

PIC microcontrollers, *for beginners too*

on-line, **FREE!**

author: Nebojsa Matic

PIC microcontrollers : low-cost computers-in-a-chip; they allow electronics designers and hobbyists add intelligence and functions that mimic big computers for almost any electronic product or project. The purpose of this book is not to make a microcontroller expert out of you, but to make you equal to those who had someone to go to for their answers. Book contains many practical examples, complete assembler instruction set, appendix on MPLAB program package and more...



In this book you can find:

Practical connection samples for

Relays, Optocouplers, LCD's, Keys, Digits, A to D Converters, Serial communication etc.

Introduction to microcontrollers

Learn what they are, how they work, and how they can be helpful in your work.

Assembler language programming

How to write your first program, use of macros, addressing modes...

Instruction Set

Description, sample and purpose for using each instruction...

MPLAB program package

How to install it, how to start the first program, following the program step by step in the simulator...



[E-mail a friend](#)
[about this item](#)

Contents:

CHAPTER I

INTRODUCTION TO MICROCONTROLLERS

[Introduction](#)
[History](#)
[Microcontrollers versus microprocessors](#)

[1.1 Memory unit](#)
[1.2 Central processing unit](#)
[1.3 Buses](#)
[1.4 Input-output unit](#)
[1.5 Serial communication](#)
[1.6 Timer unit](#)
[1.7 Watchdog](#)
[1.8 Analog to digital converter](#)
[1.9 Program](#)

CHAPTER II

MICROCONTROLLER PIC16F84

[Introduction](#)
[CISC, RISC](#)
[Applications](#)
[Clock/instruction cycle](#)
[Pipelining](#)
[Pin description](#)

[2.1 Clock generator - oscillator](#)
[2.2 Reset](#)
[2.3 Central processing unit](#)
[2.4 Ports](#)
[2.5 Memory organization](#)
[2.6 Interrupts](#)
[2.7 Free timer TMR0](#)
[2.8 EEPROM Data memory](#)

CHAPTER III

ASSEMBLY LANGUAGE PROGRAMMING

[Introduction](#)
[3.1 Representing numbers in assembler](#)
[3.2 Assembly language elements](#)
[3.3 Writing a sample program](#)
[3.4 Control directives](#)
[3.5 Files created as a result of program translation](#)

CHAPTER IV

MPLAB

[Introduction](#)
[4.1 Installing the MPLAB program package](#)
[4.2 Welcome to MPLAB](#)
[4.3 Designing a project](#)
[4.4 Creating a new Assembler file](#)
[4.5 Writing a program](#)
[4.6 Toolbar icons](#)
[4.7 MPSIM simulator](#)

CHAPTER V

MACROS AND SUBPROGRAMS

[Introduction](#)
[5.1 Macros](#)
[5.2 Subprograms](#)
[5.3 Macros used in the examples](#)

CHAPTER VI

EXAMPLES FOR SUBSYSTEMS WITHIN MICROCONTROLLER

[Introduction](#)
[6.1 Writing to and reading from EEPROM](#)
[6.2 Processing interrupt caused by changes on pins RB4-RB7](#)
[6.3 Processing interrupt caused by change on pin RB0](#)
[6.4 Processing interrupt caused by overflow on timer TMR0](#)
[6.5 Processing interrupt caused by overflow on TMR0 connected to external input \(TOCKI\)](#)

CHAPTER VII

EXAMPLES

[Introduction](#)
[7.1 The microcontroller power supply](#)
[7.2 LED diodes](#)
[7.3 Push buttons](#)
[7.4 Optocouplers](#)
[7.4.1 Optocoupler on input line](#)
[7.4.2 Optocoupler on output line](#)
[7.5 Relay](#)
[7.6 Generating sound](#)
[7.7 Shift registers](#)
[7.7.1 Input shift register](#)
[7.7.2 Output shift register](#)
[7.8 7-segment display \(multiplexing\)](#)
[7.9 LCD display](#)
[7.10 Software SCI communication](#)

CHAPTER 1

Introduction to Microcontrollers

[Introduction](#)

[History](#)

[Microcontrollers versus microprocessors](#)

[1.1 Memory unit](#)

[1.2 Central processing unit](#)

[1.3 Buses](#)

[1.4 Input-output unit](#)

[1.5 Serial communication](#)

[1.6 Timer unit](#)

[1.7 Watchdog](#)

[1.8 Analog to digital converter](#)

[1.9 Program](#)

Introduction

Circumstances that we find ourselves in today in the field of microcontrollers had their beginnings in the development of technology of integrated circuits. This development has made it possible to store hundreds of thousands of transistors into one chip. That was a prerequisite for production of microprocessors , and the first computers were made by adding external peripherals such as memory, input-output lines, timers and other. Further increasing of the volume of the package resulted in creation of integrated circuits. These integrated circuits contained both processor and peripherals. That is how the first chip containing a microcomputer , or what would later be known as a microcontroller came about.

History

It was year 1969, and a team of Japanese engineers from the BUSICOM company arrived to United States with a request that a few integrated circuits for calculators be made using their projects. The proposition was set to INTEL, and Marcian Hoff was responsible for the project. Since he was the one who has had experience in working with a computer (PC) PDP8, it occured to him to suggest a fundamentally different solution instead of the suggested construction. This solution presumed that the function of the integrated circuit is determined by a program stored in it. That meant that configuration would be more simple, but that it would require far more memory than the project that was proposed by Japanese engineers would require. After a while, though Japanese engineers tried finding an easier solution, Marcian's idea won, and the first microprocessor was born. In transforming an idea into a

ready made product , Frederico Faggin was a major help to INTEL. He transferred to INTEL, and in only 9 months had succeeded in making a product from its first conception. INTEL obtained the rights to sell this integral block in 1971. First, they bought the license from the BUSICOM company who had no idea what treasure they had. During that year, there appeared on the market a microprocessor called 4004. That was the first 4-bit microprocessor with the speed of 6 000 operations per second. Not long after that, American company CTC requested from INTEL and Texas Instruments to make an 8-bit microprocessor for use in terminals. Even though CTC gave up this idea in the end, Intel and Texas Instruments kept working on the microprocessor and in April of 1972, first 8-bit microprocessor appeared on the market under a name 8008. It was able to address 16Kb of memory, and it had 45 instructions and the speed of 300 000 operations per second. That microprocessor was the predecessor of all today's microprocessors. Intel kept their developments up in April of 1974, and they put on the market the 8-bit processor under a name 8080 which was able to address 64Kb of memory, and which had 75 instructions, and the price began at \$360.

In another American company Motorola, they realized quickly what was happening, so they put out on the market an 8-bit microprocessor 6800. Chief constructor was Chuck Peddle, and along with the processor itself, Motorola was the first company to make other peripherals such as 6820 and 6850. At that time many companies recognized greater importance of microprocessors and began their own developments. Chuck Peddle leaved Motorola to join MOS Technology and kept working intensively on developing microprocessors.

At the WESCON exhibit in United States in 1975, a critical event took place in the history of microprocessors. The MOS Technology announced it was marketing microprocessors 6501 and 6502 at \$25 each, which buyers could purchase immediately. This was so sensational that many thought it was some kind of a scam, considering that competitors were selling 8080 and 6800 at \$179 each. As an answer to its competitor, both Intel and Motorola lowered their prices on the first day of the exhibit down to \$69.95 per microprocessor. Motorola quickly brought suit against MOS Technology and Chuck Peddle for copying the protected 6800. MOS Technology stopped making 6501, but kept producing 6502. The 6502 was a 8-bit microprocessor with 56 instructions and a capability of directly addressing 64Kb of memory. Due to low cost , 6502 becomes very popular, so it was installed into computers such as: KIM-1, Apple I, Apple II, Atari, Comodore, Acorn, Oric, Galeb, Orazo, Ultra, and many others. Soon appeared several makers of 6502 (Rockwell, Sznertek, GTE, NCR, Ricoh, and Comodore takes over MOS Technology) which was at the time of its prosperity sold at a rate of 15 million processors a year!

Others were not giving up though. Frederico Faggin leaves Intel, and starts his own Zilog Inc.

In 1976 Zilog announced the Z80. During the making of this microprocessor, Faggin made a pivotal decision. Knowing that a great deal of programs have been already developed for 8080, Faggin realized that many would stay faithful to that microprocessor because of great expenditure which redoing of all of the programs would result in. Thus he decided that a new processor had to be compatible with 8080, or that it had to be capable of performing all of the programs which had already been written for 8080. Beside these characteristics, many new ones have been added, so that Z80 was a very powerful microprocessor in its time. It was able to address directly 64 Kb of memory, it had 176 instructions, a large number of registers, a built in option for refreshing the dynamic RAM memory, single-supply,

greater speed of work etc. Z80 was a great success and everybody converted from 8080 to Z80. It could be said that Z80 was without a doubt commercially most successful 8-bit microprocessor of that time. Besides Zilog, other new manufacturers like Mostek, NEC, SHARP, and SGS also appeared. Z80 was the heart of many computers like Spectrum, Partner, TRS703, Z-3 .

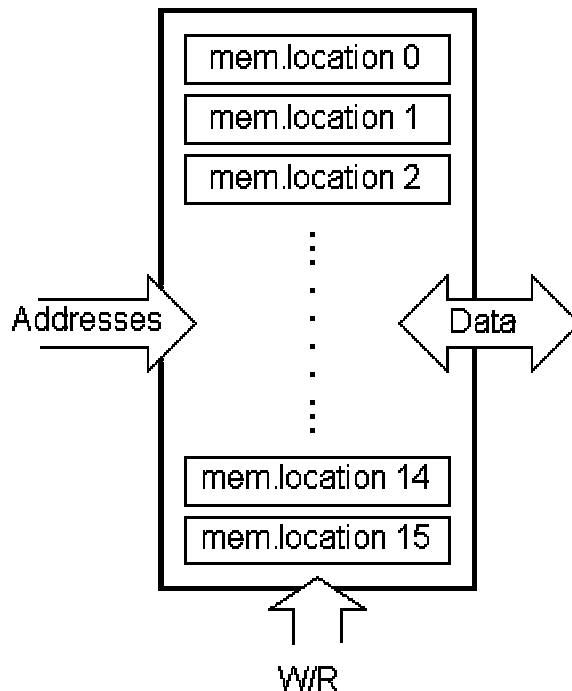
In 1976, Intel came up with an improved version of 8-bit microprocessor named 8085. However, Z80 was so much better that Intel soon lost the battle. Although a few more processors appeared on the market (6809, 2650, SC/MP etc.), everything was actually already decided. There weren't any more great improvements to make manufacturers convert to something new, so 6502 and Z80 along with 6800 remained as main representatives of the 8-bit microprocessors of that time.

Microcontrollers versus Microprocessors

Microcontroller differs from a microprocessor in many ways. First and the most important is its functionality. In order for a microprocessor to be used, other components such as memory, or components for receiving and sending data must be added to it. In short that means that microprocessor is the very heart of the computer. On the other hand, microcontroller is designed to be all of that in one. No other external components are needed for its application because all necessary peripherals are already built into it. Thus, we save the time and space needed to construct devices.

1.1 Memory unit

Memory is part of the microcontroller whose function is to store data. The easiest way to explain it is to describe it as one big closet with lots of drawers. If we suppose that we marked the drawers in such a way that they can not be confused, any of their contents will then be easily accessible. It is enough to know the designation of the drawer and so its contents will be known to us for sure.

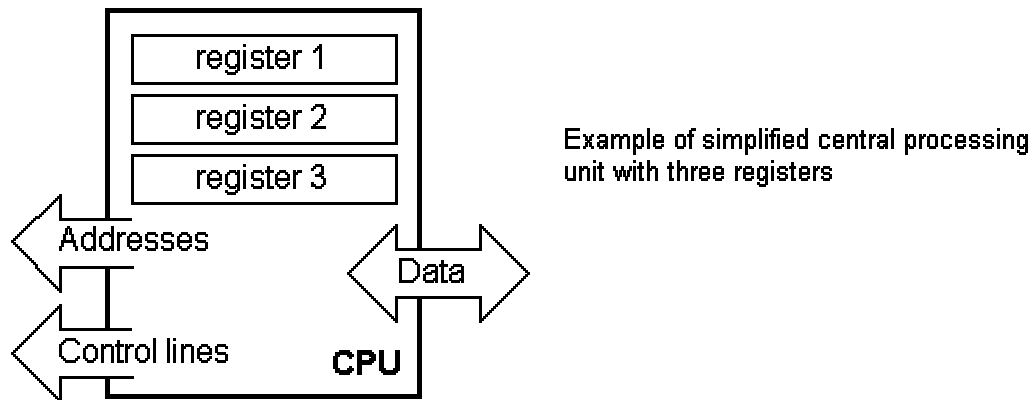


Example of simplified model of a memory unit. For a specific input we get a corresponding output. Line R/W determines wheather we are reading from or writing to memory

Memory components are exactly like that. For a certain input we get the contents of a certain addressed memory location and that's all. Two new concepts are brought to us: addressing and memory location. Memory consists of all memory locations, and addressing is nothing but selecting one of them. This means that we need to select the desired memory location on one hand, and on the other hand we need to wait for the contents of that location. Beside reading from a memory location, memory must also provide for writing onto it. This is done by supplying an additional line called control line. We will designate this line as R/W (read/write). Control line is used in the following way: if $r/w=1$, reading is done, and if opposite is true then writing is done on the memory location. Memory is the first element, and we need a few operation of our microcontroller .

1.2 Central Processing Unit

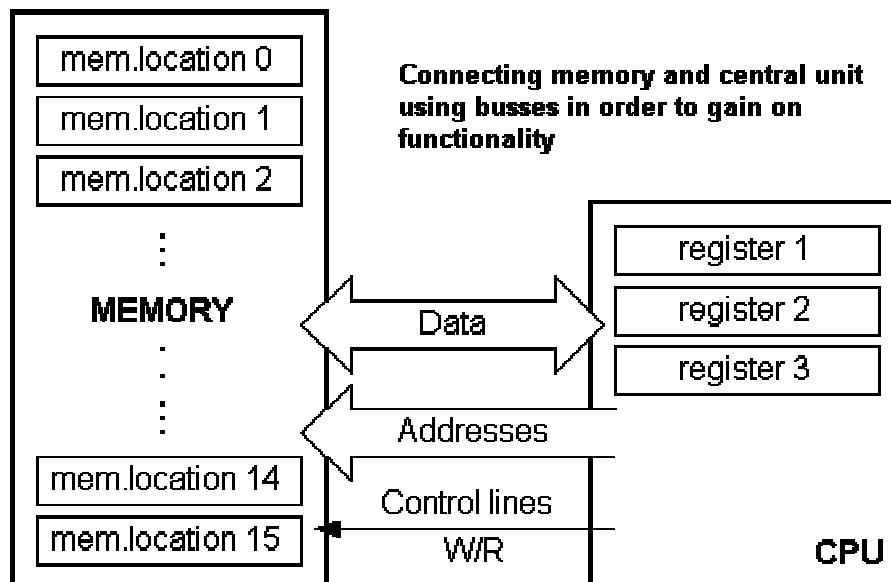
Let add 3 more memory locations to a specific block that will have a built in capability to multiply, divide, subtract, and move its contents from one memory location onto another. The part we just added in is called "central processing unit" (CPU). Its memory locations are called registers.



Registers are therefore memory locations whose role is to help with performing various mathematical operations or any other operations with data wherever data can be found. Look at the current situation. We have two independent entities (memory and CPU) which are interconnected, and thus any exchange of data is hindered, as well as its functionality. If, for example, we wish to add the contents of two memory locations and return the result again back to memory, we would need a connection between memory and CPU. Simply stated, we must have some "way" through data goes from one block to another.

1.3 Bus

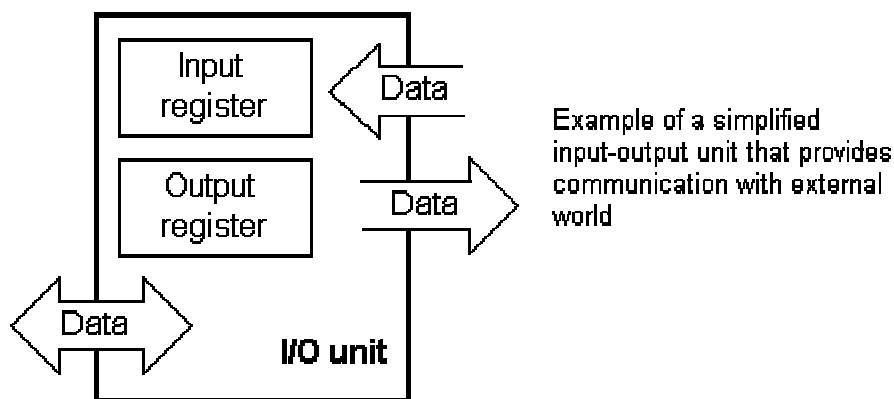
That "way" is called "bus". Physically, it represents a group of 8, 16, or more wires. There are two types of buses: address and data bus. The first one consists of as many lines as the amount of memory we wish to address, and the other one is as wide as data, in our case 8 bits or the connection line. First one serves to transmit address from CPU memory, and the second to connect all blocks inside the microcontroller.



As far as functionality, the situation has improved, but a new problem has also appeared: we have a unit that's capable of working by itself, but which does not have any contact with the outside world, or with us! In order to remove this deficiency, let's add a block which contains several memory locations whose one end is connected to the data bus, and the other has connection with the output lines on the microcontroller which can be seen as pins on the electronic component.

1.4 Input-output unit

Those locations we've just added are called "ports". There are several types of ports : input, output or bidirectional ports. When working with ports, first of all it is necessary to choose which port we need to work with, and then to send data to, or take it from the port.

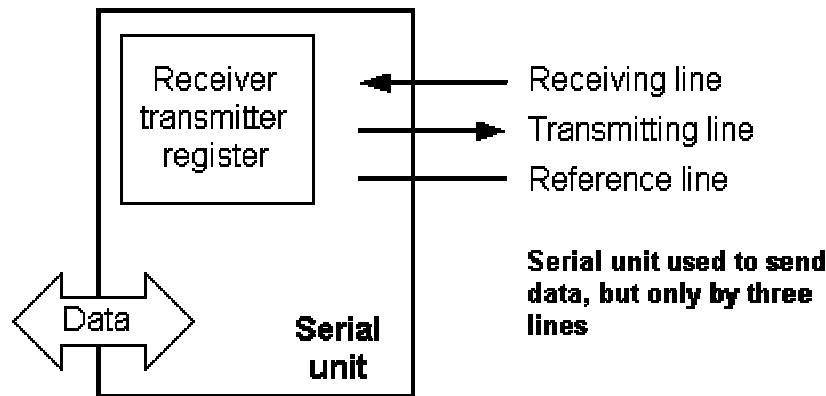


When working with it the port acts like a memory location. Something is simply being written into or read from it, and it could be noticed on the pins of the microcontroller.

1.5 Serial communication

Beside stated above we've added to the already existing unit the possibility of communication with an outside world. However, this way of communicating has its drawbacks. One of the basic drawbacks is the number of lines which need to be used in order to transfer data. What if it is being transferred to a distance of several kilometers? The number of lines times number of kilometers doesn't promise the economy of the project. It leaves us having to reduce the number of lines in such a way that we don't lessen its functionality. Suppose we are working with three lines only, and that one line is used for sending data, other for receiving, and the third one is used as a reference line for both the input and the output side. In order for this to work, we need to set the rules of exchange of data. These rules are called protocol. Protocol is therefore defined in advance so there wouldn't be any misunderstanding between the sides that are communicating with each other. For example, if one man is speaking in French, and the other in English, it is highly unlikely that they will quickly and effectively understand each other. Let's suppose we have the following protocol. The logical unit "1" is set up on the transmitting line until transfer begins. Once the transfer starts, we lower the transmission line to logical "0" for a period of time (which we will designate as T), so the receiving side will know that it is receiving data, and so it will activate its mechanism for reception.

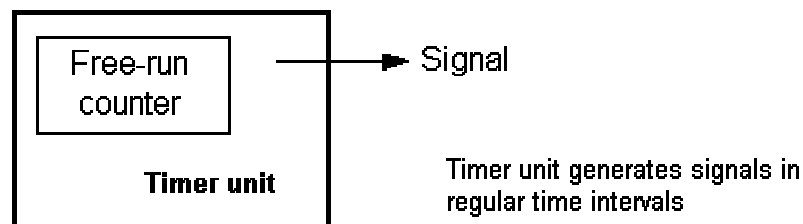
Let's go back now to the transmission side and start putting logic zeros and ones onto the transmitter line in the order from a bit of the lowest value to a bit of the highest value. Let each bit stay on line for a time period which is equal to T , and in the end, or after the 8th bit, let us bring the logical unit "1" back on the line which will mark the end of the transmission of one data. The protocol we've just described is called in professional literature NRZ (Non-Return to Zero).



As we have separate lines for receiving and sending, it is possible to receive and send data (info.) at the same time. So called full-duplex mode block which enables this way of communication is called a serial communication block. Unlike the parallel transmission, data moves here bit by bit, or in a series of bits what defines the term serial communication comes from. After the reception of data we need to read it from the receiving location and store it in memory as opposed to sending where the process is reversed. Data goes from memory through the bus to the sending location, and then to the receiving unit according to the protocol.

1.6 Timer unit

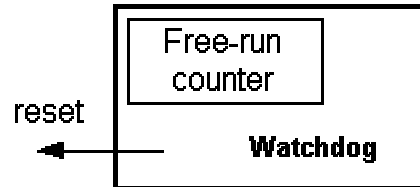
Since we have the serial communication explained, we can receive, send and process data.



However, in order to utilize it in industry we need a few additionally blocks. One of those is the timer block which is significant to us because it can give us information about time, duration, protocol etc. The basic unit of the timer is a free-run counter which is in fact a register whose numeric value increments by one in even intervals, so that by taking its value during periods T_1 and T_2 and on the basis of their difference we can determine how much time has elapsed. This is a very important part of the microcontroller whose understanding requires most of our time.

1.7 Watchdog

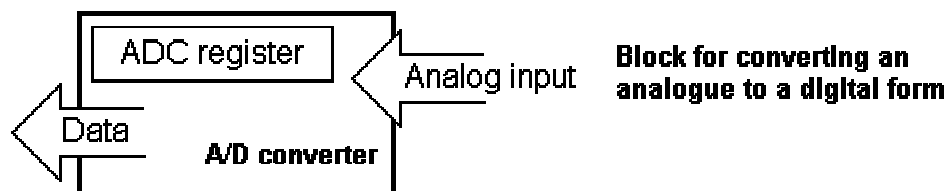
One more thing is requiring our attention is a flawless functioning of the microcontroller during its run-time. Suppose that as a result of some interference (which often does occur in industry) our microcontroller stops executing the program, or worse, it starts working incorrectly.



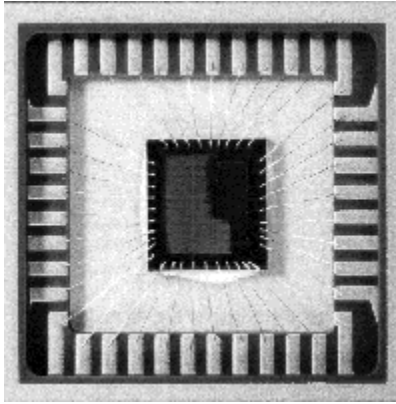
Of course, when this happens with a computer, we simply reset it and it will keep working. However, there is no reset button we can push on the microcontroller and thus solve our problem. To overcome this obstacle, we need to introduce one more block called watchdog. This block is in fact another free-run counter where our program needs to write a zero in every time it executes correctly. In case that program gets "stuck", zero will not be written in, and counter alone will reset the microcontroller upon achieving its maximum value. This will result in executing the program again, and correctly this time around. That is an important element of every program to be reliable without man's supervision.

