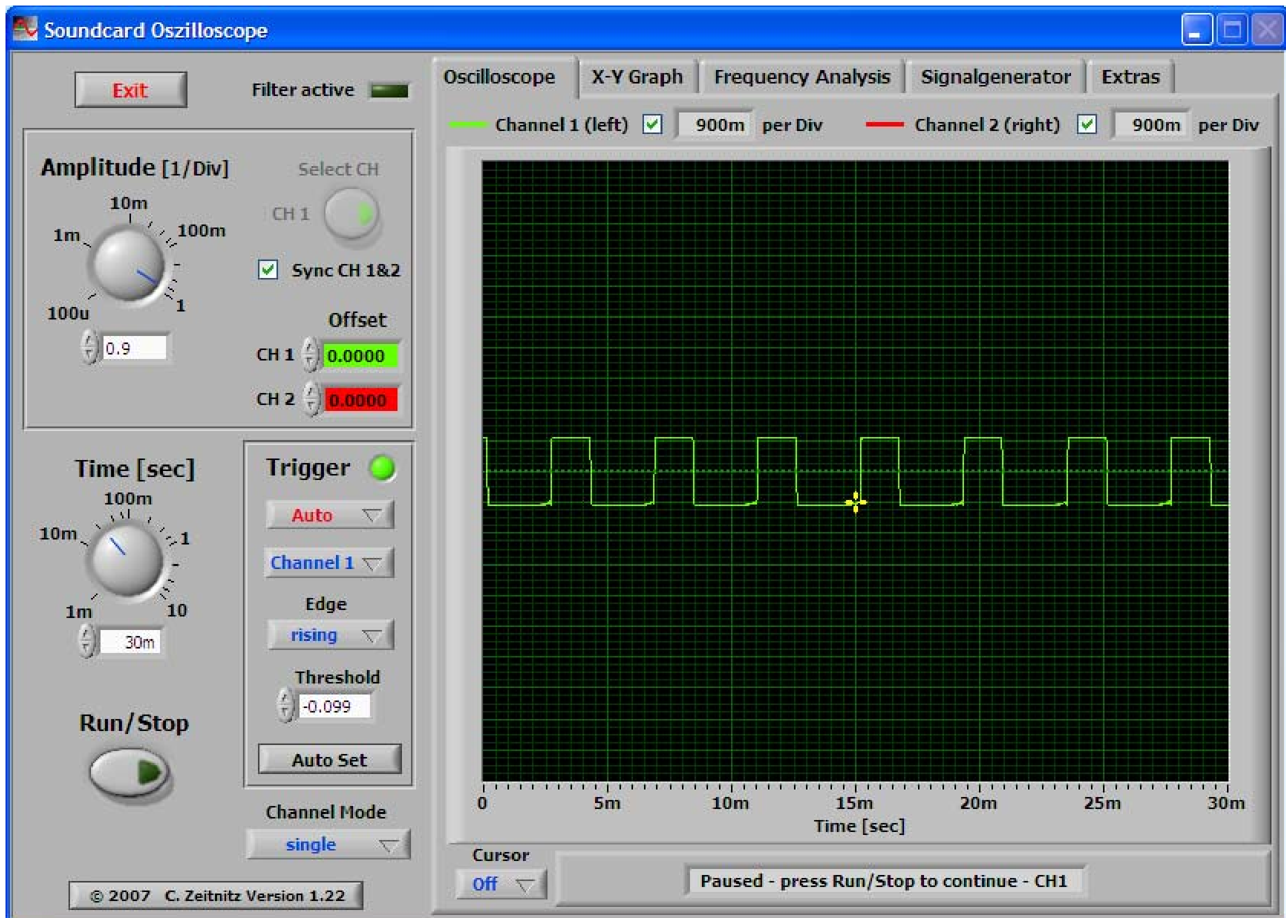
Since we are going to use TMR0 as our counter, it has an associated T0CON register which configures the properties of this timer. Instead of remembering its bits and their function, it is better to declare its as useful bit names and declare them as symbols in your program.

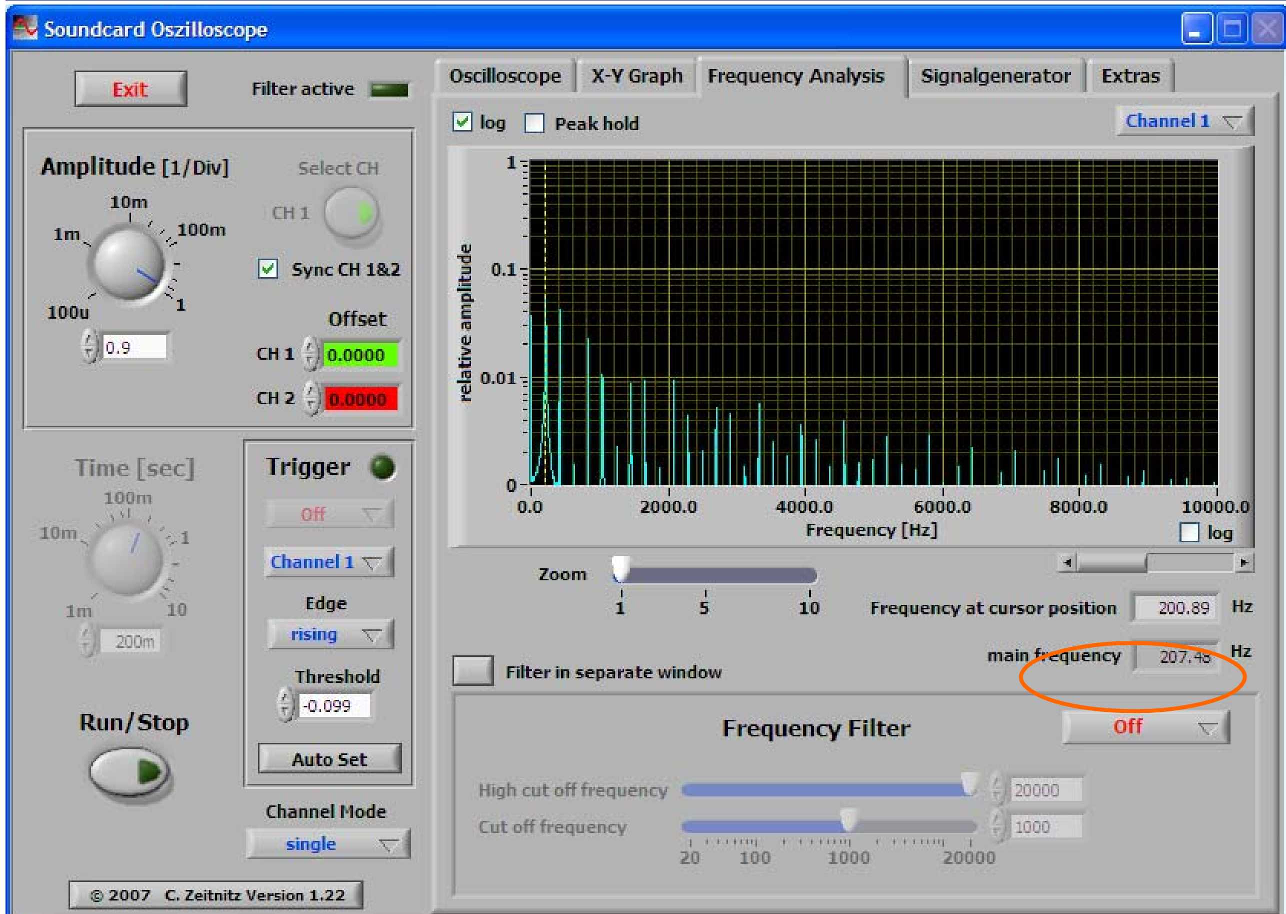
We have declared the entire T0CON register, however in this very program, we need only to manipulate few bits.

- Configure TMR0 to get clock pulses from RA4 pin
- Put TMR0 into 16 bits mode
- Disable Pre-Scalar
- Enable / Disable Timer when required

After appropriate configuration, the TMR0 output registers, TMR0H:TRM0L are initialized to 0. The Timer is then enabled the value of TMR0 registers is recorded in a variable and microcontroller put to wait for 1 second. After this the TMR0 value is again recorded in another variable. The difference in two variables is the accumulated value. We

want to put these two registers into 1 16bit variable. This is done by using **x.HighByte** and **x.LowByte**. The x now contains the 16 bit count of TMR0 register. The difference of x-y is the actual count and can be





displayed directly, as the accumulated number is the count of pulses in 1 second. The process is repeated again and again, to display real time frequency.