1.8 Analog to Digital Converter

As the peripheral signals usually are substantially different from the ones that microcontroller can understand (zero and one), they have to be converted into a pattern which can be comprehended by a microcontroller. This task is performed by a block for analog to digital conversion or by an ADC. This block is responsible for converting an information about some analog value to a binary number and for follow it through to a CPU block so that CPU block can further process it.

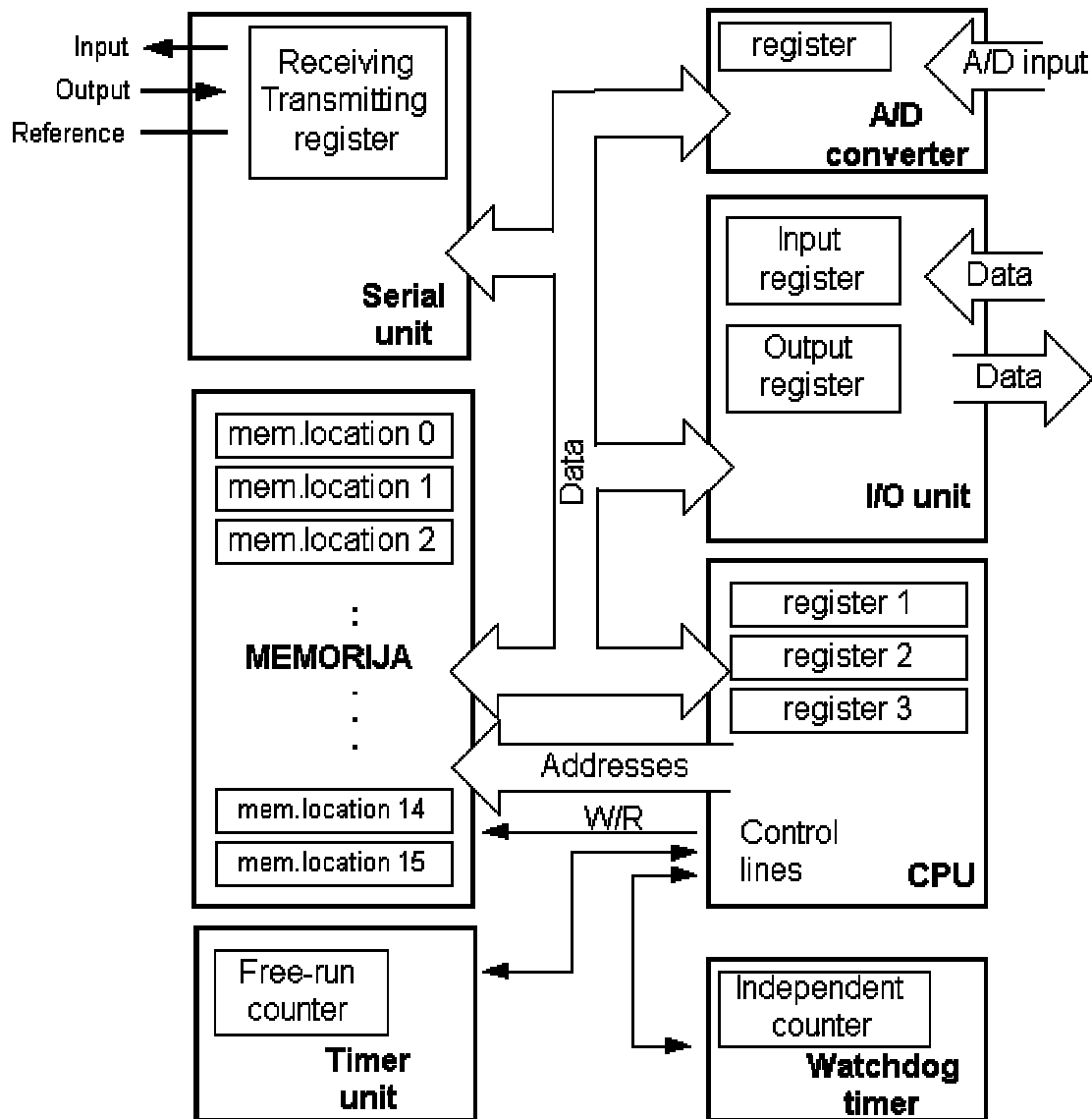


Finally, the microcontroller is now completed, and all we need to do now is to assemble it into an electronic component where it will access inner blocks through the outside pins. The picture below shows what a microcontroller looks like inside.



Physical configuration of the interior of a microcontroller

Thin lines which lead from the center towards the sides of the microcontroller represent wires connecting inner blocks with the pins on the housing of the microcontroller so called bonding lines. Chart on the following page represents the center section of a microcontroller.



Microcontroller outline with its basic elements and internal connections

For a real application, a microcontroller alone is not enough. Beside a microcontroller, we need a program that would be executed, and a few more elements which make up a interface logic towards the elements of regulation (which will be discussed in later chapters).

1.9 Program

Program writing is a special field of work with microcontrollers and is called "programming". Try to write a small program in a language that we will make up ourselves first and then would be understood by anyone.

START

REGISTER1=MEMORY LOCATION_A

REGISTER2=MEMORY LOCATION_B

PORTA=REGISTER1 + REGISTER2

END

The program adds the contents of two memory locations, and views their sum on port A. The first line of the program stands for moving the contents of memory location "A" into one of the registers of central processing unit. As we need the other data as well, we will also move it into the other register of the central processing unit. The next instruction instructs the central processing unit to add the contents of those two registers and send a result to port A, so that sum of that addition would be visible to the outside world. For a more complex problem, program that works on its solution will be bigger.

Programming can be done in several languages such as Assembler, C and Basic which are most commonly used languages. Assembler belongs to lower level languages that are programmed slowly, but take up the least amount of space in memory and gives the best results where the speed of program execution is concerned. As it is the most commonly used language in programming microcontrollers it will be discussed in a later chapter. Programs in C language are easier to be written, easier to be understood, but are slower in executing from assembler programs. Basic is the easiest one to learn, and its instructions are nearest a man's way of reasoning, but like C programming language it is also slower than assembler. In any case, before you make up your mind about one of these languages you need to consider carefully the demands for execution speed, for the size of memory and for the amount of time available for its assembly.

After the program is written, we would install the microcontroller into a device and run it. In order to do this we need to add a few more external components necessary for its work. First we must give life to a microcontroller by connecting it to a power supply (power needed for operation of all electronic instruments) and oscillator whose role is similar to the role that heart plays in a human body. Based on its clocks microcontroller executes instructions of a program. As it receives supply microcontroller will perform a small check up on itself, look up the beginning of the program and start executing it. How the device will work depends on many parameters, the most important of which is the skillfulness of the developer of hardware, and on programmer's expertise in getting the maximum out of the device with his program.

CHAPTER 2

Microcontroller PIC16F84

[Introduction](#)

[CISC, RISC](#)

[Applications](#)

[Clock/instruction cycle](#)

[Pipelining](#)

[Pin description](#)

[2.1 Clock generator - oscillator](#)

[2.2 Reset](#)

[2.3 Central processing unit](#)

[2.4 Ports](#)

[2.5 Memory organization](#)

[2.6 Interrupts](#)

[2.7 Free timer TMR0](#)

[2.8 EEPROM Data memory](#)

Introduction

PIC16F84 belongs to a class of 8-bit microcontrollers of RISC architecture. Its general structure is shown on the following map representing basic blocks.

Program memory (FLASH)- for storing a written program.
Since memory made in FLASH technology can be programmed and cleared more than once, it makes this microcontroller suitable for device development.

EEPROM - data memory that needs to be saved when there is no supply.
It is usually used for storing important data that must not be lost if power supply suddenly stops. For instance, one such data is an assigned temperature in temperature regulators. If during a loss of power supply this data was lost, we would have to make the adjustment once again upon return of supply. Thus our device loses on self-reliance.

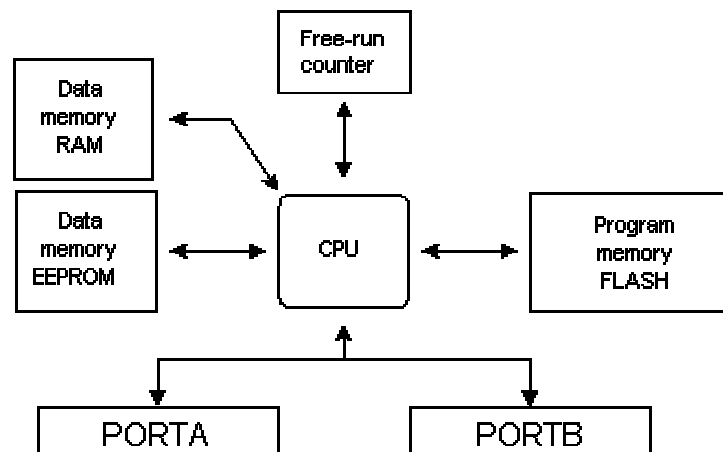
RAM - data memory used by a program during its execution.
In RAM are stored all inter-results or temporary data during run-time.

PORTA and PORTB are physical connections between the microcontroller and the outside world. Port A has five, and port B has eight pins.

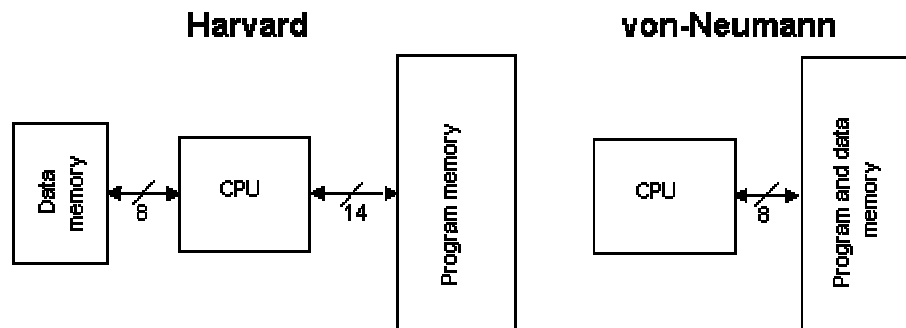
FREE-RUN TIMER is an 8-bit register inside a microcontroller that works independently of the program. On every fourth clock of the oscillator it increments its

value until it reaches the maximum (255), and then it starts counting over again from zero. As we know the exact timing between each two increments of the timer contents, timer can be used for measuring time which is very useful with some devices.

CENTRAL PROCESSING UNIT has a role of connective element between other blocks in the microcontroller. It coordinates the work of other blocks and executes the user program.



PIC16F84 microcontroller outline



Harvard vs. von Neuman Block Architectures

CISC, RISC

It has already been said that PIC16F84 has a RISC architecture. This term is often found in computer literature, and it needs to be explained here in more detail. Harvard architecture is a newer concept than von-Neumann's. It rose out of the need to speed up the work of a microcontroller. In Harvard architecture, data bus and

address bus are separate. Thus a greater flow of data is possible through the central processing unit, and of course, a greater speed of work. Separating a program from data memory makes it further possible for instructions not to have to be 8-bit words. PIC16F84 uses 14 bits for instructions which allows for all instructions to be one word instructions. It is also typical for Harvard architecture to have fewer instructions than von-Neumann's, and to have instructions usually executed in one cycle.

Microcontrollers with Harvard architecture are also called "RISC microcontrollers". RISC stands for Reduced Instruction Set Computer. Microcontrollers with von-Neumann's architecture are called 'CISC microcontrollers'. Title CISC stands for Complex Instruction Set Computer.

Since PIC16F84 is a RISC microcontroller, that means that it has a reduced set of instructions, more precisely 35 instructions. (ex. Intel's and Motorola's microcontrollers have over hundred instructions) All of these instructions are executed in one cycle except for jump and branch instructions. According to what its maker says, PIC16F84 usually reaches results of 2:1 in code compression and 4:1 in speed in relation to other 8-bit microcontrollers in its class.

Applications

PIC16F84 perfectly fits many uses, from automotive industries and controlling home appliances to industrial instruments, remote sensors, electrical door locks and safety devices. It is also ideal for smart cards as well as for battery supplied devices because of its low consumption.

EEPROM memory makes it easier to apply microcontrollers to devices where permanent storage of various parameters is needed (codes for transmitters, motor speed, receiver frequencies, etc.). Low cost, low consumption, easy handling and flexibility make PIC16F84 applicable even in areas where microcontrollers had not previously been considered (example: timer functions, interface replacement in larger systems, coprocessor applications, etc.).

In System Programmability of this chip (along with using only two pins in data transfer) makes possible the flexibility of a product, after assembling and testing have been completed. This capability can be used to create assembly-line production, to store calibration data available only after final testing, or it can be used to improve programs on finished products.

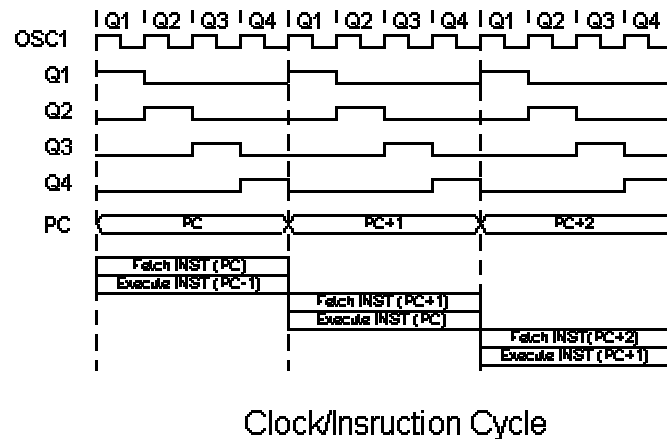
Clock / instruction cycle

Clock is microcontroller's main starter, and is obtained from an external component called an "oscillator". If we want to compare a microcontroller with a time clock, our "clock" would then be a ticking sound we hear from the time clock. In that case, oscillator could be compared to a spring that is wound so time clock can run. Also, force used to wind the time clock can be compared to an electrical supply.

Clock from the oscillator enters a microcontroller via OSC1 pin where internal circuit of a microcontroller divides the clock into four even clocks Q1, Q2, Q3, and Q4 which

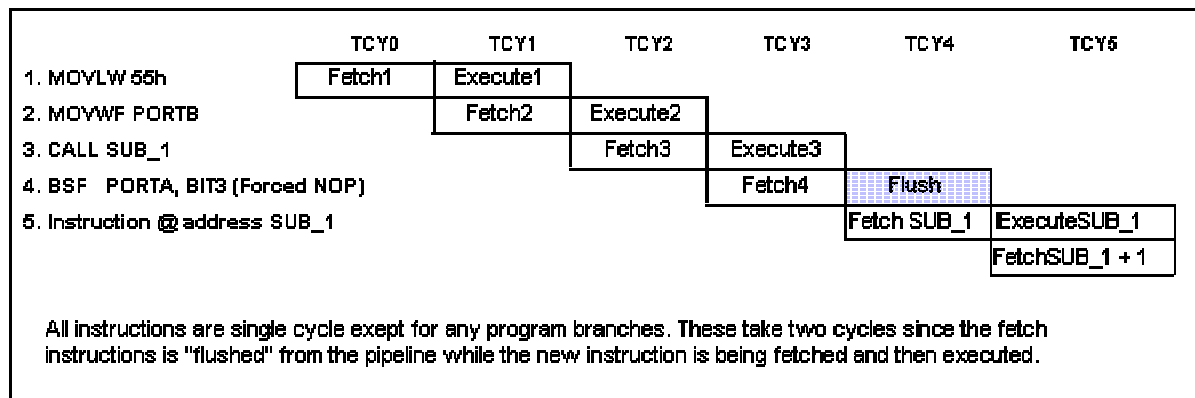
do not overlap. These four clocks make up one instruction cycle (also called machine cycle) during which one instruction is executed.

Execution of instruction starts by calling an instruction that is next in string. Instruction is called from program memory on every Q1 and is written in instruction register on Q4. Decoding and execution of instruction are done between the next Q1 and Q4 cycles. On the following diagram we can see the relationship between instruction cycle and clock of the oscillator (OSC1) as well as that of internal clocks Q1-Q4. Program counter (PC) holds information about the address of the next instruction.



Pipelining

Instruction cycle consists of cycles Q1, Q2, Q3 and Q4. Cycles of calling and executing instructions are connected in such a way that in order to make a call, one instruction cycle is needed, and one more is needed for decoding and execution. However, due to pipelining, each instruction is effectively executed in one cycle. If instruction causes a change on program counter, and PC doesn't point to the following but to some other address (which can be the case with jumps or with calling subprograms), two cycles are needed for executing an instruction. This is so because instruction must be processed again, but this time from the right address. Cycle of calling begins with Q1 clock, by writing into instruction register (IR). Decoding and executing begins with Q2, Q3 and Q4 clocks.



Instruction Pipeline Flow

TCY0 reads in instruction MOVLW 55h (it doesn't matter to us what instruction was executed, because there is no rectangle pictured on the bottom).

TCY1 executes instruction MOVLW 55h and reads in MOVWF PORTB.

TCY2 executes MOVWF PORTB and reads in CALL SUB_1.

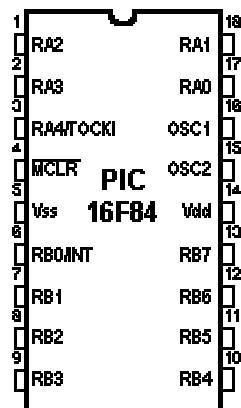
TCY3 executes a call of a subprogram CALL SUB_1, and reads in instruction BSF PORTA, BIT3. As this instruction is not the one we need, or is not the first instruction of a subprogram SUB_1 whose execution is next in order, instruction must be read in again. This is a good example of an instruction needing more than one cycle.

TCY4 instruction cycle is totally used up for reading in the first instruction from a subprogram at address SUB_1.

TCY5 executes the first instruction from a subprogram SUB_1 and reads in the next one.

Pin description

PIC16F84 has a total of 18 pins. It is most frequently found in a DIP18 type of case but can also be found in SMD case which is smaller from a DIP. DIP is an abbreviation for Dual In Package. SMD is an abbreviation for Surface Mount Devices suggesting that holes for pins to go through when mounting, aren't necessary in soldering this type of a component.



Pins on PIC16F84 microcontroller have the following meaning:

Pin no.1 **RA2** Second pin on port A. Has no additional function

Pin no.2 **RA3** Third pin on port A. Has no additional function.

Pin no.3 **RA4** Fourth pin on port A. TOCK1 which functions as a timer is also found on this pin

Pin no.4 **MCLR** Reset input and Vpp programming voltage of a microcontroller

Pin no.5 **Vss** Ground of power supply.

Pin no.6 **RB0** Zero pin on port B. Interrupt input is an additional function.

Pin no.7 **RB1** First pin on port B. No additional function.

Pin no.8 **RB2** Second pin on port B. No additional function.

Pin no.9 **RB3** Third pin on port B. No additional function.

Pin no.10 **RB4** Fourth pin on port B. No additional function.

Pin no.11 **RB5** Fifth pin on port B. No additional function.

Pin no.12 **RB6** Sixth pin on port B. 'Clock' line in program mode.

Pin no.13 **RB7** Seventh pin on port B. 'Data' line in program mode.

Pin no.14 **Vdd** Positive power supply pole.

Pin no.15 **OSC2** Pin assigned for connecting with an oscillator

Pin no.16 **OSC1** Pin assigned for connecting with an oscillator

Pin no.17 **RA2** Second pin on port A. No additional function

Pin no.18 **RA1** First pin on port A. No additional function.

2.1 Clock generator - oscillator

Oscillator circuit is used for providing a microcontroller with a clock. Clock is needed so that microcontroller could execute a program or program instructions.

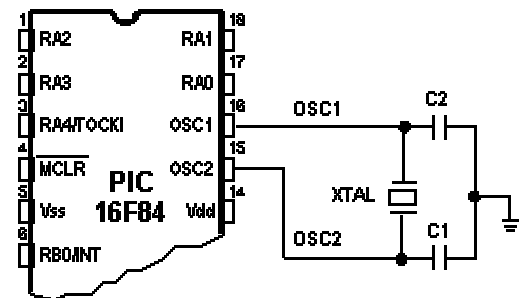
Types of oscillators

PIC16F84 can work with four different configurations of an oscillator. Since configurations with crystal oscillator and resistor-capacitor (RC) are the ones that are used most frequently, these are the only ones we will mention here. Microcontroller type with a crystal oscillator has in its designation XT, and a microcontroller with resistor-capacitor pair has a designation RC. This is important because you need to mention the type of oscillator when buying a microcontroller.

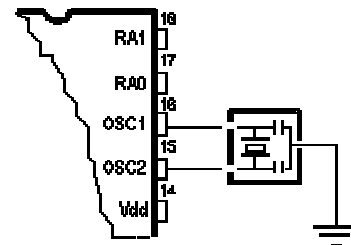
XT Oscillator

Crystal oscillator is kept in metal housing with two pins where you have written down the frequency at which crystal oscillates. One ceramic capacitor of 30pF whose other end is connected to the ground needs to be connected with each pin.

Oscillator and capacitors can be packed in joint case with three pins. Such element is called ceramic resonator and is represented in charts like the one below. Center pins of the element is the ground, while end pins are connected with OSC1 and OSC2 pins on the microcontroller. When designing a device, the rule is to place an oscillator nearer a microcontroller, so as to avoid any interference on lines on which microcontroller is receiving a clock.



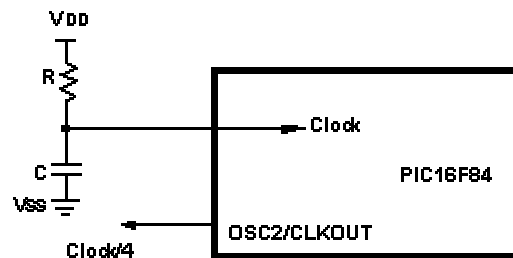
Connecting the quartz oscillator to give clock to a microcontroller



Connecting a resonator onto a microcontroller

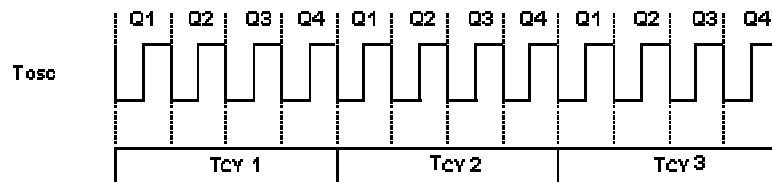
RC Oscillator

In applications where great time precision is not necessary, RC oscillator offers additional savings during purchase. Resonant frequency of RC oscillator depends on supply voltage rate, resistance R, capacity C and working temperature. It should be mentioned here that resonant frequency is also influenced by normal variations in process parameters, by tolerance of external R and C components, etc.



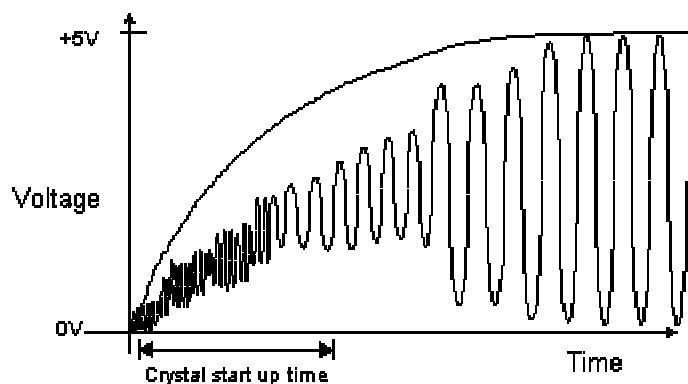
Note: This pin can be configured as input/output pin

Above diagram shows how RC oscillator is connected with PIC16F84. With value of resistor R being below 2.2k, oscillator can become unstable, or it can even stop the oscillation. With very high value of R (ex. 1M) oscillator becomes very sensitive to noise and humidity. It is recommended that value of resistor R should be between 3 and 100k. Even though oscillator will work without an external capacitor ($C=0pF$), capacitor above 20pF should still be used for noise and stability. No matter which oscillator is being used, in order to get a clock that microcontroller works upon, a clock of the oscillator must be divided by 4. Oscillator clock divided by 4 can also be obtained on OSC2/CLKOUT pin, and can be used for testing or synchronizing other logical circuits.



Relationship between a clock and a number of instruction cycles

Following a supply, oscillator starts oscillating. Oscillation at first has an unstable period and amplitude, but after some period of time it becomes stabilized.



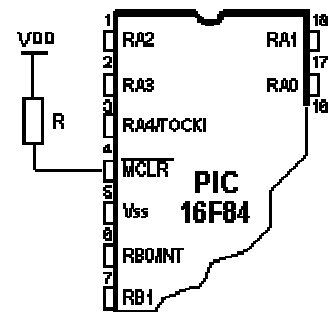
Signal of an oscillator clock after receiving the supply of a microcontroller

To prevent such inaccurate clock from influencing microcontroller's performance, we need to keep the microcontroller in reset state during stabilization of oscillator's clock. Diagram above shows a typical shape of a signal which microcontroller gets from the quartz oscillator.

2.2 Reset

Reset is used for putting the microcontroller into a 'known' condition. That practically means that microcontroller can behave rather inaccurately under certain undesirable conditions. In order to continue its proper functioning it has to be reset, meaning all registers would be placed in a starting position. Reset is not only used when microcontroller doesn't behave the way we want it to, but can also be used when trying out a device as an interrupt in program execution, or to get a microcontroller ready when loading a program.

In order to prevent from bringing a logical zero to MCLR pin accidentally (line above it means that reset is activated by a logical zero), MCLR has to be connected via resistor to the positive supply pole. Resistor should be between 5 and 10K. This kind of resistor whose function is to keep a certain line on a logical one as a preventive, is called a pull up.



Using the internal reset circuit

Microcontroller PIC16F84 knows several sources of resets:

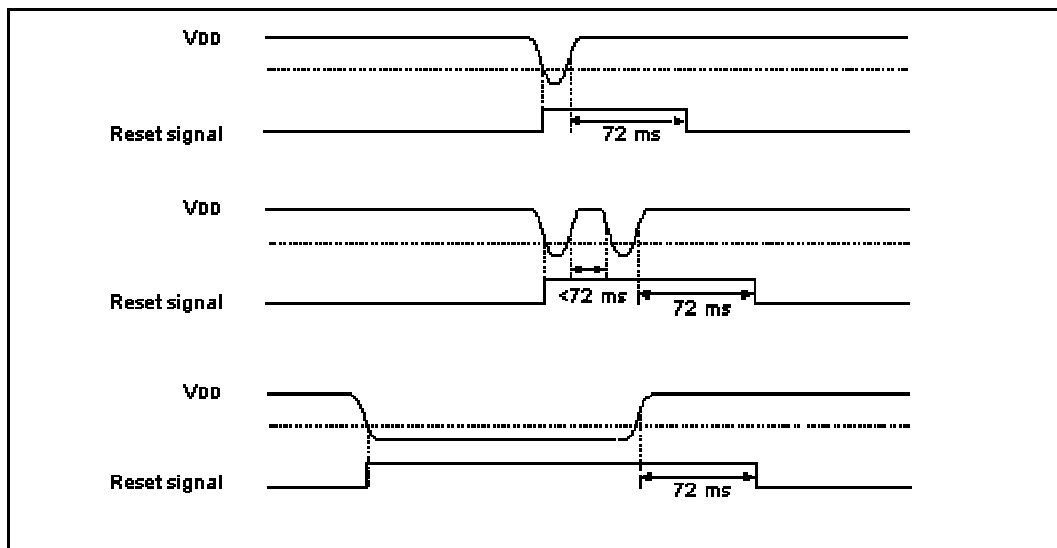
- Reset during power on, POR (Power-On Reset)
- Reset during regular work by bringing logical zero to MCLR microcontroller's pin.
- Reset during SLEEP regime
- Reset at watchdog timer (WDT) overflow
- Reset during at WDT overflow during SLEEP work regime.

The most important reset sources are a) and b). The first one occurs each time a power supply is brought to the microcontroller and serves to bring all registers to a starting position initial state. The second one is a product of purposeful bringing in of a logical zero to MCLR pin during normal operation of the microcontroller. This second one is often used in program development.

During a reset, RAM memory locations are not being reset. They are unknown during a power up and are not changed at any reset. Unlike these, SFR registers are reset to a starting position initial state. One of the most important effects of a reset is setting a program counter (PC) to zero (0000h) , which enables the program to start executing from the first written instruction.

Reset at supply voltage drop below the permissible (Brown-out Reset)

Impulse for resetting during voltage voltage-up is generated by microcontroller itself when it detects an increase in supply Vdd (in a range from 1.2V to 1.8V). That impulse lasts 72ms which is enough time for an oscillator to get stabilized. These 72ms are provided by an internal PWRT timer which has its own RC oscillator. Microcontroller is in a reset mode as long as PWRT is active. However, as device is working, problem arises when supply doesn't drop to zero but falls below the limit that guarantees microcontroller's proper functioning. This is a likely case in practice, especially in industrial environment where disturbances and instability of supply are an everyday occurrence. To solve this problem we need to make sure that microcontroller is in a reset state each time supply falls below the approved limit.



Examples of voltage supply drop below the proper level

If, according to electrical specification, internal reset circuit of a microcontroller can not satisfy the needs, special electronic components can be used which are capable of generating the desired reset signal. Beside this function, they can also function in watching over supply voltage. If voltage drops below specified level, a logical zero would appear on MCLR pin which holds the microcontroller in reset state until voltage is not within limits that guarantee accurate performance.

2.3 Central Processing Unit

Central processing unit (CPU) is the brain of a microcontroller. That part is responsible for finding and fetching the right instruction which needs to be executed, for decoding that instruction, and finally for its execution.

Central processing unit connects all parts of the microcontroller into one whole. Surely, its most important function is to decode program instructions. When programmer writes a program, instructions have a clear form like `MOVLW 0x20`. However, in order for a microcontroller to understand that, this 'letter' form of an instruction must be translated into a series of zeros and ones which is called an 'opcode'. This transition from a letter to binary form is done by translators such as assembler translator (also known as an assembler). Instruction thus fetched from program memory must be decoded by a central processing unit. We can then select from the table of all the instructions a set of actions which execute a assigned task defined by instruction. As instructions may within themselves contain assignments which require different transfers of data from one memory into another, from memory onto ports, or some other calculations, CPU must be connected with all parts of the microcontroller. This is made possible through a data bus and an address bus.

Arithmetic logic unit is responsible for performing operations of adding, subtracting, moving (left or right within a register) and logic operations. Moving data inside a register is also known as 'shifting'. PIC16F84 contains an 8-bit arithmetic logic unit and 8-bit work registers.

In instructions with two operands, ordinarily one operand is in work register (W register), and the other is one of the registers or a constant. By operand we mean the contents on which some operation is being done, and a register is any one of the GPR or SFR registers. GPR is an abbreviation for 'General Purposes Registers', and SFR for 'Special Function Registers'. In instructions with one operand, an operand is either W register or one of the registers. As an addition in doing operations in arithmetic and logic, ALU controls status bits (bits found in STATUS register). Execution of some instructions affects status bits, which depends on the result itself. Depending on which instruction is being executed, ALU can affect values of Carry (C), Digit Carry (DC), and Zero (Z) bits in STATUS register.

STATUS Register

R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	\overline{TO}	\overline{PD}	Z	DC	C

bit7

Legend:

R = Readable bit **W** = Writable bit

U = Unimplemented bit, read as '00' - n = Value at power-on reset

bit 7 **IRP** (Register Bank Select bit)

Bit whose role is to be an eighth bit for purposes of indirect addressing the internal

RAM.

1 = bank 2 and 3

0 = bank 0 and 1 (from 00h to FFh)

bits 6:5 **RP1:RP0** (Register Bank Select bits)

These two bits are upper part of the address for direct addressing. As instructions which address the memory directly have only seven bits, they need one more bit in order to address all 256 bytes which is how many bytes PIC16F84 has. RP1 bit is not used, but is left for some future expansions of this microcontroller.

01 = first bank

00 = zero bank

bit 4 **TO** Time-out ; Watchdog overflow.

Bit is set after turning on the supply and execution of CLRWDT and SLEEP instructions. Bit is reset when watchdog gets to the end signaling that overflow took place.

1 = overflow did not occur

0 = overflow did occur

bit 3 **PD** (Power-down bit)

This bit is set whenever power supply is brought to a microcontroller : as it starts running, after each regular reset and after execution of instruction CLRWDT. Instruction SLEEP resets it when microcontroller falls into low consumption mode. Its repeated setting is possible via reset or by turning the supply off/on . Setting can be triggered also by a signal on RB0/INT pin, change on RB port, upon writing to internal DATA EEPROM, and by a Watchdog.

1 = after supply has been turned on

0 = executing SLEEP instruction

bit 2 **Z** (Zero bit) Indication of a zero result

This bit is set when the result of an executed arithmetic or logic operation is zero.

1 = result equals zero

0 = result does not equal zero

bit 1 **DC** (Digit Carry) DC Transfer

Bit affected by operations of addition, subtraction. Unlike C bit, this bit represents transfer from the fourth resulting place. It is set in case of subtracting smaller from greater number and is reset in the other case.

1 = transfer occurred on the fourth bit according to the order of the result

0 = transfer did not occur

DC bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

bit 0 **C** (Carry) Transfer

Bit that is affected by operations of addition, subtraction and shifting.

1 = transfer occurred from the highest resulting bit

0 = transfer did not occur

C bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

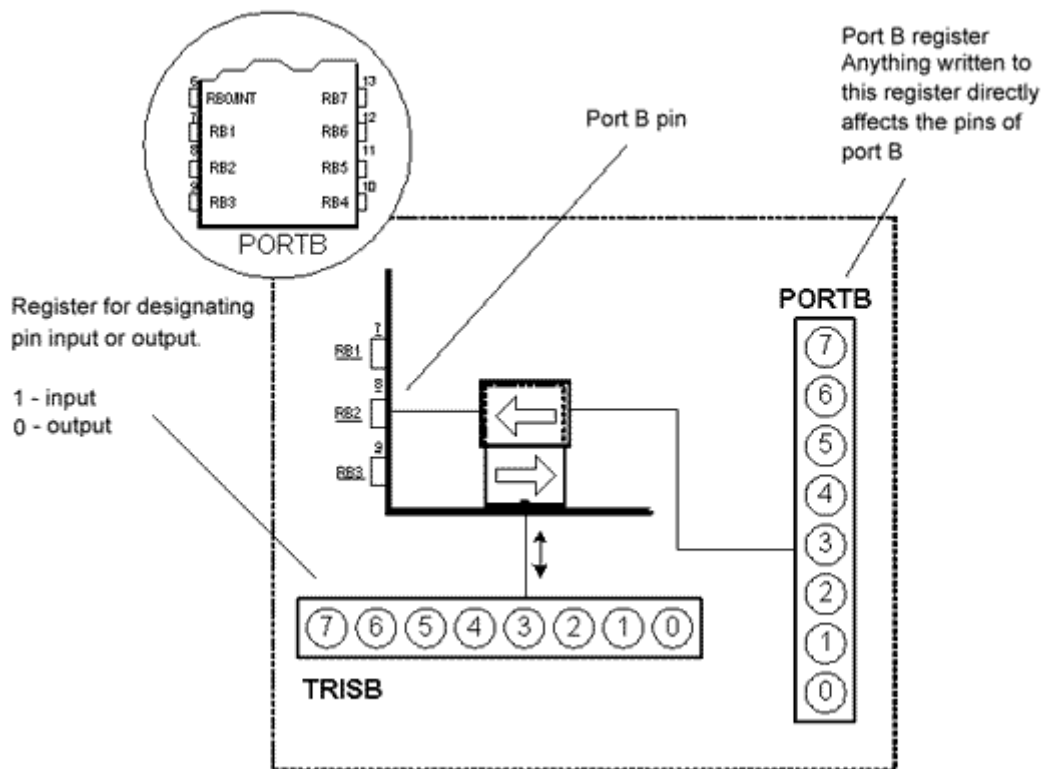
2.4 Ports

Term "port" refers to a group of pins on a microcontroller which can be accessed simultaneously, or on which we can set the desired combination of zeros and ones, or read from them an existing status. Physically, port is a register inside a microcontroller which is connected by wires to the pins of a microcontroller. Ports represent physical connection of Central Processing Unit with an outside world. Microcontroller uses them in order to monitor or control other components or devices. Due to functionality, some pins have twofold roles like PA4/T0CKI for instance, which is in the same time the fourth bit of port A and an external input for free-run counter. Selection of one of these two pin functions is done in one of the configuration registers. An illustration of this is the fifth bit T0CS in OPTION register. By selecting one of the functions the other one is disabled.

All port pins can be designated as input or output, according to the needs of a device that's being developed. In order to define a pin as input or output pin, the right combination of zeros and ones must be written in TRIS register. If the appropriate bit of TRIS register contains logical "1", then that pin is an input pin, and if the opposite is true, it's an output pin. Every port has its proper TRIS register. Thus, port A has TRISA, and port B has TRISB. Pin direction can be changed during the course of work which is particularly fitting for one-line communication where data flow constantly changes direction. PORTA and PORTB state registers are located in bank 0, while TRISA and TRISB pin direction registers are located in bank 1.

PORTB and TRISB

PORTB has adjoined 8 pins. The appropriate register for data direction is TRISB. Setting a bit in TRISB register defines the corresponding port pin as input, and resetting a bit in TRISB register defines the corresponding port pin as output.



Each PORTB pin has a weak internal pull-up resistor (resistor which defines a line to logic one) which can be activated by resetting the seventh bit RBPU in OPTION register. These 'pull-up' resistors are automatically being turned off when port pin is configured as an output. When a microcontroller is started, pull-ups are disabled.

Four pins PORTB, RB7:RB4 can cause an interrupt which occurs when their status changes from logical one into logical zero and opposite. Only pins configured as input can cause this interrupt to occur (if any RB7:RB4 pin is configured as an output, an interrupt won't be generated at the change of status.) This interrupt option along with internal pull-up resistors makes it easier to solve common problems we find in practice like for instance that of matrix keyboard. If rows on the keyboard are connected to these pins, each push on a key will then cause an interrupt. A microcontroller will determine which key is at hand while processing an interrupt. It is not recommended to refer to port B at the same time that interrupt is being processed.

```

bsf    STATUS, RP0    ;Bank1
movlw  0x0F           ;Defining input and output pins
movwf  TRISB          ;Writing to TRISB register
bcf    STATUS, RP0    ;Bank0
bsf    PORTB, 4        ;PORTB <7:4>=0
bsf    PORTB, 5
bsf    PORTB, 6
bsf    PORTB, 7

```

The above example shows how pins 0, 1, 2, and 3 are designated input, and pins 4, 5, 6, and 7 for output, after which PORTB output pins are set to one.

PORTA and TRISA

PORTA has 5 adjoining pins. The corresponding register for data direction is TRISA at address 85h. Like with port B, setting a bit in TRISA register defines also the corresponding port pin as input, and clearing a bit in TRISA register defines the corresponding port pin as output.

It is important to note that PORTA pin RA4 can be input only. On that pin is also situated an external input for timer TMR0. Whether RA4 will be a standard input or an input for a counter depends on T0CS bit (*TMR0 Clock Source Select bit*). This pin enables the timer TMR0 to increment either from internal oscillator or via external impulses on RA4/T0CKI pin.

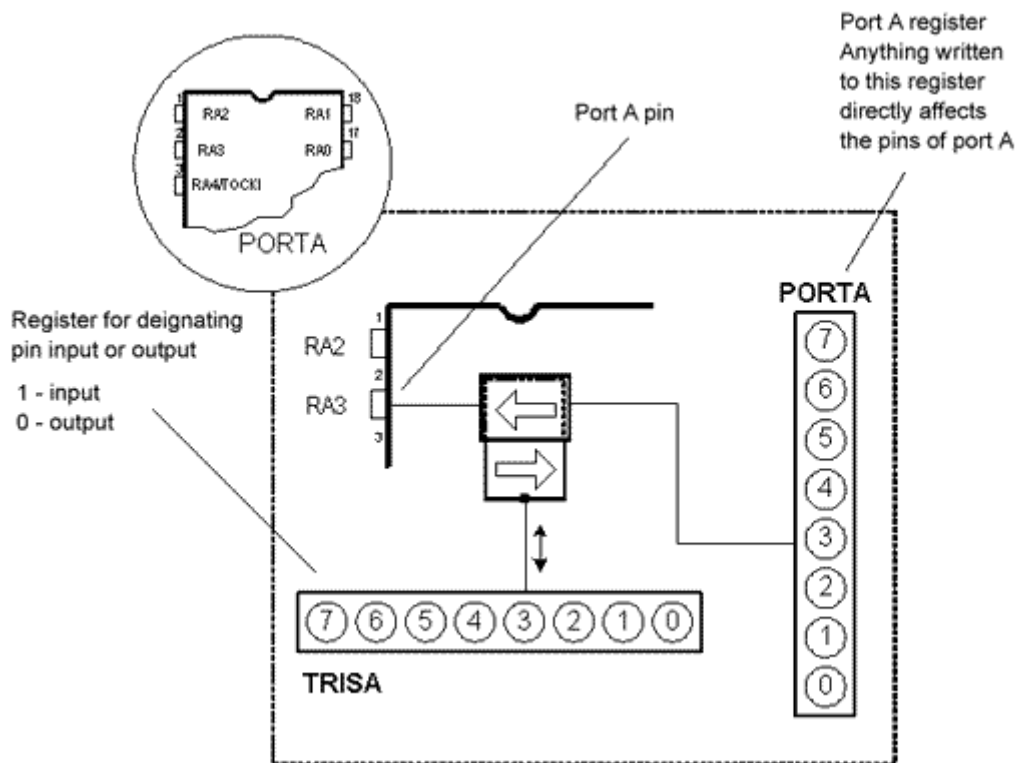


RA4 pin can be designated output, but in that case it has to be externally connected to PULL-UP resistor.

Configuring port A:

```
bsf      STATUS, RP0      ;Bank1
movlw    b'11111100'      ;Defining input and output pins
movwf    TRISA            ;Writing to TRISA register
bcf      STATUS, RP0      ;Bank0
```

Example shows how pins 0, 1, 2, 3, and 4 are designated input, and pins 5, 6, and 7 output. After this, it is possible to read the pins RA2, RA3, RA4, and to set logical zero or one to pins RA0 and RA1.



2.5 Memory organization

PIC16F84 has two separate memory blocks, one for data and the other for program. EEPROM memory with GPR and SFR registers in RAM memory make up the data block, while FLASH memory makes up the program block.

Program memory

Program memory has been carried out in FLASH technology which makes it possible to program a microcontroller many times before it's installed into a device, and even after its installment if eventual changes in program or process parameters should occur. The size of program memory is 1024 locations with 14 bits width where locations zero and four are reserved for reset and interrupt vector.

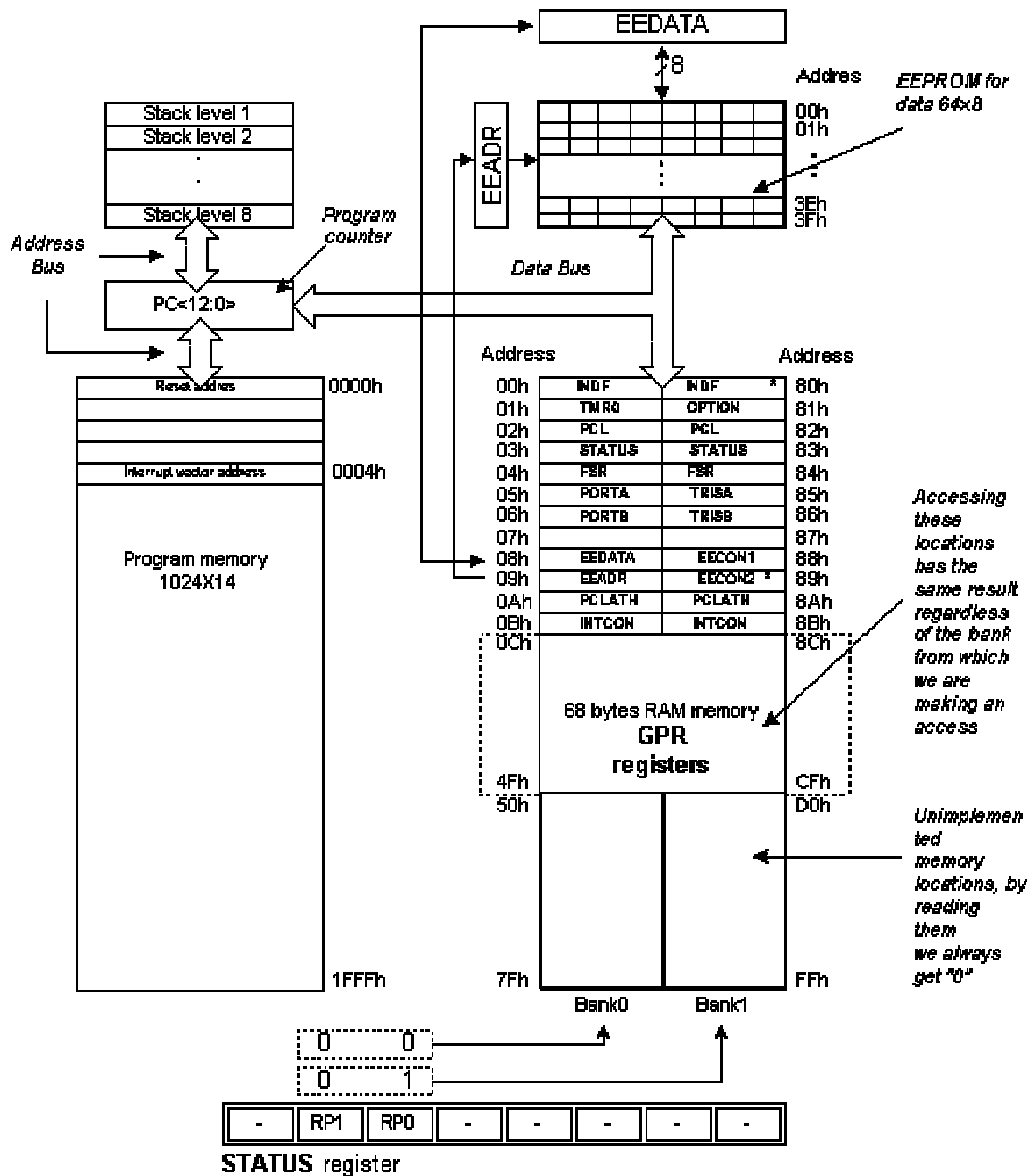
Data memory

Data memory consists of EEPROM and RAM memories. EEPROM memory consists of 64 eight bit locations whose contents is not lost during loosing of power supply. EEPROM is not directly addressable, but is accessed indirectly through EEADR and EEDATA registers. As EEPROM memory usually serves for storing important parameters (for example, of a given temperature in temperature regulators), there is a strict procedure for writing in EEPROM which must be followed in order to avoid

accidental writing. RAM memory for data occupies space on a memory map from location 0x0C to 0x4F which comes to 68 locations. Locations of RAM memory are also called GPR registers which is an abbreviation for *General Purpose Registers*. GPR registers can be accessed regardless of which bank is selected at the moment.