The oscilloscope shows the pulse train, at RA4 pin, showing output of 555 timer as TTL, pulses. The figure below shows frequency analysis, note the main frequency to be 207.48Hz, which is fairly close to the one measured by our frequency counter.

Since this frequency counter uses 16bit timer, it can measure a maximum of 65535 pulses, which will correspond to 65.535KHz. A frequency beyond that will reset the counters to 0, and there is no way to determine, if this was due to high frequency or it's the actual frequency.

There are two ways to counteract this problem, using the same technique.

First method is to reduce the time-base, so lets say we allow half a second to count the pulses, and then multiply the counted pulses by 2 to get the exact frequency. This will double the frequency range, however resolution will also be reduced, the minimum frequency measured will be 2Hz and its multiples. However the upper frequency will be 131070 Hz, or 131.07KHz. Further reducing the time-base by 1/4 seconds and multiplying the result with 4 gives a resolution of 4Hz to 262.140KHz.

The second method involves using pre-scalar. The prescalar will divide the count by 2 to 256 depending upon the settings in PSA bits. If we use a prescaler of 1:2 and a time-base of 1 second, this will be effectively same as 1/2 second time-base. We will have to multiply the count by 2. if we use the prescaler to 1:256, the minimum frequency will be 256 Hz and highest frequency will be, 65535 * 256=16776960 Hz or 16776.960 KHz or 16.776MHz.

bit 2-0 **T0PS2:T0PS0**: Timer0 Prescaler Select bits

111 = 1:256 prescale value
110 = 1:128 prescale value
101 = 1:64 prescale value
100 = 1:32 prescale value
011 = 1:16 prescale value
010 = 1:8 prescale value
001 = 1:4 prescale value
000 = 1:2 prescale value

Thus using this simple technique, which does not involve any interrupts, we can measure up to 16MHz , however the higher the range, the resolution also drops. So at this frequency range, we can measure minimum of 256 Hz, the next frequency would be 512 Hz. Frequencies in between can not be measured. Even at higher ends, the frequencies will be measured in multiples of 256. this is ok for a general purpose crude system, but certainly not acceptable for a professional system.

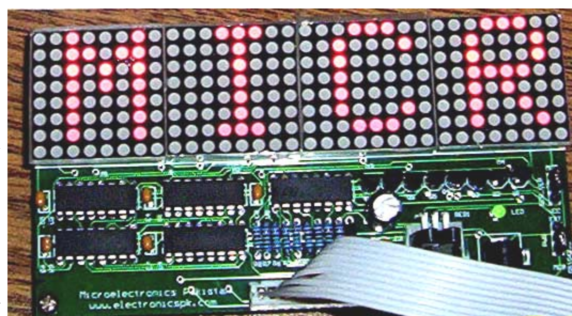
You can think of advanced techniques, to make a more versatile professional type frequency counter, I have seen PIC based projects that can count from 1 MHz to 50MHz.

Project 2

LED Matrix

LEDs are great small devices that emit light, yet do not consume much energy and do not emit heat. They can be arranged in many fashions, to produce visual effects, one of the most common arrangement is to put them in the form of a matrix, just like key pad. So if we have a matrix of 5 columns and 7 rows we have 35 LEDs. However when put in this format they do not have 35 lines to control them, instead they are controlled by 12 lines, 7 for individual row and 5 for individual columns. The LED to be lightened is controlled by selecting a row and column, the led at its intersection will light up. Thus by selecting the rows and columns, very quickly and using persistence of vision, a number of patterns and animations can be made.

In this section we will discuss briefly Microtronics 8x32 matrix LED device.



This device contains 4, 8x8 matrix LED modules, connected through shift registers. The entire module therefore has 8 rows starting from top and 32 columns starting from left.

The data is shifted one bit at a time, using Serial Parallel Interface, which is timed by clock signals.

The columns are negative and rows are positive. Thus a logical 0 on row will lighten up the corresponding LED.

This project is an excellent guide to understanding shift registers as well.

In order to send 4 bytes, they are sent serially one bit at a time, along with clock signals, when all the data has been transferred, it is still in shift registers, to show data on pins, and therefore displays, the shift registers are sent a latch impulse. When one row is displayed, the data is again sent and before latching, the row counter is given a pulse, so that next row is selected. The entire process is repeated 8 times till all rows are displayed, remember when a new row is selected the previous row is deselected. Thus you can display one row at a time. After all eight rows have been scanned, the row counter is sent a reset pulse, so that row 1 is selected again. This process is repeated again and again, and at very rapid speed, so that it looks that all rows are ON at the same time.

The connector on this board is a 10 pin connector, which is compatible with PIC-Lab-II connectors. The various pins in this connector are arranged as:

1	2	3	4	5	6	7	8	9	10
SER	CLK	CLR	LAT	RCLK	RST			GND	VCC

The pin numbers start from left. The function of these pins is described below:

SER is serial data in pin, which receives one bit at a time.

CLK is the clock pin, which gets impulses to accept data on SER.

CLR is for clearing the shift registers.

LAT is to latch the shift register data to output lines.

RCLK is to clock the row selection

RST is to reset the row counter.

GND and VCC are 5V power supply from motherboard.

The board can have its own power supply, in that case a jumper on board has to be selected to select the source either Mother board or external.

Programming The Display

Programming the display board is simple. However it can get quite complex, when you want to display animations and special effects. The display itself does not stop you from any innovation. It simply requires that one row of data , which is 32 bits or 8 bytes be clocked into the shift register, and the appropriate row selected using row counter.

The following prototype examples are only basic guidelines on using this display. We will be using BASIC as programming language, and PIC microcontrollers as controlling device. You may adapt these guidelines to your particular scenario.

We assume following connections from microcontroller to display board and define them in our program as constants.

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTB
Symbol SER = PORTB.0     ' Serial data Pin
Symbol SRCLK = PORTB.1   ' Serial data Clock Pin
Symbol SRClr = PORTB.2   ' Serial data Clear
Symbol Latch = PORTB.3   ' Columns, Latch
Symbol RowClk = PORTB.4 ' Row clock, to select new row
Symbol Rowrst = PORTB.5 ' Row reset, selects row 0
High SRClr               ' Turn off the serial register clear
```

Remember, if no data is clocked to shift registers, they will be in state of 0 on their outputs, and since 0 on a column selects it, so all columns will be selected. A logical 1 on a column will turn the column off, and a 0 will turn the column ON.

```
Device=18f452
XTAL=20
ALL_DIGITAL=true
Output PORTB
Input PORTA
Symbol SER = PORTB.0     ' Serial data Pin
Symbol SRCLK = PORTB.1   ' Serial data Clock Pin
Symbol SRClr = PORTB.2   ' Serial data Clear
Symbol Latch = PORTB.3   ' Columns, Latch
Symbol RowClk = PORTB.4 ' Row clock, to select new row
Symbol Rowrst = PORTB.5 ' Row reset, selects row 0
High SRClr               ' Turn off the serial register clear
PulsOut Rowrst,2     ' give a pulse on row reset pin, to select row 0
SHOut SER,SRCLK,lsbfirst,[%11111110,%11111111,%11111111,%11111111] ' send data
on serial pin
PulsOut Latch,4
End
```

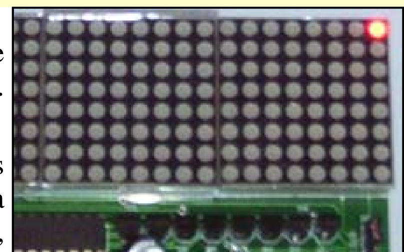


Fig. 6 Single LED is ON

The **PulsOut** command gives a small high pulse on the specified pin. The number indicates duration of pulse. 2 in this case will give a 4us pulse. However thgis timing is not crucial for the function of display.

SHOut is the command that transmits the contents of a byte, or word, as a stream of bits on a single pin. The arguments are the pin, on which data is to be transmitted, the Clock pin to be used to clock the shift registers, **lsbfirst** is a reserved word, indicating least significant bit first. So it will transmit, the bit 0 of data first and bit 7 last. The numbers in square brackets are the data to be sent. These are 4 bytes of data the first byte is sent first, from left side, the next three bytes push the first sent byte successively to right, so at the end of all 4 bytes the first Byte has been sent to the right most 8 columns. You can play with these four bytes to make various leds, ON. You must have noticed that the column corresponding to 0 is turned on. So to turn the entire row ON, just send this data:

```
SHOut SER,SRCLK,lsbfirst,[%00000000,%00000000,%00000000,%00000000] ' send data
```

```
on serial pin
```

Now lets send some pattern, to show this effect.

```
SHOut SER,SRCLK,lsbfirst,[%
11111110,%10101010,%11001100,%
00011100]
```

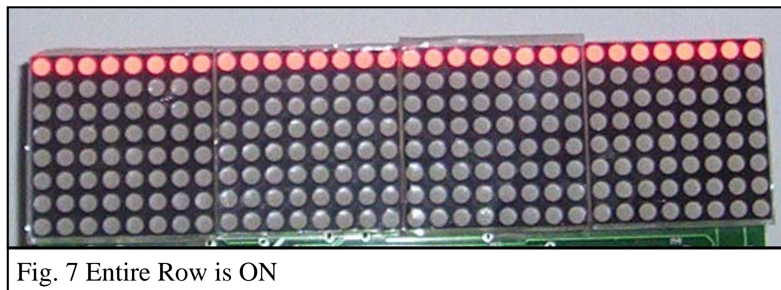


Fig. 7 Entire Row is ON

Notice the data, being sent, and the appearance of ON LEDs in Fig. 8. this clearly shows that the first byte sent, goes to extreme right, and last byte sent goes to extreme left.

If we want to reverse the pattern , we can send Most significant bit first.

Blinking an LED

In order to make an LED Blink, you have to turn the corresponding column On, and OFF repeatedly after a set interval.

```
PulsOut Rowrst,2    ' give a pulse on row reset pin, to select row 0
Loop:
SHOut SER,SRCLK,lsbfirst,[%11111110,%11111111,%11111111,%11111111] ' send data
on serial pin ON
PulsOut Latch,4
DelayMS 500
SHOut SER,SRCLK,lsbfirst,[%11111111,%11111111,%11111111,%11111111] ' send data
on serial pin OFF
PulsOut Latch,4
DelayMS 500
GoTo Loop
End
```

Now lets make the entire row turn ON, and then select the next row, till all 8 rows are show, one after the other.