SFR registers

Registers which take up first 12 locations in banks 0 and 1 are registers of specialized function assigned with certain blocks of the microcontroller. These are called *Special Function Registers*.



Memory organization of microcontroller PIC16F84

Memory Banks

Beside this 'length' division to SFR and GPR registers, memory map is also divided in 'width' (see preceding map) to two areas called 'banks'. Selecting one of the banks is done via **RP0** bit in **STATUS** register.

Example:

```
bcf STATUS, RP0
```

Instruction BCF clears bit RP0 (RP0=0) in STATUS register and thus sets up bank 0.

```
bsf STATUS, RP0
```

Instruction BSF sets the bit RP0 (RP0=1) in STATUS register and thus sets up bank1.

It is useful to consider what would happen if the wrong bank was selected. Let's assume that we have selected bank 0 at the beginning of the program, and that we now want to write to certain register located in bank 1, say TRISB. Although we specified the name of the register TRISB, data will be actually stored to a bank 0 register at the appropriate address, which is PORTB in our example.

```
BANK0 macro
    Bcf STATUS, RP0    ;Select
memory bank 0
endm

BANK1 macro
    Bsf STATUS, RP0    ;Select
memory bank 1
endm
```

Bank selection can be also made via directive *banksel* after which name of the register to be accessed is specified. In this manner, there is no need to memorize which register is in which bank.



Locations 0Ch - 4Fh are general purpose registers (GPR) which are used as RAM memory. When locations 8Ch - CFh in Bank 1 are accessed, we actually access the exact same locations in Bank 0. In other words, whenever you wish to access one of the GPR registers, there is no need to worry about which bank we are in!

Program Counter

Program counter (PC) is a 13-bit register that contains the address of the instruction being executed. It is physically carried out as a combination of a 5-bit register PCLATH for the five higher bits of the address, and the 8-bit register PCL for the lower 8 bits of the address.

By its incrementing or change (i.e. in case of jumps) microcontroller executes program instructions step-by-step.

Stack

PIC16F84 has a 13-bit stack with 8 levels, or in other words, a group of 8 memory locations, 13 bits wide, with special purpose. Its basic role is to keep the value of program counter after a jump from the main program to an address of a subprogram. In order for a program to know how to go back to the point where it started from, it has to return the value of a program counter from a stack. When moving from a program to a subprogram, program counter is being pushed onto a stack (example of this is CALL instruction). When executing instructions such as RETURN, RETLW or

RETFIE which were executed at the end of a subprogram, program counter was taken from a stack so that program could continue where was stopped before it was interrupted. These operations of placing on and taking off from a program counter stack are called PUSH and POP, and are named according to similar instructions on some bigger microcontrollers.

In System Programming

In order to program a program memory, microcontroller must be set to special working mode by bringing up MCLR pin to 13.5V, and supply voltage V_{dd} has to be stabilized between 4.5V to 5.5V. Program memory can be programmed serially using two 'data/clock' pins which must previously be separated from device lines, so that errors wouldn't come up during programming.

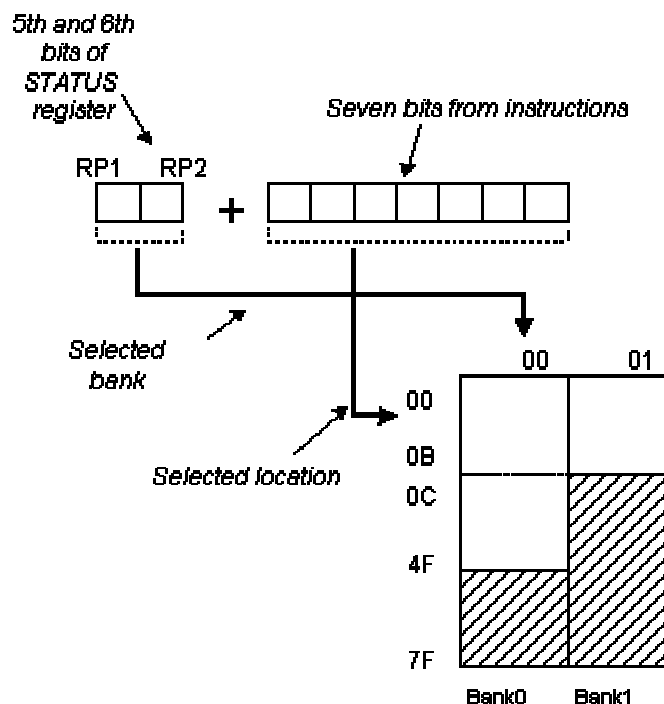
Addressing modes

RAM memory locations can be accessed directly or indirectly.

Direct Addressing

Direct Addressing is done through a 9-bit address. This address is obtained by connecting 7th bit of direct address of an instruction with two bits (RP1, RP0) from STATUS register as is shown on the following picture. Any access to SFR registers is an example of direct addressing.

```
Bsf STATUS, RP0 ;Bank1
movlw 0xFF      ;w=0xFF
movwf TRISA     ;address of TRISA register is taken from
                ;instruction movwf
```

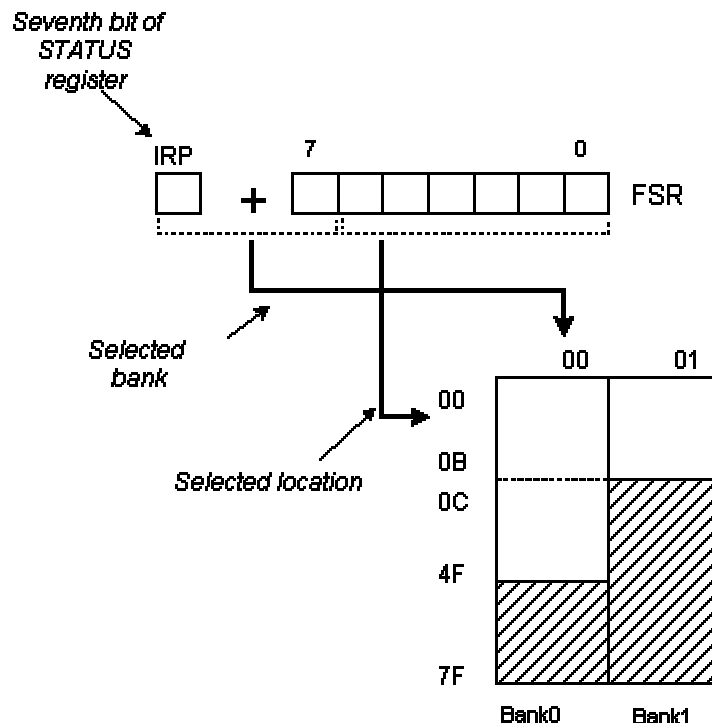


Direct addressing

Indirect Addressing

Indirect unlike direct addressing does not take an address from an instruction but derives it from IRP bit of STATUS and FSR registers. Addressed location is accessed via INDF register which in fact holds the address indicated by a FSR. In other words, any instruction which uses INDF as its register in reality accesses data indicated by a FSR register. Let's say, for instance, that one general purpose register (GPR) at address 0Fh contains a value of 20. By writing a value of 0Fh in FSR register we will get a register indicator at address 0Fh, and by reading from INDF register, we will get a value of 20, which means that we have read from the first register its value without accessing it directly (but via FSR and INDF). It appears that this type of addressing does not have any advantages over direct addressing, but certain needs do exist during programming which can be solved smoothly only through indirect addressing.

Indirect addressing is very convenient for manipulating data arrays located in GPR registers. In this case, it is necessary to initialize FSR register with a starting address of the array, and the rest of the data can be accessed by incrementing the FSR register.



Indirect addressing

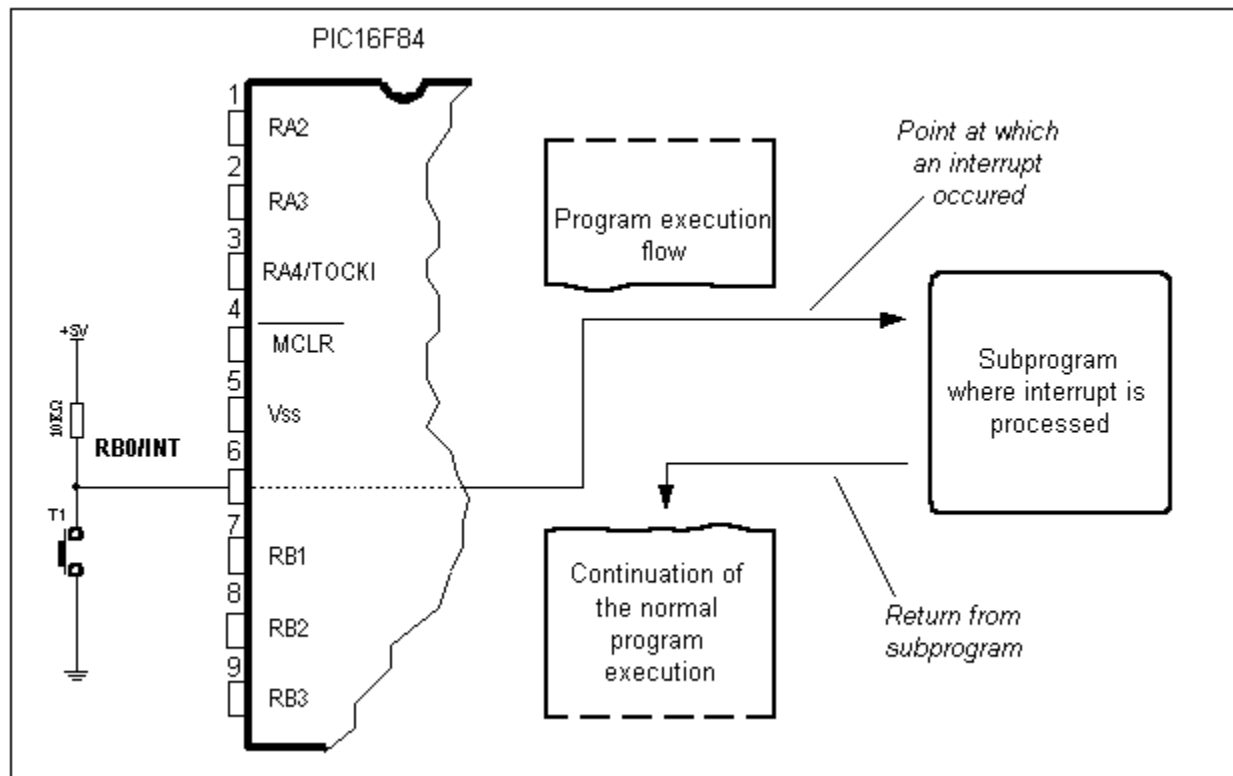
Such examples include sending a set of data via serial communication, working with buffers and indicators (which will be discussed further in a chapter with examples), or erasing a part of RAM memory (16 locations) as in the following instance.

```
        Movlw 0x0C          ;initialization of starting address
        Movwf FSR           ;FSR indicates address 0x0C
LOOP    clrf INDF           ;INDF = 0
        incf FSR            ;address = initial address + 1
        btfss FSR,4         ;are all locations erased
        goto loop          ;no, go through a loop again
CONTINUE
        :                  ; yes, continue with program
```

Reading data from INDF register when the contents of FSR register is equal to zero returns the value of zero, and writing to it results in NOP operation (no operation).

2.6 Interrupts

Interrupts are a mechanism of a microcontroller which enables it to respond to some events at the moment they occur, regardless of what microcontroller is doing at the time. This is a very important part, because it provides connection between a microcontroller and environment which surrounds it. Generally, each interrupt changes the program flow, interrupts it and after executing an interrupt subprogram (interrupt routine) it continues from that same point on.



One of the possible sources of interrupt and how it affects the main program

Control register of an interrupt is called INTCON and can be accessed regardless of the bank selected. Its role is to allow or disallow interrupts, and in case they are not allowed, it registers single interrupt requests through its own bits.

INTCON Register

R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0

GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
-----	------	------	------	------	------	------	------

bit7

Legend:

R = Readable bit **W** = Writable bit

U = Unimplemented bit, read as '0' - n = Value at power-on reset

Bit 7 GIE (*Global Interrupt Enable bit*) Bit which enables or disables all interrupts.
 1 = all interrupts are enabled
 0 = all interrupts are disabled

Bit 6 EEIE (*EEPROM Write Complete Interrupt Enable bit*) Bit which enables an interrupt at the end of a writing routine to EEPROM
 1 = interrupt enabled

0 = interrupt disabled

If EEIE and EEIF (which is in EECON1 register) are set simultaneously, an interrupt will occur.

bit 5 TOIE (*TMR0 Overflow Interrupt Enable bit*) Bit which enables interrupts during counter TMR0 overflow.

1 = interrupt enabled

0 = interrupt disabled

If TOIE and TOIF are set simultaneously, interrupt will occur.

bit 4 INTE (*INT External Interrupt Enable bit*) Bit which enables external interrupt from pin RB0/INT.

1 = external interrupt enabled

0 = external interrupt disabled

If INTE and INTF are set simultaneously, an interrupt will occur.

bit 3 RBIE (*RB port change Interrupt Enable bit*) Enables interrupts to occur at the change of status of pins 4, 5, 6, and 7 of port B.

1 = enables interrupts at the change of status

0 = interrupts disabled at the change of status

If RBIE and RBIF are simultaneously set, an interrupt will occur.

bit 2 TOIF (*TMR0 Overflow Interrupt Flag bit*) Overflow of counter TMR0.

1 = counter changed its status from FFh to 00h

0 = overflow did not occur

Bit must be cleared in program in order for an interrupt to be detected.

bit 1 INTF (*INT External Interrupt Flag bit*) External interrupt occurred.

1 = interrupt occurred

0 = interrupt did not occur

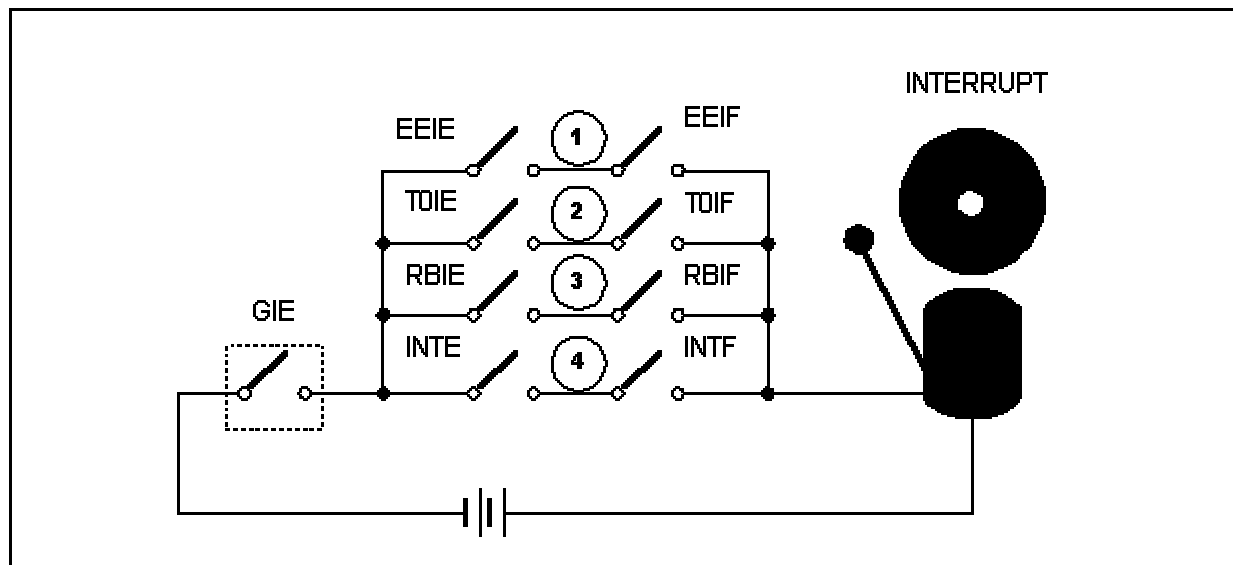
If a rising or falling edge was detected on pin RB0/INT, (which is defined with bit INTEDG in OPTION register), bit INTF is set.

bit 0 RBIF (*RB Port Change Interrupt Flag bit*) Bit which informs about changes on pins 4, 5, 6 and 7 of port B.

1 = at least one pin has changed its status

0 = no change occurred on any of the pins

Bit has to be cleared in an interrupt subroutine to be able to detect further interrupts.



Simplified outline of PIC16F84 microcontroller interrupt

PIC16F84 has four interrupt sources:

1. Termination of writing data to EEPROM
2. TMR0 interrupt caused by timer overflow
3. Interrupt during alteration on RB4, RB5, RB6 and RB7 pins of port B.
4. External interrupt from RB0/INT pin of microcontroller

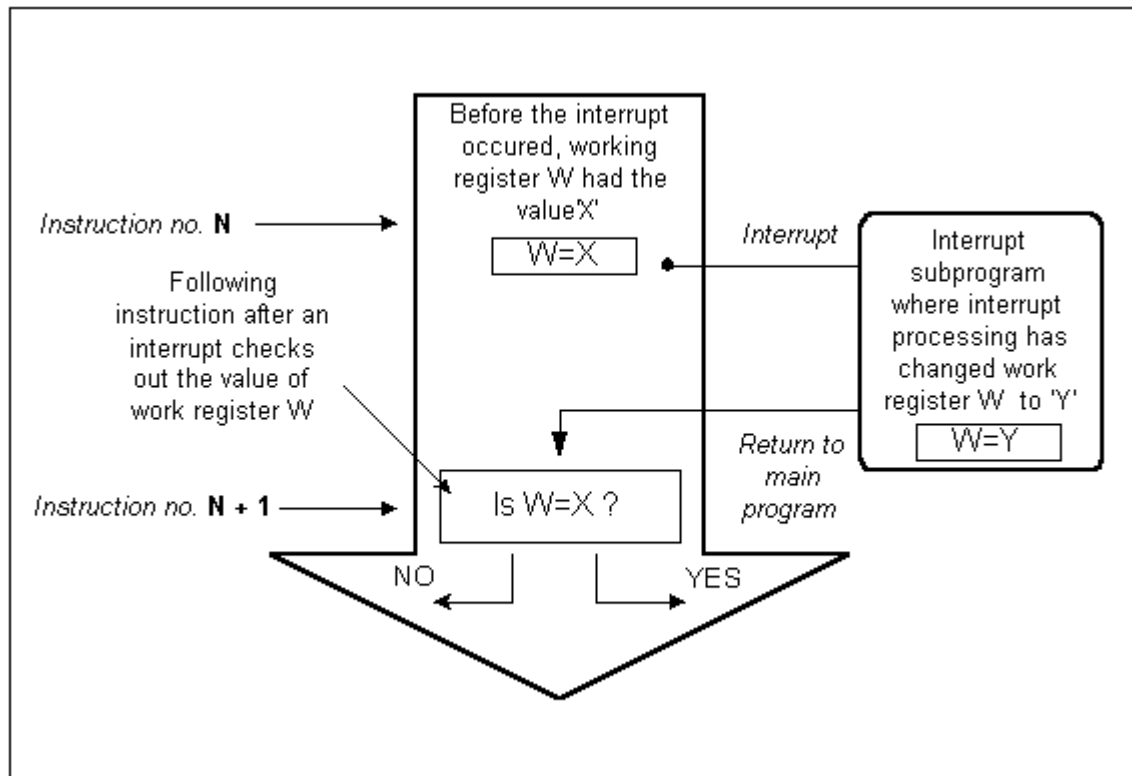
Generally speaking, each interrupt source has two bits joined to it. One enables interrupts, and the other detects when interrupts occur. There is one common bit called GIE which can be used to disallow or enable all interrupts simultaneously. This bit is very useful when writing a program because it allows for all interrupts to be disabled for a period of time, so that execution of some important part of a program would not be interrupted. When instruction which resets GIE bit was executed (GIE=0, all interrupts disallowed), any interrupt that remained unsolved should be ignored. Interrupts which remained unsolved and were ignored, are processed when GIE bit (GIE=1, all interrupts allowed) would be cleared. When interrupt was answered, GIE bit was cleared so that any additional interrupts would be disabled, return address was pushed onto stack and address 0004h was written in program counter - only after this does replying to an interrupt begin! After interrupt is processed, bit whose setting caused an interrupt must be cleared, or interrupt routine would automatically be processed over again during a return to the main program.

Keeping the contents of important registers

Only return value of program counter is stored on a stack during an interrupt (by return value of program counter we mean the address of the instruction which was to be executed, but wasn't because interrupt occurred). Keeping only the value of program counter is often not enough. Some registers which are already in use in the main program can also be in use in interrupt routine. If they were not retained, main program would during a return from an interrupt routine get completely different values in those registers, which would cause an error in the program. One example

for such a case is contents of the work register W. If we suppose that main program was using work register W for some of its operations, and if it had stored in it some value that's important for the following instruction, then an interrupt which occurs before that instruction would change the value of work register W which would directly be influenced the main program.

Procedure of recording important registers before going to an interrupt routine is called PUSH, while the procedure which brings recorded values back, is called POP. PUSH and POP are instructions with some other microcontrollers (Intel), but are so widely accepted that a whole operation is named after them. PIC16F84 does not have instructions like PUSH and POP, and they have to be programmed.



Common error: saving the value wasn't done before entering the interrupt routine

Due to simplicity and frequent usage, these parts of the program can be made as macros. The concept of a Macro is explained in "Program assembly language". In the following example, contents of W and STATUS registers are stored in W_TEMP and STATUS_TEMP variables prior to interrupt routine. At the beginning of PUSH routine we need to check presently selected bank because W_TEMP and STATUS_TEMP are found in bank 0. For exchange of data between these registers, SWAPF instruction is used instead of MOVF because it does not affect the STATUS register bits.

Example is an assembler program for following steps:

1. Testing the current bank
2. Storing W register regardless of the current bank

3. Storing STATUS register in bank 0.
4. Executing interrupt routine for interrupt processing (ISR)
5. Restores STATUS register
6. Restores W register

If there are some more variables or registers that need to be stored, then they need to be kept after storing STATUS register (step 3), and brought back before STATUS register is restored (step 5).

```

Push
    BTFSS STATUS, RP0          ; Bank 0
    GOTO RPOCLEAR              ; Yes
    BCF STATUS, RP0            ; NO, go to Bank 0
    MOVWF W_TEMP               ; Save W register
    SWAPF STATUS, W             ; W <- STATUS
    MOVWF STATUS_TEMP          ; STATUS_TEMP <- W
    BSF STATUS_TEMP, 1         ; RP0(STATUS_TEMP)=1
    GOTO ISR_Code              ; Push completed
RPOCLEAR
    MOVWF W_TEMP               ; Save W register
    SWAPF STATUS, W             ; W <- STATUS
    MOVWF STATUS_TEMP          ; STATUS_TEMP <- W
;
ISR_Code
    :
    : (Interrupt subprogram )
    :
;
Pop
    SWAPF STATUS_TEMP, W       ; W <- STATUS_TEMP
    MOVWF STATUS               ; STATUS <- W
    BTFSS STATUS, RP0          ; Bank 1?
    GOTO Return_WREG           ; NO,
    BCF STATUS, RP0            ; YES, go to Bank 0
    SWAPF W_TEMP, F            ; Return contents of W register
    SWAPF W_TEMP, W            ;
    BSF STATUS, RP0            ; Return to Bank 1
    RETFIE                     ; POP complete
Return_WREG
    SWAPF W_TEMP, F            ; Return contents of W register
    SWAPF W_TEMP, W            ;
    RETFIE                     ; POP completed

```

The same example can be carried out using macros, thus getting a more legible program. Macros that are already defined can be used for writing new macros. Macros BANK1 and BANK0 which are explained in "Memory organization" chapter are used with macros 'push' and 'pop'.


```

push    macro
movwf   W_Temp           ;W_Temp <- W
swapf   W_Temp,F         ;Swap them
BANK1   ;Macro for switching to Bank1
swapf   OPTION_REG,W     ;W <- OPTION_REG
movwf   Option_Temp      ;Option_Temp <- W
BANK0   ;macro for switching to Bank0
swapf   STATUS,W         ;W <- STATUS
movwf   Stat_Temp        ;Stat_Temp <-W
endm     ;End of push macro

pop     macro
swapf   Stat_Temp,W       ;W <- Stat_Temp
movwf   STATUS            ;STATUS <- W
BANK1   ;Macro for switching to Bank1
swapf   Option_Temp,W     ;W <- Option_Temp
movwf   OPTION_REG        ;OPTION_REG <- W
BANK0   ;Macro for switching to Bank0
swapf   W_Temp,W          ;W <- W_Temp
endm     ;End of a pop macro

```

External interrupt on RB0/INT pin of microcontroller

External interrupt on RB0/INT pin is triggered by rising signal edge (if bit INTEDG=1 in OPTION<6> register), or falling edge (if INTEDG=0). When correct signal appears on INT pin, INTF bit is set in INTCON register. INTF bit (INTCON<1>) must be cleared in interrupt routine, so that interrupt wouldn't occur again while going back to the main program. This is an important part of the program which programmer must not forget, or program will constantly go into interrupt routine. Interrupt can be turned off by resetting INTE control bit (INTCON<4>). Possible application of this interrupt could be measuring the impulse width or pause length, i.e. input signal frequency. Impulse duration can be measured by first enabling the interrupt on rising edge, and upon its appearing, starting the timer and then enabling the interrupt on falling edge. Timer should be stopped upon the appearing of falling edge - measured time period represents the impulse duration.

Interrupt during a TMR0 counter overflow

Overflow of TMR0 counter (from FFh to 00h) will set T0IF (INTCON<2>) bit. This is very important interrupt because many real problems can be solved using this interrupt. One of the examples is time measurement. If we know how much time counter needs in order to complete one cycle from 00h to FFh, then a number of interrupts multiplied by that amount of time will yield the total of elapsed time. In interrupt routine some variable would be incremented in RAM memory, value of that variable multiplied by the amount of time the counter needs to count through a whole cycle, would yield total elapsed time. Interrupt can be turned on/off by setting/resetting T0IE (INTCON<5>) bit.

Interrupt upon a change on pins 4, 5, 6 and 7 of port B

Change of input signal on PORTB <7:4> sets RBIF (INTCON<0>) bit. Four pins RB7, RB6, RB5 and RB4 of port B, can trigger an interrupt which occurs when status on

them changes from logic one to logic zero, or vice versa. For pins to be sensitive to this change, they must be defined as input. If any one of them is defined as output, interrupt will not be generated at the change of status. If they are defined as input, their current state is compared to the old value which was stored at the last reading from port B.

Interrupt upon finishing write-subroutine to EEPROM

This interrupt is of practical nature only. Since writing to one EEPROM location takes about 10ms (which is a long time in the notion of a microcontroller), it doesn't pay off to a microcontroller to wait for writing to end. Thus interrupt mechanism is added which allows the microcontroller to continue executing the main program, while writing in EEPROM is being done in the background. When writing is completed, interrupt informs the microcontroller that writing has ended. EEIF bit, through which this informing is done, is found in EECON1 register. Occurrence of an interrupt can be disabled by resetting the EEIE bit in INTCON register.

Interrupt initialization

In order to use an interrupt mechanism of a microcontroller, some preparatory tasks need to be performed. These procedures are in short called "initialization". By initialization we define to what interrupts the microcontroller will respond, and which ones it will ignore. If we do not set the bit that allows a certain interrupt, program will not execute an interrupt subprogram. Through this we can obtain control over interrupt occurrence, which is very useful.

```
clrf INTCON           ; all interrupts disabled
movlw B'00010000'     ; external interrupt only is enabled
bsf INTCON, GIE       ; occurrence of interrupts allowed
```

The above example shows initialization of external interrupt on RB0 pin of a microcontroller. Where we see one being set, that means that interrupt is enabled. Occurrence of other interrupts is not allowed, and interrupts are disabled altogether until GIE bit is set to one.

The following example shows a typical way of handling interrupts. PIC16F84 has got a single location for storing the address of an interrupt subroutine. This means that first we need to detect which interrupt is at hand (if more than one interrupt source is available), and then we can execute that part of a program which refers to that interrupt.

```

org ISR_ADDR          ;ISR_ADDR is interrupt routine address
btfsc INTCON, GIE      ;GIE bit turned off?
goto ISR_ADR          ;no, go back to the beginning
PUSH                  ;keep the contents of important registers
btfsc INTCON, RBIF     ;change on pins 4, 5, 6 and 7 of port B?
goto ISR_PORTB        ;jump to that section
btfsc INTCON, INTF     ;external interrupt occurred?
goto ISR_RBO          ;jump to that part
btfsc INTCON, TOIF     ;overflow of timer TMRO?
goto ISR_TMRO         ;jump to that section
BANK1                 ;Bank1 because of EECON1
Btfsc EECON1, EEIF     ;writing to EEPROM completed?
goto ISR_EEPROM       ;jump to that section
BANK0                 ;Bank0

ISR_PORTB
:                     ;section of code which is processed by an
                     ;interrupt ?
:
    goto END_ISR      ;jump to the exit of an interrupt
ISR_RBO
:                     ;section of code processing an interrupt?
:
    goto END_ISR      ;jump to exit of an interrupt.
ISR_TMRO
:                     ;section of code processing an interrupt
:
    goto END_ISR      ;jump to the exit of an interrupt
ISR_EEPROM
:                     ;section of code which processes an interrupt
:
    goto END_ISR      ;jump to an exit from an interrupt.
END_ISR
:
POP                  ;bringing back the contents of important
                     ;registers
    RETFIE            ;return and setting of GIE bit

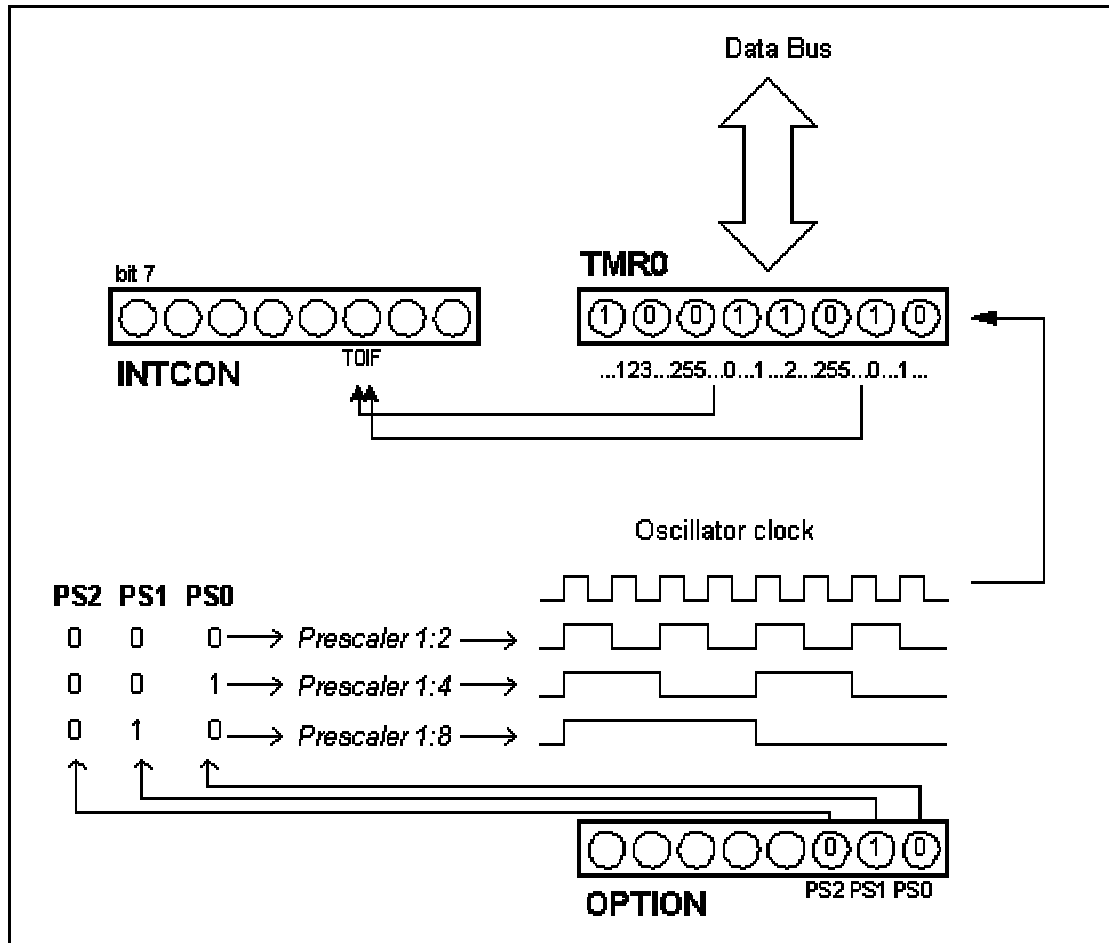
```



Return from interrupt routine can be accomplished with instructions RETURN, RETLW and RETFIE. It is recommended that instruction RETFIE be used because that instruction is the only one which automatically sets the GIE bit which allows new interrupts to occur.

2.7 Free-run timer TMR0

Timers are usually the most complicated parts of a microcontroller, so it is necessary to set aside more time for understanding them thoroughly. Through their application it is possible to establish relations between a real dimension such as "time" and a variable which represents status of a timer within a microcontroller. Physically, timer is a register whose value is continually increasing to 255, and then it starts all over again: 0, 1, 2, 3, 4...255...0, 1, 2, 3.....etc.

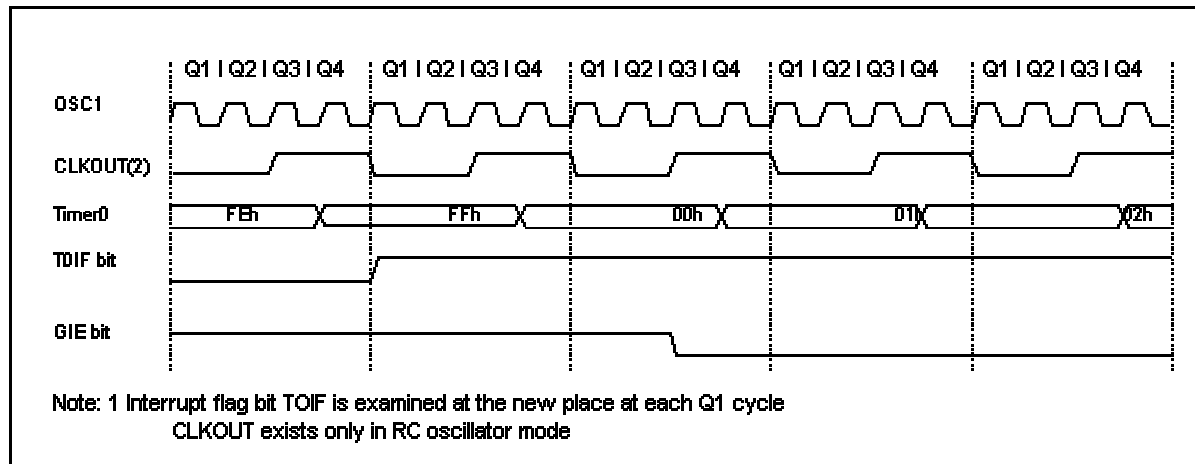


Relation between the timer TMR0 and prescaler

This incrementing is done in the background of everything a microcontroller does. It is up to programmer to think up a way how he will take advantage of this characteristic for his needs. One of the ways is increasing some variable on each timer overflow. If we know how much time a timer needs to make one complete round, then multiplying the value of a variable by that time will yield the total amount of elapsed time.

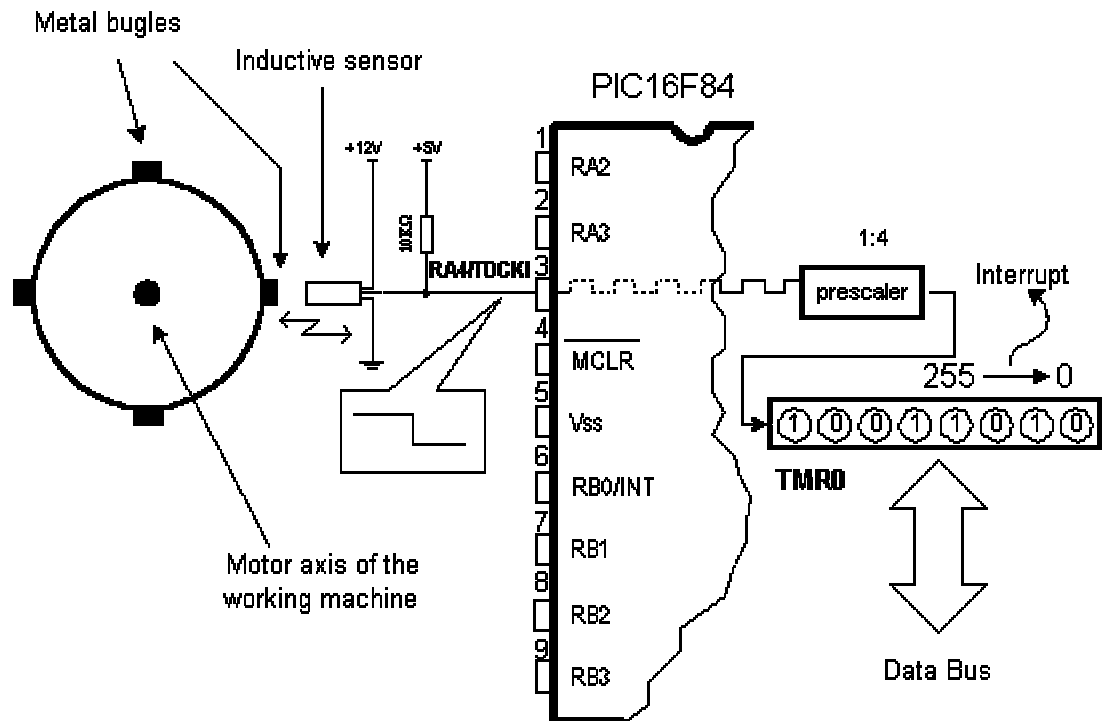
PIC16F84 has an 8-bit timer. Number of bits determines what value timer counts to before starting to count from zero again. In the case of an 8-bit timer, that number

is 256. A simplified scheme of relation between a timer and a prescaler is represented on the previous diagram. Prescaler is a name for the part of a microcontroller which divides oscillator clock before it will reach logic that increases timer status. Number which divides a clock is defined through first three bits in OPTION register. The highest divisor is 256. This actually means that only at every 256th clock, timer value would increase by one. This provides us with the ability to measure longer timer periods.



Time diagram of interrupt occurrence with TMR0 timer

After each count up to 255, timer resets its value to zero and starts with a new cycle of counting to 255. During each transition from 255 to zero, T0IF bit in INTCOM register is set. If interrupts are allowed to occur, this can be taken advantage of in generating interrupts and in processing interrupt routine. It is up to programmer to reset T0IF bit in interrupt routine, so that new interrupt, or new overflow could be detected. Beside the internal oscillator clock, timer status can also be increased by the external clock on RA4/TOCKI pin. Choosing one of these two options is done in OPTION register through T0CS bit. If this option of external clock was selected, it would be possible to define the edge of a signal (rising or falling), on which timer would increase its value.



Determining a number of full axis turns of the motor

In practice, one of the typical example that is solved via external clock and a timer is counting full turns of an axis of some production machine, like transformer winder for instance. Let's wind four metal screws on the axis of a winder. These four screws will represent metal convexity. Let's place now the inductive sensor at a distance of 5mm from the head of a screw. Inductive sensor will generate the falling signal every time the head of the screw is parallel with sensor head. Each signal will represent one fourth of a full turn, and the sum of all full turns will be found in TMR0 timer. Program can easily read this data from the timer through a data bus.

The following example illustrates how to initialize timer to signal falling edges from external clock source with a prescaler 1:4. Timer works in "polig" mode.

```

    clrf TMR0          ;TMR0=0
    clrf INTCON        ;Interrupts and TOIF=0 disallowed
    bsf STATUS,RPO     ;Bank1 because of OPTION_REG
    movlw B'00110001'  ;prescaler 1:4, falling edge selected external
                        ;clock source and pull up ;selected resistors
                        ;on port B activated
    movwf OPTION_REG   ;OPTION_REG <- W
TO_OVFL
    btfss INTCON, TOIF ;testing overflow bit
    goto TO_OVFL      ;interrupt has not occurred yet, wait
;
; (Part of the program which processes data regarding a number of turns)
;
goto TO_OVFL          ;waiting for new overflow

```

The same example can be carried out through an interrupt in the following way:

```

push    macro
        movwf W_Temp           ;W_Temp <- W
        swapf W_Temp,F         ;Swap them
        BANK1                  ;Macro for switching to Bank1
        swapf OPTION_REG,W     ;W <- OPTION_REG
        movwf Option_Temp      ;Option_Temp <- W
        BANK0                  ;macro for switching to Bank0
        swapf STATUS,W         ;W <- STATUS
        movwf Stat_Temp        ;Stat_Temp <-W
        endm                   ;End of push macro

pop     macro
        swapf Stat_Temp,W       ;W <- Stat_Temp
        movwf STATUS            ;STATUS <- W
        BANK1                  ;Macro for switching to Bank1
        swapf Option_Temp,W     ;W <- Option_Temp
        movwf OPTION_REG        ;OPTION_REG <- W
        BANK0                  ;Macro for switching to Bank0
        swapf W_Temp,W          ;W <- W_Temp
        endm                   ;End of a pop macro

```

Prescaler can be assigned either timer TMR0 or a watchdog. Watchdog is a mechanism which microcontroller uses to defend itself against programs getting stuck. As with any other electrical circuit, so with a microcontroller too can occur failure, or some work impairment. Unfortunately, microcontroller also has program where problems can occur as well. When this happens, microcontroller will stop working and will remain in that state until someone resets it. Because of this, watchdog mechanism has been introduced. After a certain period of time, watchdog resets the microcontroller (microcontroller in fact resets itself). Watchdog works on a simple principle: if timer overflow occurs, microcontroller is reset, and it starts executing a program all over again. In this way, reset will occur in case of both correct and incorrect functioning. Next step is preventing reset in case of correct functioning, which is done by writing zero in WDT register (instruction CLRWDT) every time it nears its overflow. Thus program will prevent a reset as long as it's executing correctly. Once it gets stuck, zero will not be written, overflow of WDT timer and a reset will occur which will bring the microcontroller back to correct functioning again.

Prescaler is accorded to timer TMR0, or to watchdog timer through PSA bit in OPTION register. By clearing PSA bit, prescaler will be accorded to timer TMR0. When prescaler is accorded to timer TMR0, all instructions of writing to TMR0 register (CLRF TMR0, MOVWF TMR0, BSF TMR0,...) will clear prescaler. When prescaler is assigned to a watchdog timer, only CLRWDT instruction will clear a prescaler and watchdog timer at the same time. Prescaler change is completely under programmer's control, and can be changed while program is running.



There is only one prescaler and one timer. Depending on the needs, they are assigned either to timer TMR0 or to a watchdog.

OPTION 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
RBP ^U	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0
Legend: R = Readable bit W = Writable bit U = Unimplemented bit, read as '0' -n = Value at POR reset							

bit 7 **RBP^U** (*PORTB Pull-up Enable bit*)

This bit turns internal pull-up resistors on port B on or off.

1 = 'pull-up' resistors turned on

0 = 'pull-up' resistors turned off

bit 6 **INTEDG** (*Interrupt Edge Select bit*)

If occurrence of interrupts was enabled, this bit would determine at what edge interrupt on RB0/INT pin would occur.

1 = rising edge

0 = falling edge

bit 5 **T0CS** (*TMR0 Clock Source Select bit*)

This pin enables a free-run timer to increment its value either from an internal oscillator, i.e. every 1/4 of oscillator clock, or via external impulses on RA4/T0CKI pin.

1 = external impulses

0 = 1/4 internal clock

bit 4 **T0SE** (*TMR0 Source Edge Select bit*)

If trigger TMR0 was enabled with impulses from a RA4/T0CKI pin, this bit would determine whether it would be on the rising or falling edge of a signal.

1 = falling edge

0 = rising edge

bit 3 **PSA** (*Prescaler Assignment bit*)

Bit which assigns prescaler between TMR0 and watchdog timer.

1 = prescaler is assigned to watchdog timer.

0 = prescaler is assigned to free timer TMR0

Bit 0:2 **PS0, PS1, PS2** (*Prescaler Rate Select bit*)

In case of 4MHz oscillator, one instruction cycle (4 internal clocks) lasts 1μs.

Numbers in the following table show the time period in μs between incrementing TMR or WDT.

Bits	TMRO	WDT
000	1:2	1:1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

2.8 EEPROM Data memory

PIC16F84 has 64 bytes of EEPROM memory locations on addresses from 00h to 63h that can be written to or read from. The most important characteristic of this memory is that it does not lose its contents with the loss of power supply. Data can be retained in EEPROM without power supply for up to 40 years (as manufacturer of PIC16F84 microcontroller states), and up to 1 million cycles of writing can be executed.

In practice, EEPROM memory is used for storing important data or process parameters.

One such parameter is a given temperature, assigned when setting up a temperature regulator to some process. If that data wasn't retained, it would be necessary to adjust a given temperature after each loss of supply. Since this is very impractical (and even dangerous), manufacturers of microcontrollers have begun installing one smaller type of EEPROM memory.

EEPROM memory is placed in a special memory space and can be accessed through special registers. These registers are:

- | | |
|---------------|---|
| EEDATA | Holds read data or that to be written. |
| EEADR | Contains an address of EEPROM location being accessed. |
| EECON1 | Contains control bits. |
| EECON2 | This register does not exist physically and serves to protect EEPROM from accidental writing. |

EECON1 register is a control register with five implemented bits. Bits 5, 6 and 7 are not used, and by reading always are zero. Interpretation of EECN1 register bits follows.