```
Dim i As Byte
Loop:
PulsOut Rowrst,2    ' give a pulse on row reset pin, to select row 0
For i=0 To 7
SHOut SER,SRCLK,lsbfirst,[%00000000,%00000000,%00000000,%00000000]
PulsOut Latch,4
DelayMS 500
PulsOut RowClk,2
Next i
GoTo Loop
```

This code sends all columns on data, then waits for 500ms, and then gives a short pulse on RowClk pin. This pulse will advance the row selection to next row, and the same data is sent again. The whole process is repeated 8 times, till the last row, number 7 is displayed. After this the row counter is reset to select row 0 and the entire process is repeated.

So far so good. Now begins the real fun. Notice that we have made a delay of 500ms between rows. If we reduce this time, rows will be displayed quicker,. Fine, if we go on reducing the time, a state will reach when our eyes will not be able to perceive the individually on rows, rather they will see all rows ON! When in fact, still, one row is being turned on, at a time. Try following timings, in place of 500. begin with 500 as in above program and then try, 300, 200, 100, 50, 20, 10, 5, 2 . A delay of 2ms will produce the best result, you will see an absolutely flicker free display.

As shown in figure 9. All 256 LEDs appear to be ON.

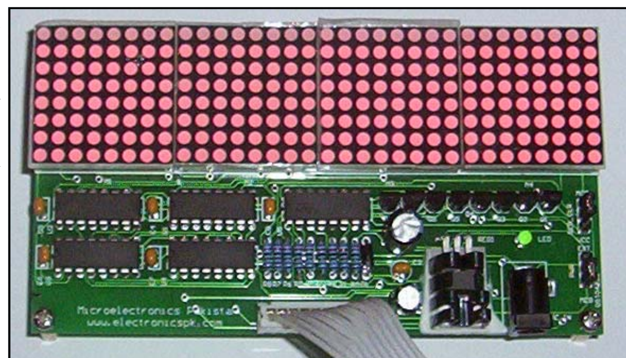


Fig. 9 All lights appearing to be ON

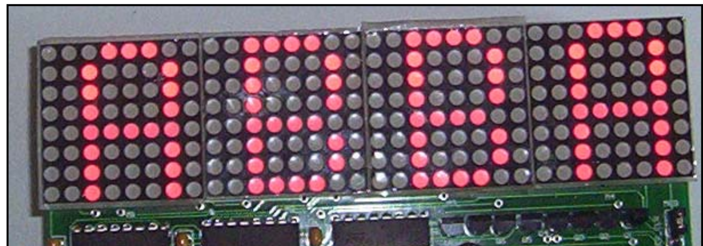
When in fact one row, of 32 are on at a time. This is due to persistence of vision.

Displaying Characters

As you have seen that, you can play with these bits, to show anything you want. Text characters are similarly not a big problem. However you will need to make a table or list of character maps, that you will send to the display. Most characters can be easily mapped in 5 x 7 array. However for simplicity of our work, we will design them as an 8 x 8 matrix. Selecting this matrix, will also be helpful in making some other more feature rich graphics. Lets see how to work Up. We are going to design letter 'A'.

								E3
								DD
								DD
								DD
								C1
								DD
								DD
								DD

The adjacent fig shows the character map. The numbers in gray column, are in hexadecimal, equivalent. 1 for Off and 0 for On. Now we can store this character in bytes of memory. EEPROM on microcontroller is a good place to keep such maps, which once defined, have to be read in by the program. We have used **EData** command to store the codes for each row byte for the letter in EEPROM. The size of table which can be stored will depend upon the EEPROM of your microcontroller. If a larger table is required you can use an external EEPROM. The loop fills in the appropriate bytes of display memory by reading the associated bitmap image from EEPROM.



The index of letters to be displayed are stored in an array z[4]. Number 0 is for letter 'A' as it is defined in position 0 in EEPROM, and number 1 is for 'B' as it is in position 1 in EEPROM. Notice how the position of first byte is calculated.

```
Device=18F452
XTAL=20
ALL_DIGITAL=true
Output PORTB
Input PORTE
Symbol SER = PORTB.0     ' Serial data Pin
Symbol SRCLK = PORTB.1   ' Serial data Clock Pin
Symbol SRClr = PORTB.2   ' Serial data Clear
Symbol Latch = PORTB.3   ' Columns, Latch
Symbol RowClk = PORTB.4  ' Row clock, to select new row
Symbol Rowrst = PORTB.5  ' Row reset, selects row 0
Symbol SW1=PORTE.0
Symbol SW2=PORTE.1
High SRClr               ' Turn off the serial register clear

Dim s[32] As Byte
Dim i As Byte
Dim n As Byte
Dim b As Byte
Dim c As Byte
Dim z[4] As Byte
z[0]=0
z[1]=1
z[2]=1
z[3]=0
' initialize the array by writing 0 to all bits
For i=0 To 31
    s[i]=$FF
Next i

For c = 0 To 3
For i= 0 To 8
    b=(i*4) + c
```

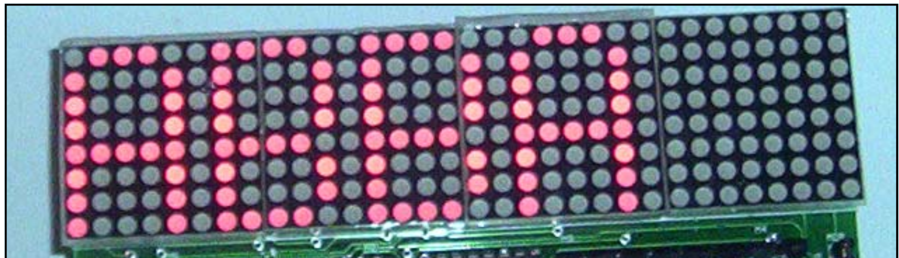
```
    s[b]=ERead (z[c]*8)+i
    Next i
Next c