EECON1 Register

U-0	U-0	U-0	R/W-1	R/W-1	R/W-x	R/S-0	R/S-x
—	—	—	EEIF (1)	WRERR	WREN	WR	RD
bit 7			bit 0				

Legend:
R = Readable bit **W** = Writable bit
U = Unimplemented bit, read as '0' -n = Value at POR reset

bit 4 **EEIF** (*EEPROM Write Operation Interrupt Flag bit*) Bit used to inform that writing data to EEPROM has ended.
When writing has terminated, this bit would be set automatically. Programmer must clear EEIF bit in his program in order to detect new termination of writing.
1 = writing terminated
0 = writing not terminated yet, or has not started

bit 3 **WRERR** (*Write EEPROM Error Flag*) Error during writing to EEPROM
This bit was set only in cases when writing to EEPROM had been interrupted by a reset signal or by running out of time in watchdog timer (if activated).
1 = error occurred
0 = error did not occur

bit 2 **WREN** (*EEPROM Write Enable bit*) Enables writing to EEPROM
If this bit was not set, microcontroller would not allow writing to EEPROM.
1 = writing allowed
0 = writing disallowed

bit 1 **WR** (*Write Control bit*)
Setting of this bit initializes writing data from EEDATA register to the address specified through EEADR register.
1 = initializes writing
0 = does not initialize writing

bit 0 **RD** (*Read Control bit*)
Setting this bit initializes transfer of data from address defined in EEADR to EEDATA register. Since time is not as essential in reading data as in writing, data from EEDATA can already be used further in the next instruction.
1 = initializes reading
0 = does not initialize reading

Reading from EEPROM Memory

Setting the RD bit initializes transfer of data from address found in EEADR register to EEDATA register. As in reading data we don't need so much time as in writing, data taken over from EEDATA register can already be used further in the next instruction.

Sample of the part of a program which reads data in EEPROM, could look something like the following:

```
bcf    STATUS, RPO      ;bank0, because EEADR is at 09h
movlw  0x00             ;address of location being read
movwf  EEADR            ;address transferred to EEADR
bsf    STATUS, RPO      ;bank1 because EECON1 is at 88h
bsf    EECON1, RD       ;reading from EEPROM
bcf    STATUS, RPO      ;Bank0 because EEDATA is at 08h
movf   EEDATA, W        ;W <-- EEDATA
```

After the last program instruction, contents from an EEPROM address zero can be found in working register w.

Writing to EEPROM Memory

In order to write data to EEPROM location, programmer must first write address to EEADR register and data to EEDATA register. Only then is it useful to set WR bit which sets the whole action in motion. WR bit will be reset, and EEIF bit set following a writing what may be used in processing interrupts. Values 55h and AAh are the first and the second key whose disallow for accidental writing to EEPROM to occur. These two values are written to EECON2 which serves only that purpose, to receive these two values and thus prevent any accidental writing to EEPROM memory. Program lines marked as 1, 2, 3, and 4 must be executed in that order in even time intervals. Therefore, it is very important to turn off interrupts which could change the timing needed for executing instructions. After writing, interrupts can be enabled again .

Example of the part of a program which writes data 0xEE to first location in EEPROM memory could look something like the following:

```
        bcf    STATUS, RPO          ;bank0, because EEADR is at 09h
        movlw  0x00                 ;address of location being
                                   ;written to
        movwf  EEADR                ;address being transferred to
                                   ;EEADR
        movlw  0xEE                 ;write the value 0xEE
        movwf  EEDATA               ;data goes to EEDATA register
        bsf    STATUS, RPO          ;Bank1 because EEADR is at 09h
        bcf    INTCON, GIE          ;all interrupts are disabled
        bsf    EECON1, WREN         ;writing enabled
        movlw  55h
1)      movwf  EECON2               ;first key 55h --> EECON2
2)      movlw  AAh
3)      movwf  EECON2               ;second key AAh --> EECON2
4)      bsf    EECON1, WR           ;initializes writing
        bsf    INTCON, GIE          ;interrupts are enabled
```



It is recommended that WREN be turned off the whole time except when writing data to EEPROM that possibility of accidental writing would be minimal.

All writing to EEPROM will automatically clear a location prior to writing a new!

CHAPTER 3

Assembly Language Programming

[Introduction](#)

[3.1 Representing numbers in assembler](#)

[3.2 Assembly language elements](#)

[3.3 Writing a sample program](#)

[3.4 Control directives](#)

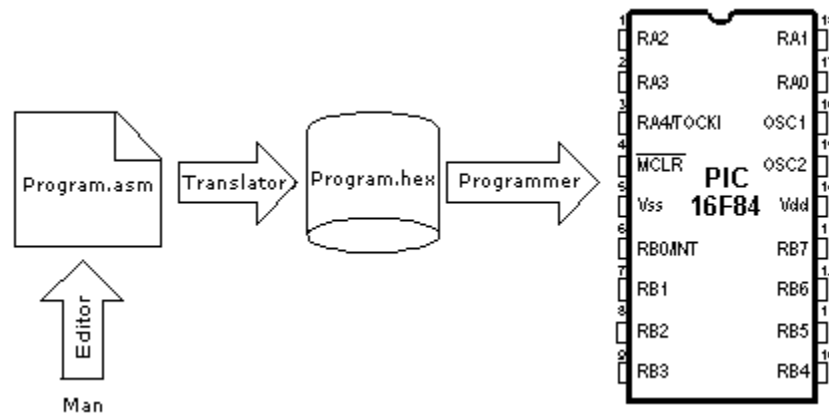
- [define](#)
- [include](#)
- [constant](#)
- [variable](#)
- [set](#)
- [equ](#)
- [org](#)
- [end](#)
- [if](#)
- [else](#)
- [endif](#)
- [while](#)
- [endw](#)
- [ifdef](#)
- [ifndef](#)
- [cblock](#)
- [endc](#)
- [db](#)
- [de](#)
- [dt](#)
- [CONFIG](#)
- [Processor](#)

[3.5 Files created as a result of program translation](#)

Introduction

The ability to communicate is of great importance in any field. However, it is only possible if both communication partners know the same language, i.e follow the same rules during communication. Using these principles as a starting point, we can also define communication that occurs between microcontrollers and man . Language that microcontroller and man use to communicate is called "assembly language". The

title itself has no deeper meaning, and is analogue to names of other languages , ex. English or French. More precisely, "assembly language" is just a passing solution. Programs written in assembly language must be translated into a "language of zeros and ones" in order for a microcontroller to understand it. "Assembly language" and "assembler" are two different notions. The first represents a set of rules used in writing a program for a microcontroller, and the other is a program on the personal computer which translates assembly language into a language of zeros and ones. A program that is translated into "zeros" and "ones" is also called "machine language".



The process of communication between a man and a microcontroller

Physically, "**Program**" represents a file on the computer disc (or in the memory if it is read in a microcontroller), and is written according to the rules of assembler or some other language for microcontroller programming. Man can understand assembler language as it consists of alphabet signs and words. When writing a program, certain rules must be followed in order to reach a desired effect. A **Translator** interprets each instruction written in assembly language as a series of zeros and ones which have a meaning for the internal logic of the microcontroller. Lets take for instance the instruction "RETURN" that a microcontroller uses to return from a sub-program.

When the assembler translates it, we get a 14-bit series of zeros and ones which the microcontroller knows how to interpret.

Example: RETURN 00 0000 0000 1000

Similar to the above instance, each assembler instruction is interpreted as corresponding to a series of zeros and ones.

The place where this translation of assembly language is found, is called an "execution" file. We will often meet the name "HEX" file. This name comes from a hexadecimal representation of that file, as well as from the suffix "hex" in the title, ex. "test.hex". Once it is generated, the execution file is read in a microcontroller through a programmer.

An **Assembly Language** program is written in a program for text processing (editor) and is capable of producing an ASCII file on the computer disc or in specialized surroundings such as MPLAB, which will be explained in the next chapter.

3.1 Representing numbers in assembler

In assembly language MPLAB, numbers can be represented in decimal, hexadecimal or binary form. We will illustrate this with a number 240:

.240	decimal
0xF0	hexadecimal
b'11110000'	binary

Decimal numbers start with a dot, hexadecimal with 0x, and binary start with b with the number itself under quotes '.

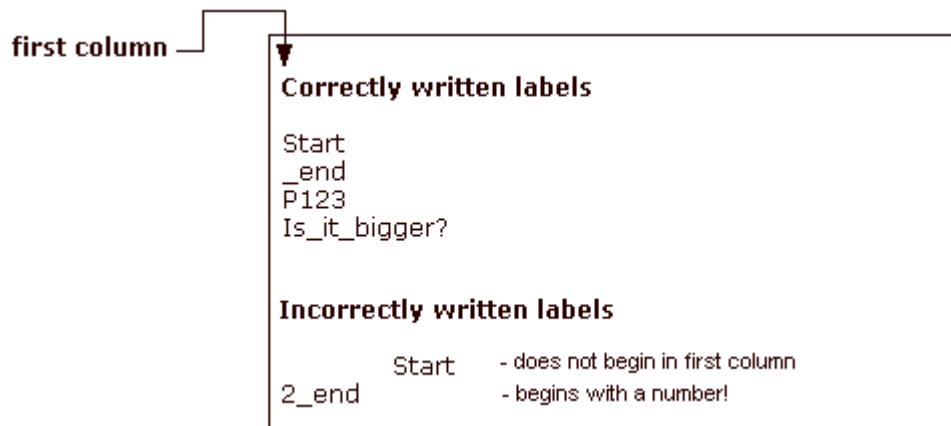
3.2 Assembly language elements

Basic elements of assembly language are:

- Labels
- Instructions
- Operands
- Directives
- Comments

Labels

A **Label** is a textual designation (generally an easy-to-read word) for a line in a program, or section of a program where the micro can jump to - or even the beginning of set of lines of a program. It can also be used to execute program branching (such as Goto) and the program can even have a condition that must be met for the Goto instruction to be executed. It is important for a label to start with a letter of the alphabet or with an underline "_". The length of the label can be up to 32 characters. It is also important that a label starts in the first column.



Instructions

Instructions are already defined by the use of a specific microcontroller, so it only remains for us to follow the instructions for their use in assembly language. The way we write an instruction is also called instruction "syntax". In the following example, we can recognize a mistake in writing because instructions `movlp` and `gotto` do not exist for the PIC16F84 microcontroller.

Correctly written instructions

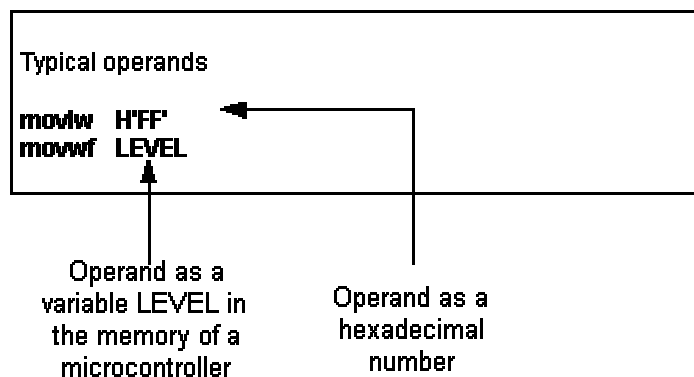
```
movlw    H'01FF'  
goto     Start
```

Incorrectly written instructions

```
movlp     H'01FF'  
gotto     Start
```

Operands

Operands are the instruction elements for the instruction is being executed. They are usually **registers** or **variables** or **constants**.



Comments

Comment is a series of words that a programmer writes to make the program more clear and legible. It is placed after an instruction, and must start with a semicolon ";".

Directives

A **directive** is similar to an instruction, but unlike an instruction it is independent on the microcontroller model, and represents a characteristic of the assembly language itself. Directives are usually given purposeful meanings via variables or registers. For example, `LEVEL` can be a designation for a variable in RAM memory at address `0Dh`. In this way, the variable at that address can be accessed via `LEVEL` designation. This is far easier for a programmer to understand than for him to try to remember address `0Dh` contains information about `LEVEL`.

Some frequently used directives:

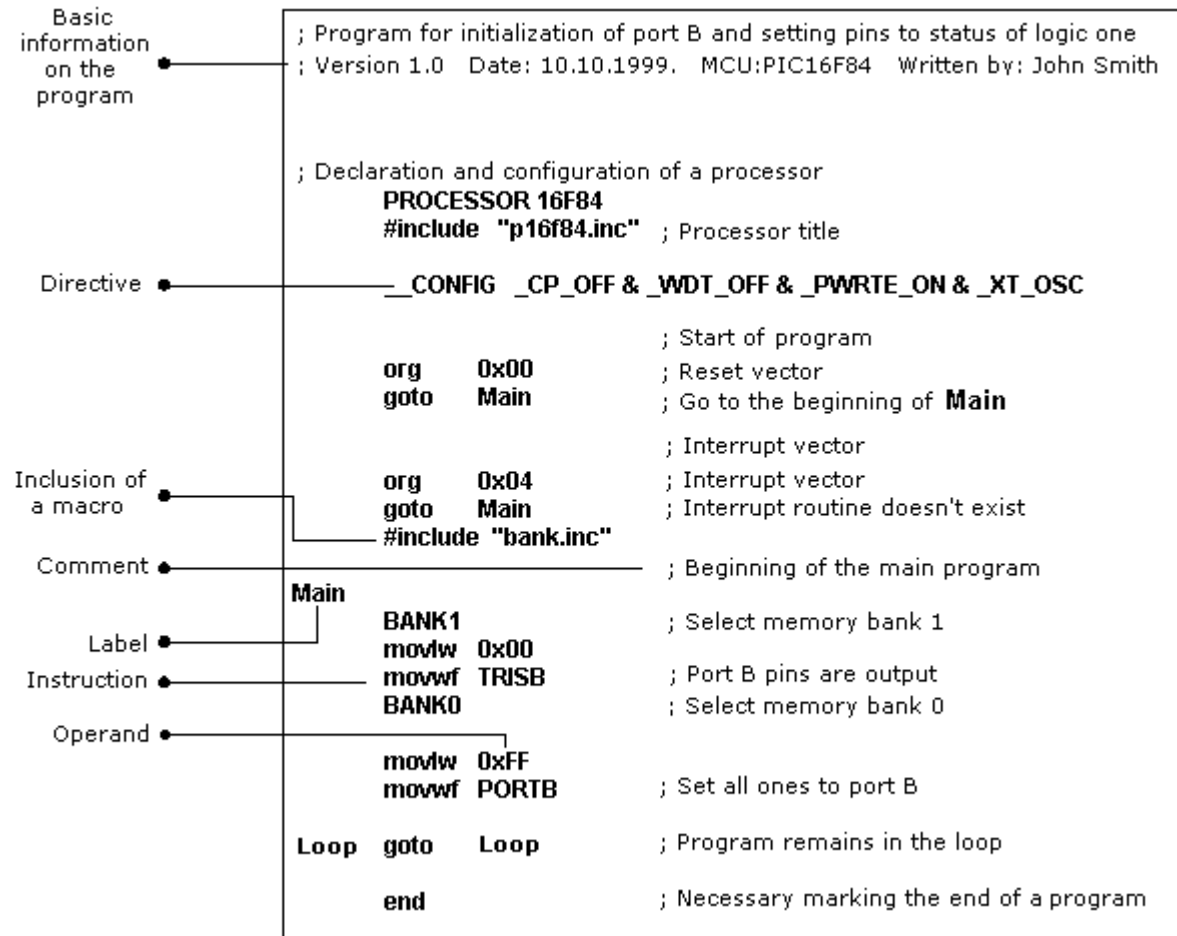
```
PROCESSOR 16F84
#include "p16f84.inc"

__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
```

3.3 Writing a sample program

The following example illustrates a simple program written in assembly language respecting the basic rules.

When writing a program, beside mandatory rules, there are also some rules that are not written down but need to be followed. One of them is to write the name of the program at the beginning, what the program does, its version, date when it was written, type of microcontroller it was written for, and the programmer's name.



Since this data isn't important for the assembly translator, it is written as **comments**. It should be noted that a comment always begins with a semicolon and it can be placed in a new row or it can follow an instruction. After the opening comment has been written, the **directive** must be included. This is

shown in the example above.

In order to function properly, we must define several microcontroller parameters such as:

- type of oscillator,
- whether watchdog timer is turned on, and
- whether internal reset circuit is enabled.

All this is defined by the following directive:

```
_CONFIG _CP_OFF&_WDT_OFF&PWRTE_ON&XT_OSC
```

When all the needed elements have been defined, we can start writing a program. First, it is necessary to determine an address from which the microcontroller starts, following a power supply start-up. This is (org 0x00).

The address from which the program starts if an interrupt occurs is (org 0x04).

Since this is a simple program, it will be enough to direct the microcontroller to the beginning of a program with a "**goto Main**" instruction.

The instructions found in the **Main** select memory bank1 (BANK1) in order to access TRISB register, so that port B can be declared as an output (movlw 0x00, movwf TRISB).

The next step is to select memory bank 0 and place status of logic one on port B (movlw 0xFF, movwf PORTB), and thus the main program is finished.

We need to make another loop where the micro will be held so it doesn't "wander" if an error occurs. For that purpose, one infinite loop is made where the micro is retained while power is connected. The necessary "end" at the end of each program informs the assembly translator that no more instructions are in the program.

3.4 Control directives

3.1 #DEFINE Exchanges one part of text for another

Syntax:

```
#define <text> [<another text>]
```

Description:

Each time <text> appears in the program , it will be exchanged for <another text >.

Example:

```
#define turned_on 1
#define turned_off 0
```

Similar directives: #UNDEFINE, #IFDEF, #IFNDEF

3.2 INCLUDE Include an additional file in a program

Syntax:

```
#include <file_name>
#include "file_name"
```

Description:

An application of this directive has the effect as though the entire file was copied to a place where the "include" directive was found. If the file name is in the square brackets, we are dealing with a system file, and if it is inside quotation marks, we are dealing with a user file. The directive "include" contributes to a better layout of the main program.

Example:

```
#include <regs.h>
#include "subprog.asm"
```

3.3 CONSTANT Gives a constant numeric value to the textual designation

Syntax:

Constant <name>=<value>

Description:

Each time that <name> appears in program, it will be replaced with <value>.

Example:

```
Constant MAXIMUM=100
Constant Length=30
```

Similar directives: SET, VARIABLE

3.4 VARIABLE Gives a variable numeric value to textual designation

Syntax:

Variable<name>=<value>

Description:

By using this directive, textual designation changes with particular value. It differs from CONSTANT directive in that after applying the directive, the value of textual designation can be changed.

Example:

```
variable level=20
variable time=13
```

Similar directives: SET, CONSTANT

3.5 SET Defining assembler variable

Syntax:

<name_variable>set<value>

Description:

To the variable <name_variable> is added expression <value>. SET directive is similar to EQU, but with SET directive name of the variable can be redefined following a definition.

Example:

```
level set 0
length set 12
level set 45
```

Similar directives: EQU, VARIABLE

3.6 EQU Defining assembler constant

Syntax:

<name_constant> equ <value>

Description:

To the name of a constant <name_constant> is added value <value>

Example:

```
five equ 5
six equ 6
seven equ 7
```

Similar instructions: SET

3.7 ORG Defines an address from which the program is stored in microcontroller memory

Syntax:

<label>org<value>

Description:

This is the most frequently used directive. With the help of this directive we define where some part of a program will be start in the program memory.

Example:

```
Start org    0x00
           movlw 0xFF
           movwf PORTB
```

The first two instructions following the first 'org' directive are stored from address 00, and the other two from address 10.

3.8 END End of program

Syntax:

end

Description:

At the end of each program it is necessary to place 'end' directive so that assembly

translator would know that there are no more instructions in the program.

Example:

```
.  
.  
movlw 0xFF  
movwf PORTB  
end
```

3.9 IF Conditional program branching

Syntax:

if<conditional_term>

Description:

If condition in <conditional_term> was met, part of the program which follows IF directive would be executed. And if it wasn't, then the part following ELSE or ENDIF directive would be executed.

Example:

```
if level=100  
goto FILL  
else  
goto DISCHARGE  
endif
```

Similar directives: #ELSE, ENDIF

3.10 ELSE The alternative to 'IF' program block with conditional terms

Syntax:

Else

Description:

Used with IF directive as an alternative if conditional term is incorrect.

Example:

```
If time< 50  
goto SPEED UP  
else goto SLOW DOWN  
endif
```

Similar instructions: ENDIF, IF

3.11 ENDIF End of conditional program section

Syntax:

endif

Description:

Directive is written at the end of a conditional block to inform the assembly translator that it is the end of the conditional block

Example:

```
If level=100
goto LOADS
else
goto UNLOADS
endif
```

Similar directives: ELSE, IF

3.12 WHILE Execution of program section as long as condition is met

Syntax:

```
while<condition>
.
endw
```

Description:

Program lines between WHILE and ENDW would be executed as long as condition was met. If a condition stopped being valid, program would continue executing instructions following ENDW line. Number of instructions between WHILE and ENDW can be 100 at the most, and number of executions 256.

Example:

```
While i<10
i=i+1
endw
```

3.13 ENDW End of conditional part of the program

Syntax:

```
endw
```

Description:

Instruction is written at the end of the conditional WHILE block, so that assembly translator would know that it is the end of the conditional block

Example:

```
while i<10
i=i+1

endw
```

Similar directives: WHILE

3.14 IFDEF Execution of a part of the program if symbol

was defined

Syntax:

`ifdef<designation>`

Description:

If designation <designation> was previously defined (most commonly by #DEFINE instruction), instructions which follow would be executed until ELSE or ENDIF directives are not would be reached.

Example:

```
#define test
.
ifdef test ;how the test was defined
.....; instructions from these lines would execute
endif
```

Similar directives: #DEFINE, ELSE, ENDIF, IFNDEF, #UNDEFINE

3.15 IFNDEF Execution of a part of the program if symbol was defined

Syntax:

`ifndef<designation>`

Description:

If designation <designation> was not previously defined, or if its definition was erased with directive #UNDEFINE, instructions which follow would be executed until ELSE or ENDIF directives would be reached.

Example:

```
#define test
.....
#undef test
.....
ifndef test ;how the test was undefined
..... .; instructions from these lines would execute
endif
```

Similar directives: #DEFINE, ELSE, ENDIF, IFDEF, #UNDEFINE

3.16 CBLOCK Defining a block for the named constants

Syntax:

```
Cblock [<term>]
      <label>[:<increment>], <label>[:<increment>].....
endc
```

Description:

Directive is used to give values to named constants. Each following term receives a value greater by one than its precursor. If <increment> parameter is also given, then value given in <increment> parameter is added to the following constant.

Value of <term> parameter is the starting value. If it is not given, it is considered to be zero.

Example:

```
Cblock 0x02
```

```
First, second, third ;first=0x02, second=0x03, third=0x04  
endc
```

```
cblock 0x02
```

```
first : 4, second : 2, third ;first=0x06, second=0x08, third=0x09  
endc
```

Similar directives: ENDC

3.17 ENDC End of constant block definition

Syntax:

```
endc
```

Description:

Directive was used at the end of a definition of a block of constants so assembly translator could know that there are no more constants.

Similar directives: CBLOCK

3.18 DB Defining one byte data

Syntax:

```
[<label>]db <term> [, <term>,,,,,<term>]
```

Description:

Directive reserves a byte in program memory. When there are more terms which need to be assigned a byte each, they will be assigned one after another.

Example:

```
db 't', 0x0f, 'e', 's', 0x12
```

Similar instructions: DE, DT

3.19 DE Defining the EEPROM memory byte

Syntax:

```
[<term>] de <term> [, <term>,,,,, <term>]
```

Description:

Directive is used for defining EEPROM memory byte. Even though it was first intended only for EEPROM memory, it could be used for any other location in any memory.

Example:

```
org H'2100'  
de "Version 1.0" , 0
```

Similar instructions: DB, DT

3.20 DT Defining the data table

Syntax:

```
[<label>] dt <term> [, <term> ,....., <term>]
```

Description:

Directive generates RETLW series of instructions, one instruction per each term.

Example:

```
dt "Message", 0  
dt first, second, third
```

Similar directives: DB, DE

3.21 _CONFIG Setting the configurational bits

Syntax:

```
_ _config<term> or _ _config<address>,<term>
```

Description:

Oscillator, watchdog timer application and internal reset circuit are defined. Before using this directive, the processor must be defined using PROCESSOR directive.