Loop:
PulsOut Rowrst,2    ' give a pulse on row reset pin, to select row 0
For i=0 To 7
n=i*4
SHOut SER,SRCLK,lsbfirst,[s[n], s[n +1], s[n+2], s[n+3]]
PulsOut Latch,4
DelayMS 1
SHOut SER,SRCLK,lsbfirst,[$FF,$FF,$FF,$FF]
PulsOut Latch,4
PulsOut RowClk,2
Next i
GoTo Loop
End
EData $E3,$DD,$DD,$DD,$c1,$DD,$DD,$DD   ' A
EData $C3,$DD,$DD,$DD,$C3,$DD,$DD,$C3   ' B
```

As we have defined the character map as 8 bits wide, whereas our actual character is using only 5 bits. There are 3 bits empty, 1 on right and 2 on left side of each character. If we want to ignore the highest 2 bits, and we just want to send the lowest 6 bits to the display, for each character, we can do that by just mentioning the number of bits in Shout command. Change the Shout command like this:

```
SHOut SER,SRCLK,lsbfirst,[s[n]\6, s[n +1]\6, s[n+2]\6, s[n+3]\6]
```

Notice the \6 with every byte sent. If this is not mentioned then default, 8 bits is assumed. You can send any number of bits you want. The result of this would be:

You can make these letters scroll, make special effects, and animations, however we leave this to you, so that you learn while explore.

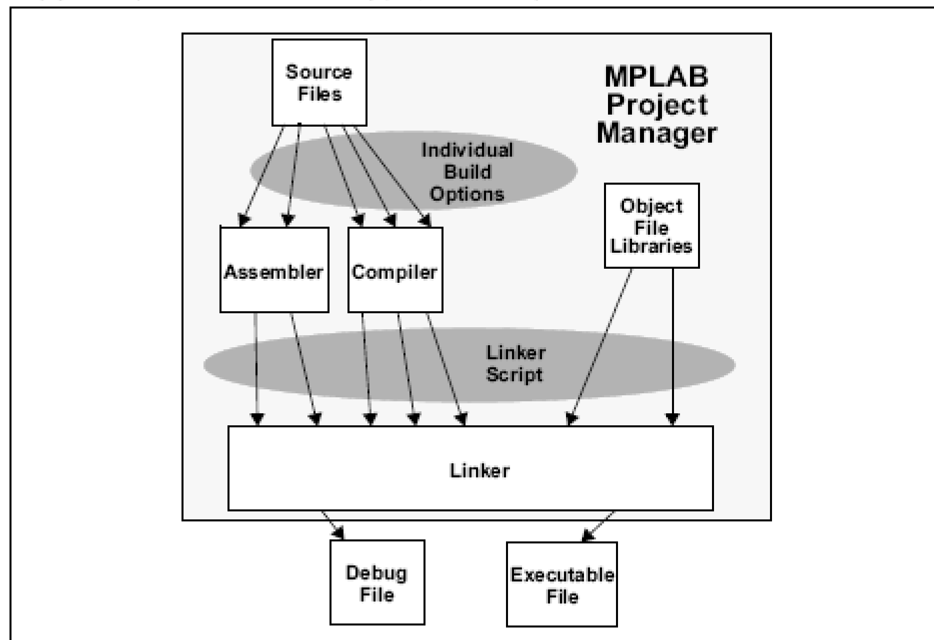


Working with MPLAB®

Microchip® MPLAB is a software, that can be downloaded free from Microchip site. As of this writing version 8.0 is available. MPLAB is an integrated development environment for PIC microcontrollers from Microchip. The platform supports native assembly language, and programs written in assembly can be directly compiled on this platform. It also supports many other supporting tools, like C17, C18, C24 etc. these are C language compilers available from microchip site. These compilers are integrated with MPLAB, and you can write software, for all supported devices, debug them, within MPLAB, see the status of various registers and then burn the hex file into the PIC using MPLAB supported programmers, or your own, while generating hex file from MPLAB.

The MPLAB organizes your entire development as a project, which may contain various source files, linker libraries and so on.

FIGURE 1-6: MPLAB PROJECT MANAGER



One of the beautiful aspects of MPLAB is integration with microchip ICD-2. This device is both a programmer as well as in circuit debugger. Your program can be run and tested right in the target board, as well as stopped and you can examine its registers.

Details of this software and Microchip ICD-2 can be found at microchip site.

Microchip® Self-Programming Boot-Loader

Programming a microcontroller needs a hardware device, called programmer. All programmers have the same basic functionality, that they accept a program (.hex) file from your PC and transfers it to the program memory of microcontroller. The number of commercial designs vary in speed, availability of serial port, parallel port or USB and the supported devices. You have been using our simple, yet fully functional programmer PIC-PG-II. We also introduced you another useful device, ICD-2 which operates under Microchip MPLAB software and not only program the MPLAB supported devices but also helpful in source level debugging of the project, right in circuit.

Microchip has introduced another technology, which has simplified the task of programming, as well as upgrading the firmware within your projects. This is called Self programming. Newer PIC microcontrollers, like 16F877 and all 18Fxxx series have this capability. If you use this technique, you do not need an external programmer at-all. The programming is done through standard USART serial interface, which almost every project has.

PIC Lab-II is equipped with this software, so that you can use direct programming without the need of intervening programmer. This does not mean you should not have the programmer at hand! It still has useful functions.

Boot Loader

Boot loader is a piece of software required to use this technology. It has two components, a firmware that resides in your microcontroller and a client program that is installed on your PC. The firmware has to be compiled for your particular microcontroller, and the board clock speed. This is a small program, that first needs to be uploaded into your microcontroller using a standard programmer. After it is loaded, you do not need programmer, unless accidently the firmware in PIC is deleted.

The other part of boot-loader is installed on your PC, and it accepts the .hex file to be uploaded, which is the software the microcontroller is supposed to execute. Your board must be connected to the serial port of your computer, and on PIC Lab-II LED Dip Switch SW1 should be off as LEDs interfere with serial communication.

Now when you press the reset button on PIC Lab-II, or when the power is turned on, the control is first transferred to boot loader software in microcontroller, this software, which is loaded in the high memory of program area, monitors the serial port if the PC is sending a new software or not. If there is nothing new, the boot loader hands over control to the already existing software in microcontroller, which starts functioning whatever it is supposed to do. However if a new program is coming the existing program, (leaving boot loader) is erased and new program is written into program memory. After that control is transferred to new program. This process does not require 12V on MCLR.

Where to get Boot Loader?

A large number of companies including microchip is offering the boot loader program, however we have found a free to download and very versatile software, called Tiny Boot Loader. This software is included on the accompanying CD. The software will consist of pre-compiled hex files for PIC Lab-II board, named as PIC18F452_20.hex this is for 18F452 microcontroller, running at 20MHz. The software also contains source files, which can be modified for your particular microcontroller if required.

To load this file into your microcontroller, attach your programmer to the motherboard, and run ICPROG locate the relevant hex file in boot loader folder, and transfer it into your microcontroller. That is all that is required. Now disconnect the programmer, and connect the serial cable to your computer and PIC Lab-II. You will notice an executable application in boot loader folder, TinyBldwin.exe just double click it. Select the speed as 115200 (maximum speed) and select your Com Port. Browse the .hex file, you want to run, like blink.hex its name and path will appear in the drop-down list box. Unlike ICPROG program code etc will not be displayed. Now make sure your Serial port is available, LEDs on PIC Lab-II are disabled. Power the PIC Lab-II on, and press the reset button, immediately press ‘W’ on your PC to write the program. The

new software will be transferred into your microcontroller. And start running. If you want to enable LEDs now you can do so. Now if you want to update the software, like you have made a few changes, in source file and compiled to get new hex file, if file name has not been changed, just open the TinyBldWin.exe, press reset on PIC Lab-II and immediately (within 10 sec) press ‘W’ on PC, the new program will be up-dated.

Thank you for being with us, thus was just the beginning, of a voyage to deep sea. I hope my this humble attempt to help you getting started would be worth while.

Dr. Amer Iqbal
206 Sikandar Block
Allama Iqbal Town
Lahore
ameriqbalqureshi@yahoo.com

www.electronicspk.com