Example:

```
_CONFIG _CP_OFF&_WDT_OFF&_PWRTE_ON&_XT_OSC
```

Similar directives: _IDLOCS, PROCESSOR

3.22 PROCESSOR Defining microcontroller model

Syntax:

```
Processor <microcontroller_type>
```

Description:

Instruction sets the type of microcontroller where programming is done.

Example:

```
processor 16F84
```

3.5 Files created as a result of program translation

As a result of the process of translating a program written in assembler language we get files like:

- Executing file (Program_Name.HEX)
- Program errors file (Program_Name.ERR)
- List file (Program_Name.LST)

The first file contains translated program which was read in microcontroller by programming. Its contents can not give any information to programmer, so it will not be considered any further.

The second file contains possible errors that were made in the process of writing, and which were noticed by assembly translator during translation process. Errors can be discovered in a "list" file as well. This file is more suitable though when program is big and viewing the 'list' file takes longer.

The third file is the most useful to programmer. Much information is contained in it, like information about positioning instructions and variables in memory, or error signalization.

Example of 'list' file for the program in this chapter follows. At the top of each page is stated information about the file name, date when it was translated, and page number. First column contains an address in program memory where a instruction from that row is placed. Second column contains a value of any variable defined by one of the directives : SET, EQU, VARIABLE, CONSTANT or CBLOCK. Third column is reserved for the form of a translated instruction which PIC is executing. The fourth column contains assembler instructions and programmer's comments. Possible errors will appear between rows following a line in which the error occurred.



```
LOC OBJECT CODE      LINE SOURCE TEXT
VALUE

00001 ;Program for initialization of port B and setting its pins
00002 ;to the state of logic one
00003 ;Version: 1.0 Date: 10.05.2000.      MCU: PIC16F84 Written
00004 ;by: Petar Petrovic
00005
00006 ;Declaration and configuration of the processor
00007 PROCESSOR 16F84
00008 #include "pl6f84.inc" ;Processor title
00001 LIST
00002 ;P16F84.INC Standard Header File, Version 2.00 Microchip
;Technology, Inc.
00136 LIST
00009
2007 3FF1 00010 __CONFIG __CP_OFF & __WDT_OFF & __PWRTE_ON & __XT_OSC
00011
000C 00012 CONSTANT BASE = 0x0c
00013
00014 ;Start of a program
0000 00015 org 0x00 ;Reset vector
0000 2805 00016 goto Main ;Go to the beginning of the main program
00017
00018 ;Interrupt vector
0004 00019 org 0x04 ;Interrupt vector
0004 2805 00020 goto Main ;Interrupt routine does not exist
00021
00022 ;Beginning of the main program
00023 #include "Bank.inc" ; File with macros
00001 ;*****
00002 ; Makros BANK0 and BANK1
00003 ;*****
00004
0000 0010 00005 W_Temp set BASE+4
0000 0011 00006 Stat_Temp set BASE+5
0000 0012 00007 Option_Temp set BASE+6
00008
00009
00010 BANK0 macro
00011 bcf STATUS,RPO ; Select memory bank 0
00012 endm
00013
00014 BANK1 macro
00015 bsf STATUS,RPO ; Select memory bank 1
00016 endm
00017
0005 00024 Main
00025 BANK1 ; Select memory bank 1
0005 1683 M bsf STATUS,RPO ; Select memory bank 1
0006 3000 00026 movlw 0x00
Message[302]: Register in operand not in bank 0. Ensure that bank bits are
correct.
0007 0086 00027 movwf TRISB ;Port B pins are output
00028
00029 BANK0 ;Select memory bank 0
0008 1283 M bcf STATUS,RPO ;Select memory bank 0
0009 30FF 00030 movlw 0xFF
000A 0086 00031 movwf PORTB ;Set all ones to port B
00032
000B 280B 00033 Loop goto Loop ; Program stays in the loop
00034
00035 END ;Necessary marking the end of a program
```

MEMORY USAGE MAP ('X' = Used, '-' = Unused)

0000 : X--XXXXXXXXX-----

At the end of the "list" file there is a table of symbols used in a program. Useful element of 'list' file is a graph of memory utilization. At the very end, there is an error statistic as well as the amount of remaining program

CHAPTER 4

MPLAB

[Introduction](#)

[4.1 Installing the MPLAB program package](#)

[4.2 Welcome to MPLAB](#)

[4.3 Designing a project](#)

[4.4 Designing new Assembler file](#)

[4.5 Writing a program](#)

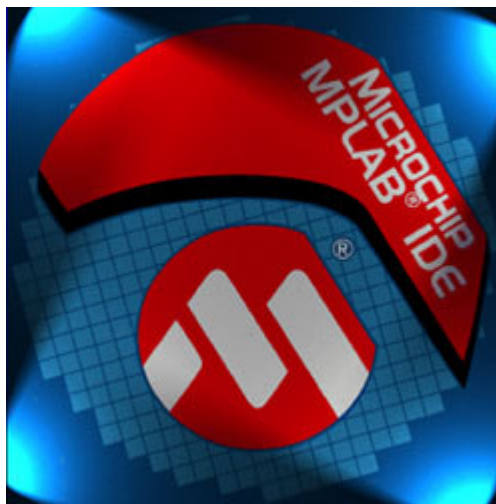
[4.6 Toolbar icons](#)

[4.7 MPSIM simulator](#)

Introduction

MPLAB is a Windows program package that makes writing and developing a program easier. It could best be described as developing environment for a standard program language that is intended for programming a PC. Some operations which were done from the instruction line with a large number of parameters until the discovery of IDE "Integrated Development Environment" are now made easier by using the MPLAB. Still, our tastes differ, so even today some programmers prefer the standard editors and compilers from instruction line. In any case, the written program is legible, and a well documented help is also available.

4.1 Installing the program -MPLAB



MPLAB consists of several parts:

- Grouping the projects files into one project (Project Manager)
- Generating and processing a program (Text Editor)
- Simulator of the written program used for simulating program function on the microcontroller.

Besides these, there are support systems for Microchip products such as PICStart Plus and ICD (In Circuit Debugger). As this book does not cover them , they will be mentioned as options only.

Minimal hardware requirements for starting the MPLAB are:

- PC compatible computer 486 or higher
- Microsoft Windows 3.1x or Windows 95 and new versions of the Windows operating system
- VGA graphic card
- 8MB memory (32MB recommended)
- 20MBs of free space on hard disk
- Mouse

In order to start the MPLAB we need to install it first. Installing is a process of copying MPLAB files from the CD onto a hard disc of your computer. There is an option on each new window which helps you return to a previous one, so errors should not present a problem or become a stressful experience. Installation itself works much the same as installation of most Windows programs. First you get the Welcome screen, then you can choose the options followed by installation itself, and, at the end, you get the message which says your installed program is ready to start.

Steps for installing MPLAB:

1. Start-up the Microsoft Windows
2. Place the Microchip CD into CD ROM drive
3. Click on START in the bottom left corner of the screen and choose the RUN option
4. Click on BROWSE and select CD ROM drive of your computer.
5. Find directory called MPLAB on your CD ROM
6. Click on MPLAB v6.31.EXE and then on OK .
7. Click again on OK in your RUN window

Installing begins after these seven steps. The following images explain specific installation stages. At the very beginning, small blue window will appear to notify you that the installation has begun.



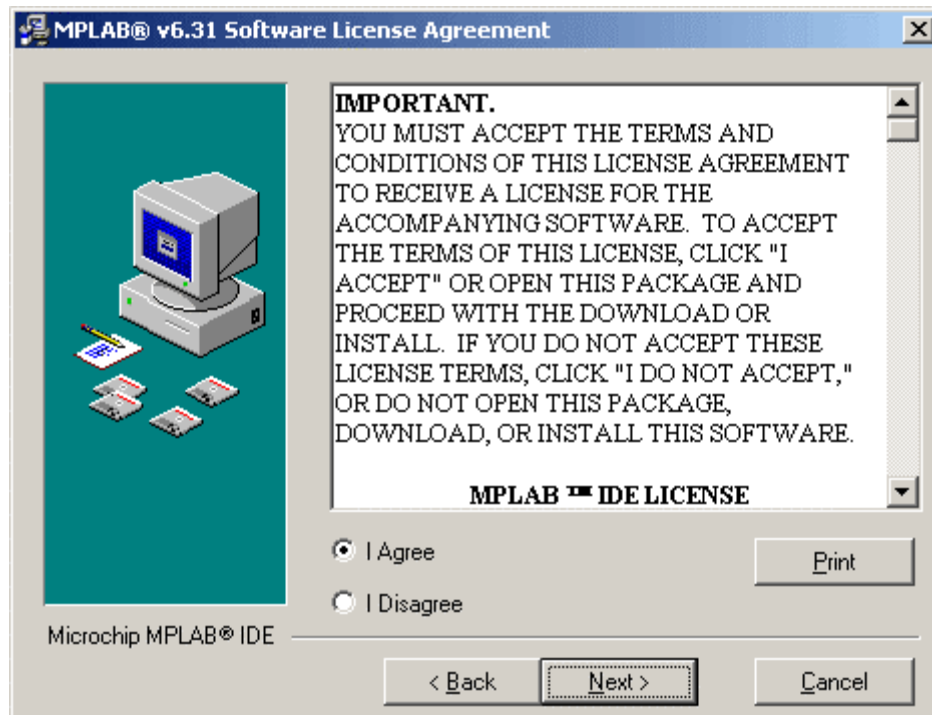
Initializing Wise Installation Wizard

The following window will warn you to close all the running applications, or preferably run this installation program from a re-boot.



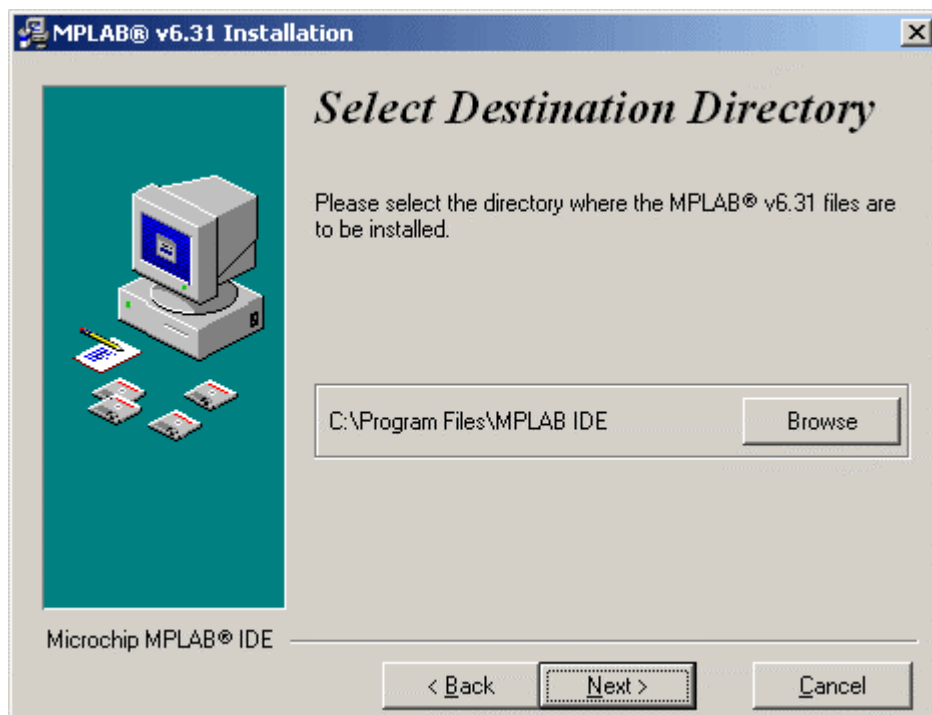
Welcome screen and an advice to re-boot prior to installation

Next is Software License Agreement window. In order to proceed with the installation, read the conditions, select the option "I Agree" and click on NEXT.



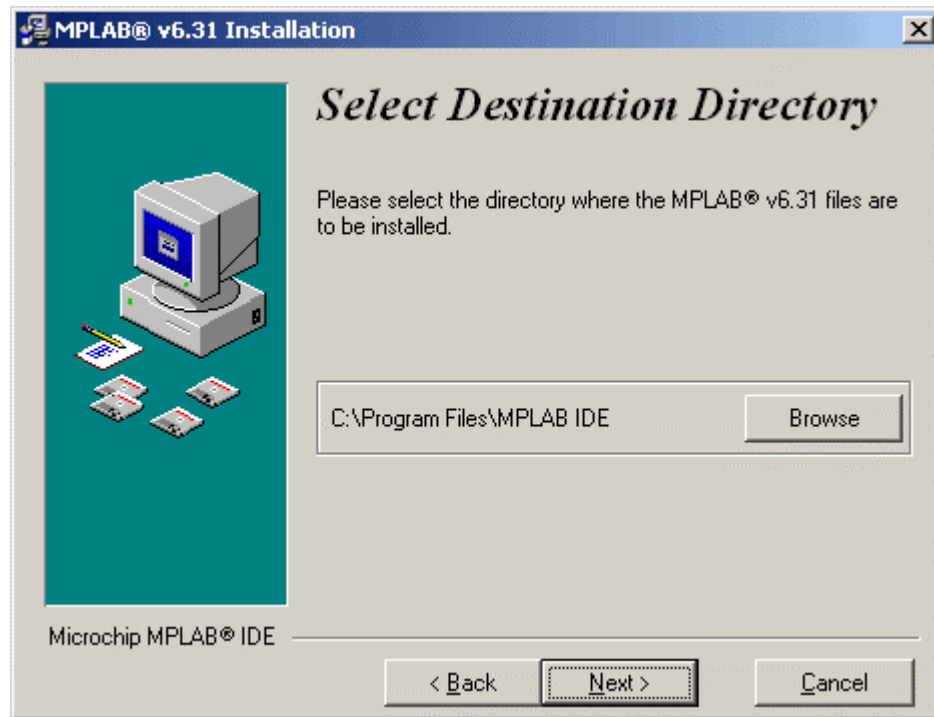
Software License Agreement window

The following window concerns the installation folder. Unless there is a specific reason for changing the default destination, you should leave it be.



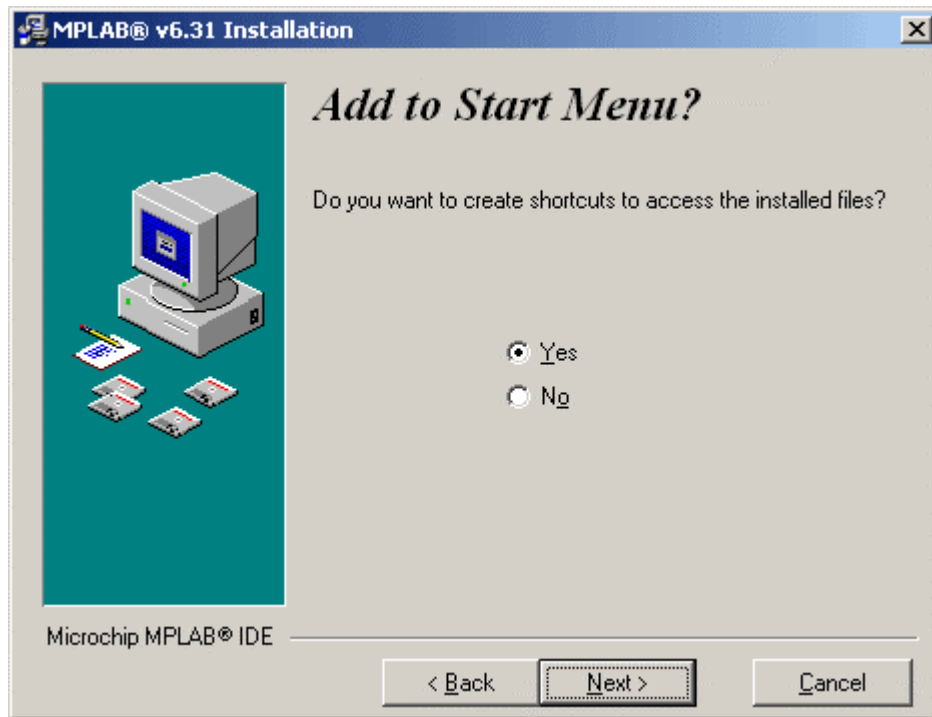
Determining the destination directory for MPLAB

Next option is meant for users who owned previous version(s) of MPLAB. Purpose of the option is to backup all files that will be replaced during the installation of the latest version. Assuming that this is our first installation of MPLAB, you should leave NO selected and click on NEXT.



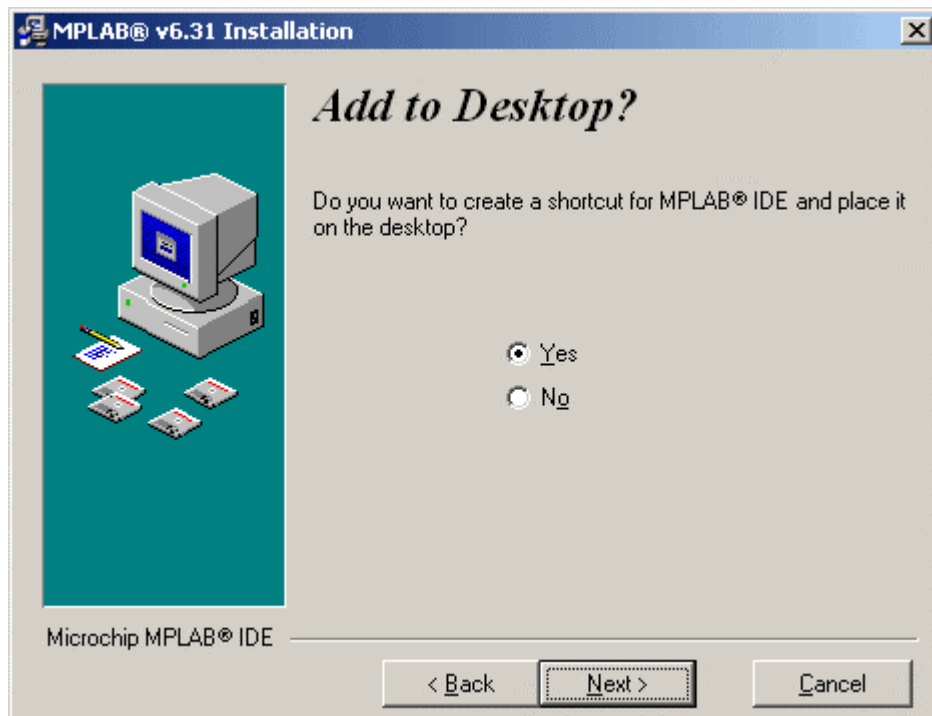
Option to backup old files from the previous version

Installation Wizard will ask you if you want to create shortcuts to access MPLAB from the start menu. Click on NEXT to proceed with installation.



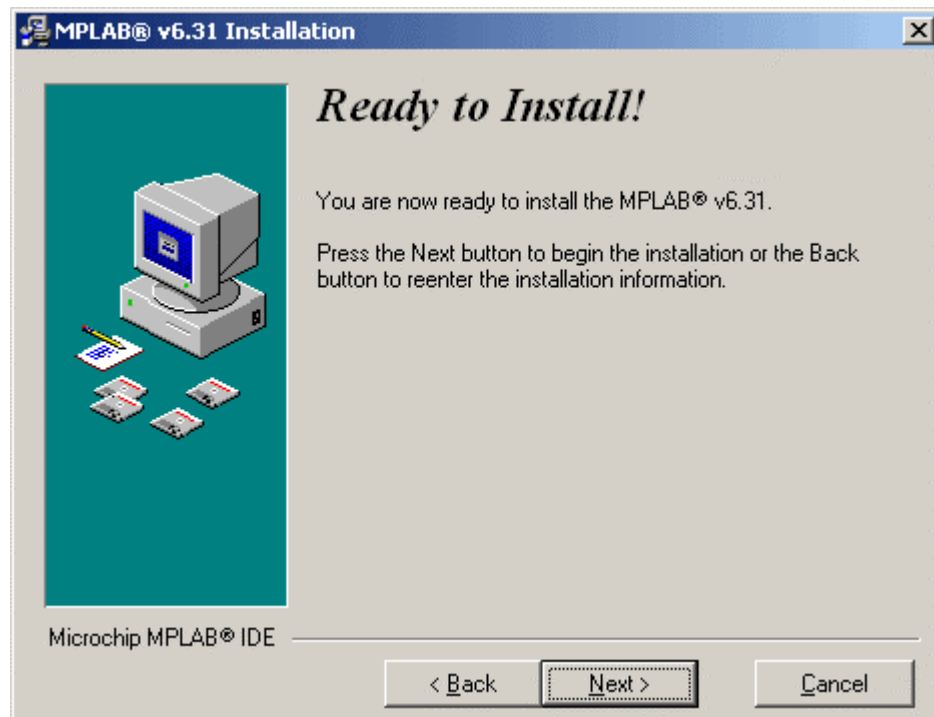
Adding MPLAB to the Start Menu

It is certainly useful to have MPLAB icon on desktop if you will be using the program frequently. We suggest clicking on Yes. Then, click on NEXT to proceed with installation.



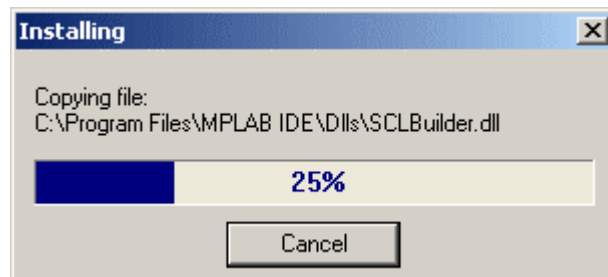
Adding MPLAB to the Start Menu

Ready to install! Click on NEXT to start copying the necessary files to your PC.



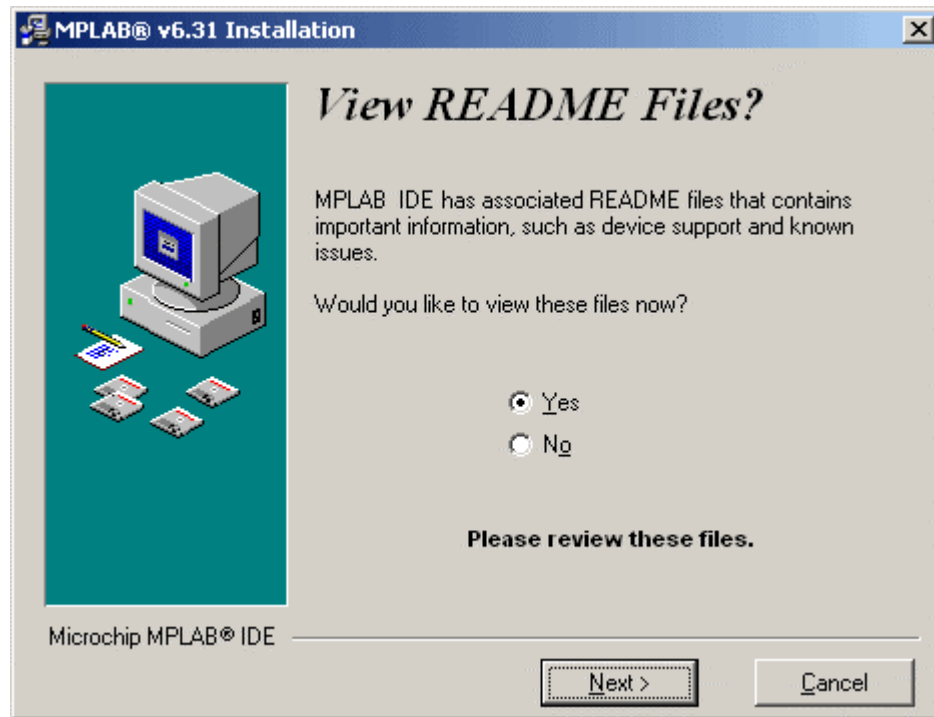
Screen prior to installation

Installation does not take long, and the installation flow can be monitored on a small window in the right corner of the screen.



Installation flow

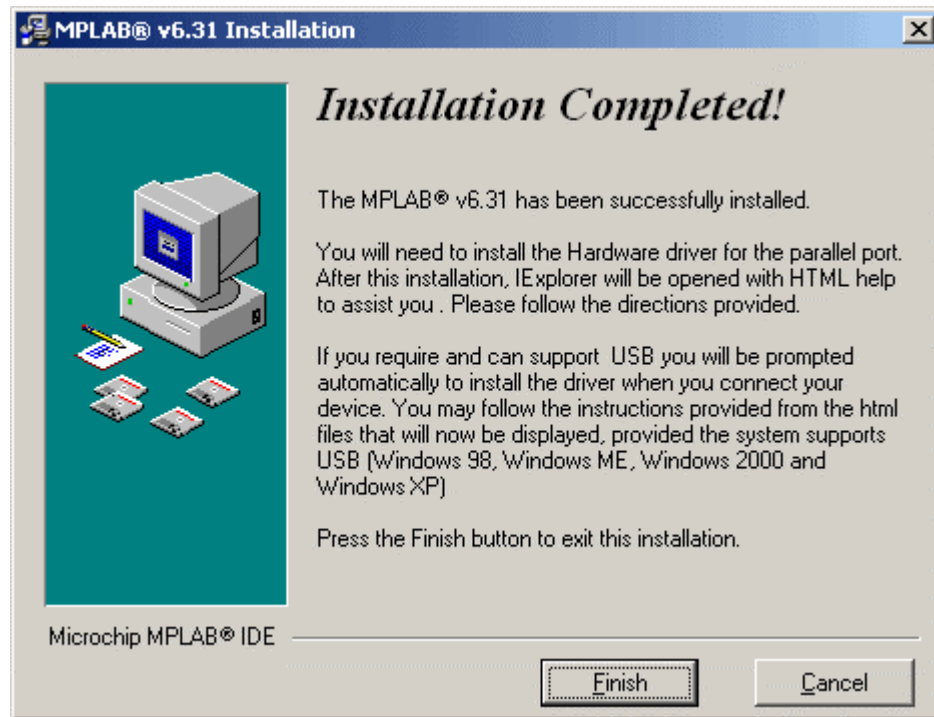
After installation have been completed, there are two dialog screens, one for the last minute information regarding program versions and corrections, and the other is the welcome screen. If text files (Readme.txt) have opened, they would need to be closed.



View README files?

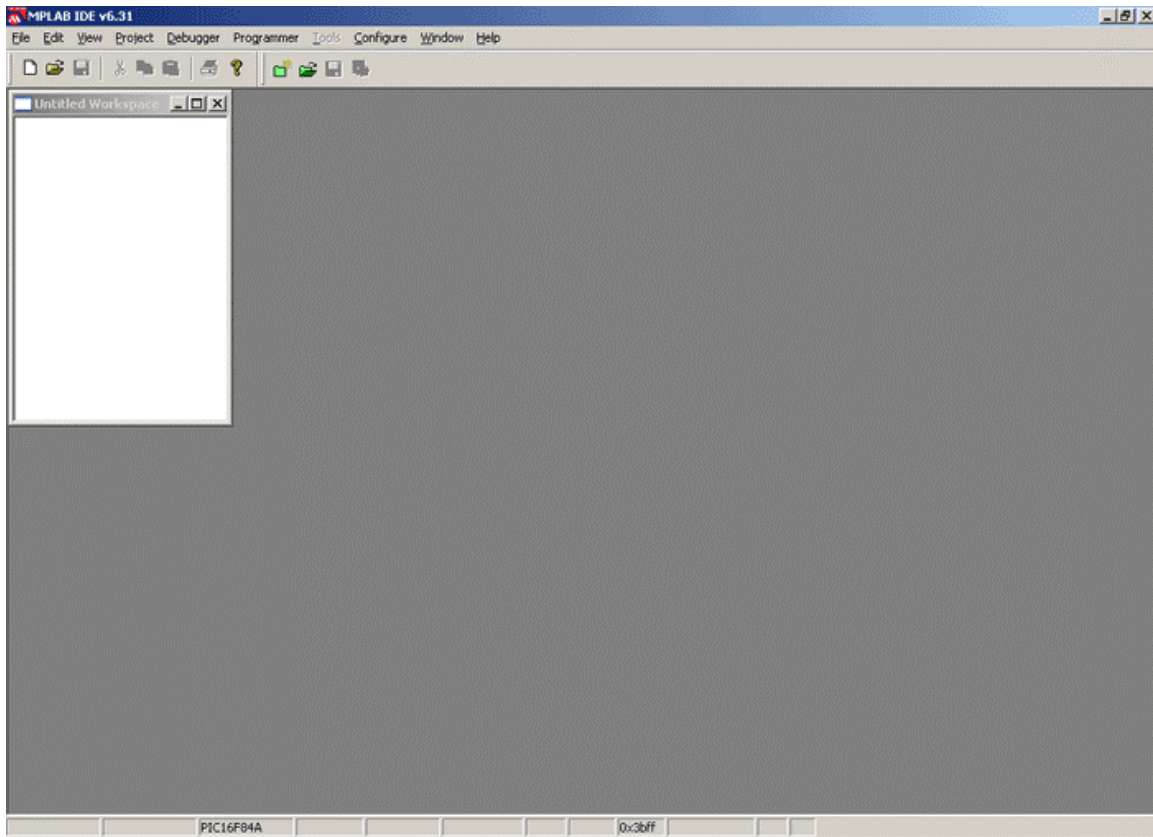
In order to install the USB device driver for MPLAB ICD2 you need to follow the instructions that will be displayed by executing Ddicd2XP.htm in the explorer window that has been opened during this installation. The installation is automatic when the MPLAB ICD2 is connected to the USB port. The instructions are for reference.

By clicking on Finish, installation of MPLAB is finished.



4.2 Program package MPLAB

Following the installation procedure, you can launch MPLAB by double-clicking the desktop icon. As you can see, MPLAB has the familiar look of Windows programs: a drop menu (uppermost line with standard options - File, Edit..etc.), toolbar (illustrated shortcuts for common actions) and a status line below the working area. There is a rule of thumb in Windows of taking the most frequently used program options and placing them below the menu, too. Thus, we can access them more quickly and speed up the work. In other words, what you have in the toolbar you also have in the menu.

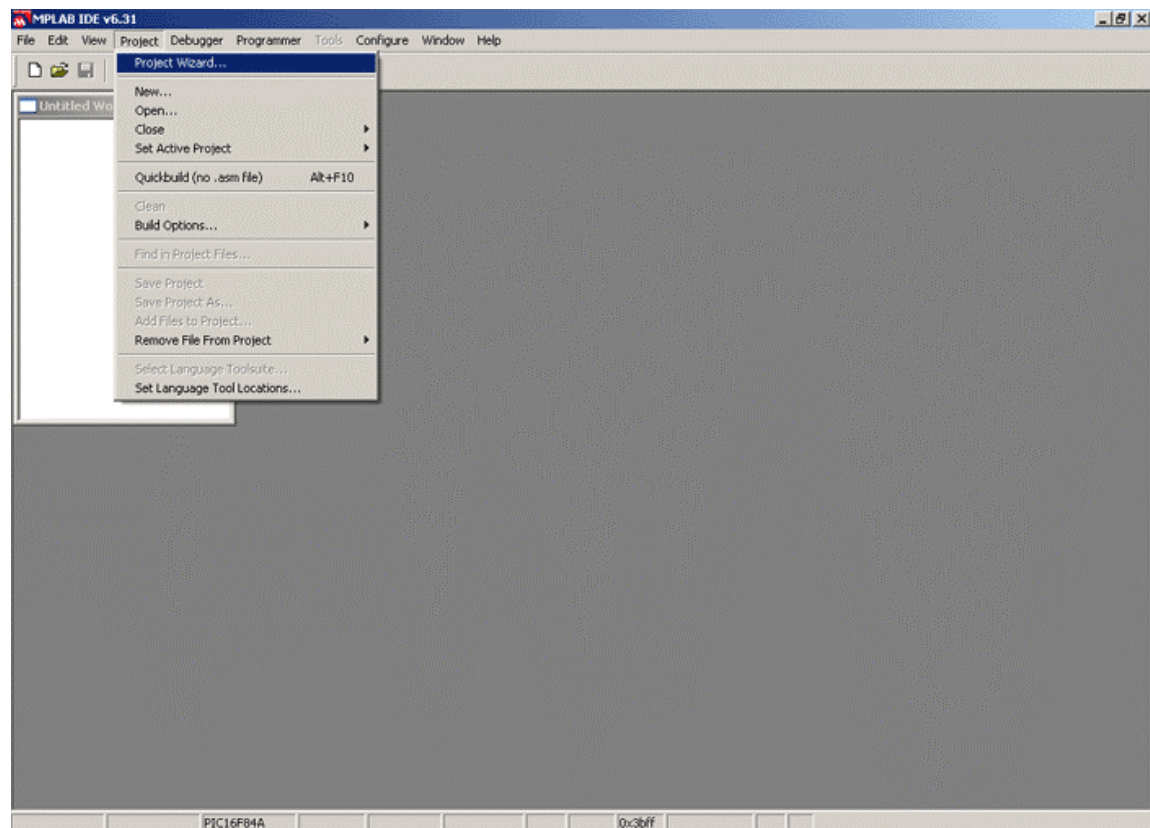


The screen after starting MPLAB

4.3 Designing a project

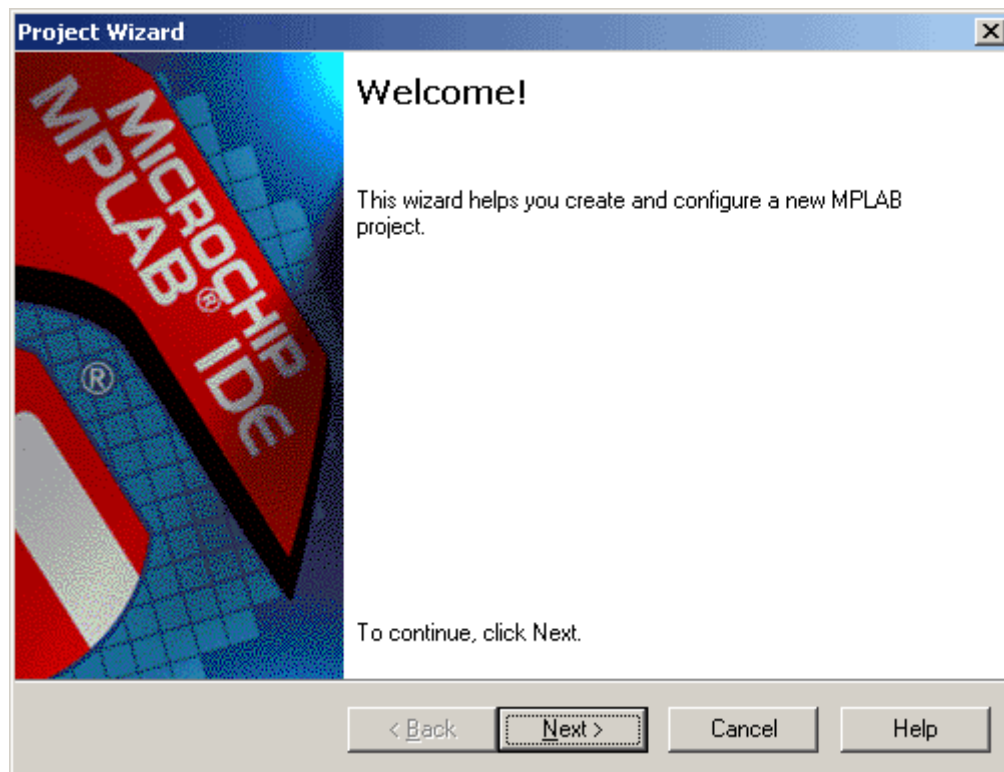
Preparing the program for loading into microcontroller can boil down to few basic steps:

1. Designing a project
2. Writing the program
3. Converting to zero-one code comprehensible by microcontroller, i.e. compiling.



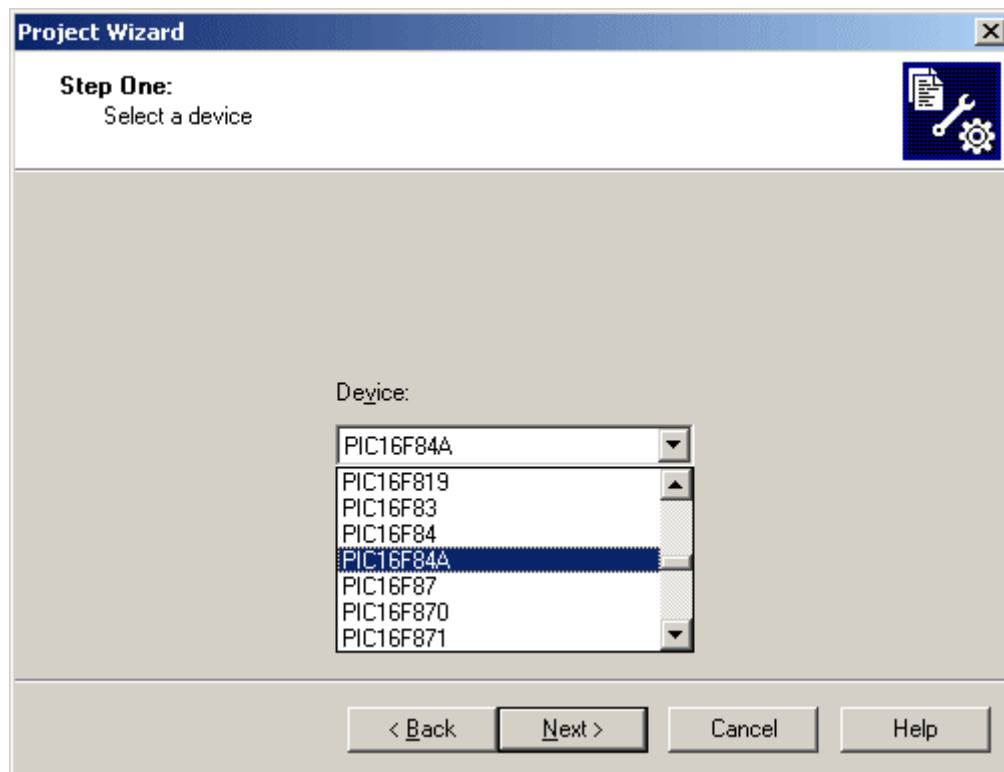
Opening a new project

To create a project, click on the option PROJECT and then click on PROJECT WIZARD, which will open the following window..



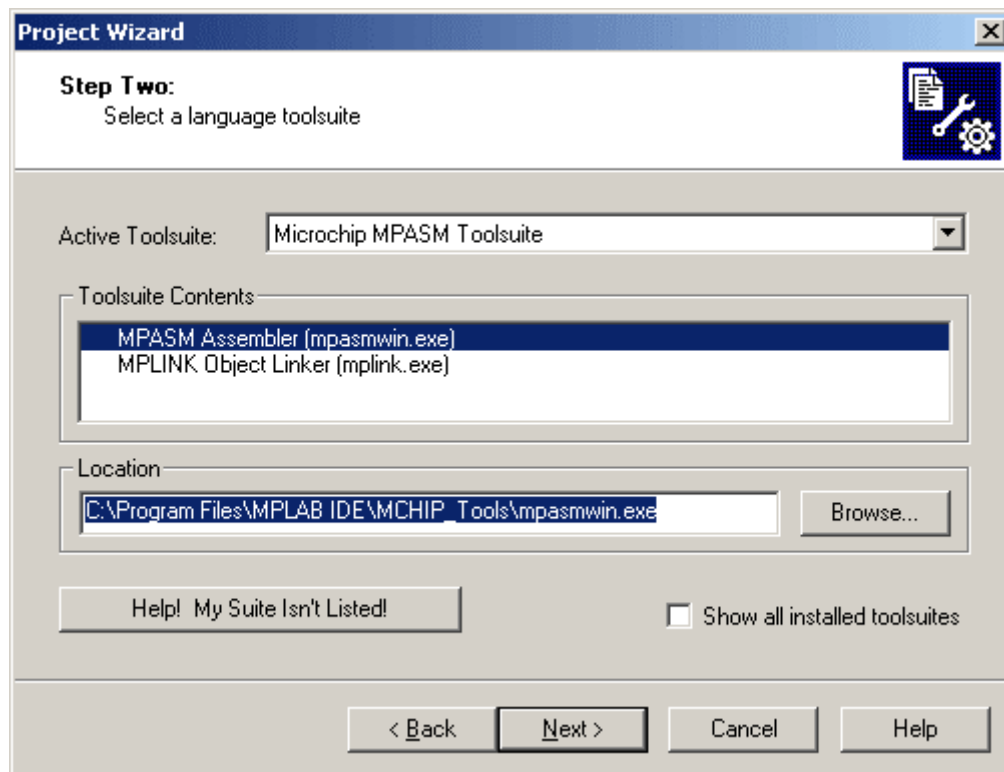
Creating a new project

Click on NEXT to continue. Next thing to do is to choose the appropriate microcontroller. In our case, it is PIC16F84A.



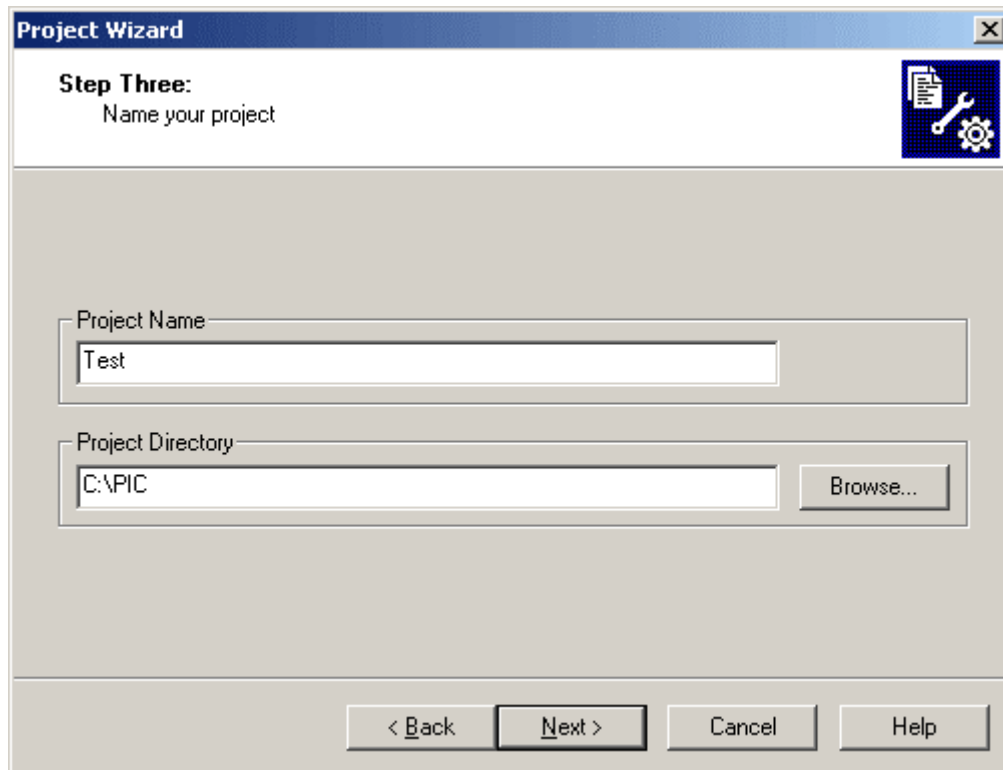
Choosing the appropriate microcontroller

Next step is defining the program language to be used. In our case it is Assembler, so we will select that option as shown on the image below.



Selecting a language toolsuite

All that is left is to name our project. The name should reflect the purpose and content of the program. Project can be stored in any folder according to your needs. It is a good thing to have that folder remind you of PIC microcontrollers; we named the folder simply PIC in the image below.



The image shows a Windows-style dialog box titled "Project Wizard". The title bar has a close button (X) on the right. The main area has a header section with the text "Step Three: Name your project" and a small icon of a document with a wrench and gear. Below this, there are two input fields. The first is labeled "Project Name" and contains the text "Test". The second is labeled "Project Directory" and contains the text "C:\PIC", with a "Browse..." button to its right. At the bottom of the dialog, there are four buttons: "< Back", "Next >", "Cancel", and "Help".

Project Wizard

Step Three:
Name your project

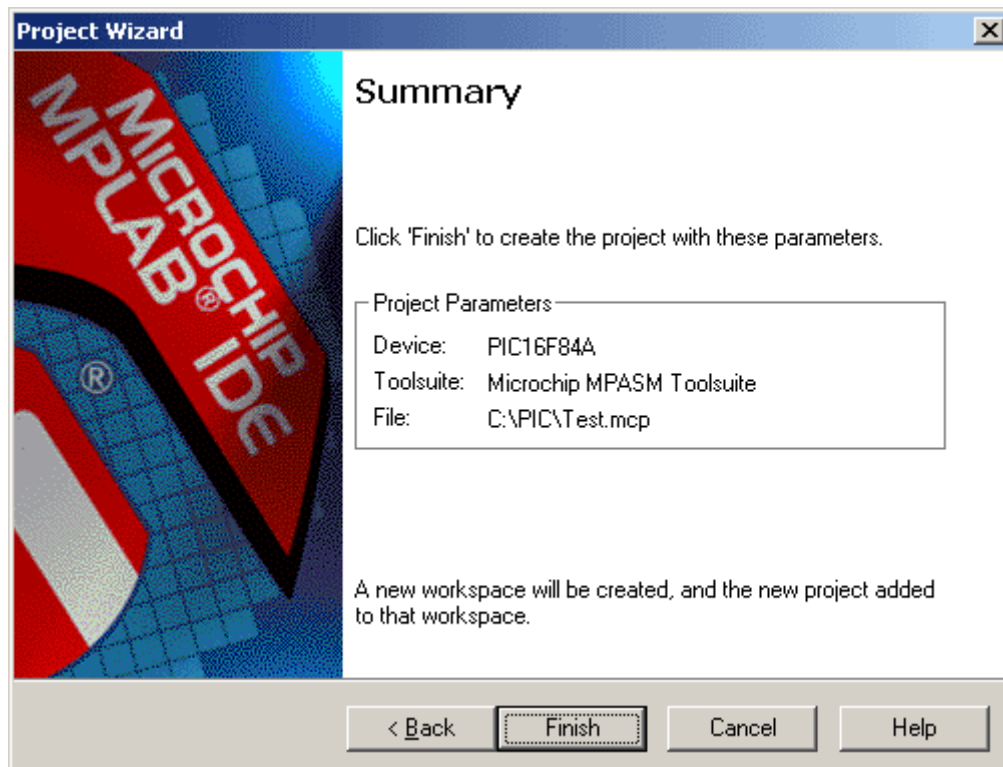
Project Name
Test

Project Directory
C:\PIC Browse...

< Back Next > Cancel Help

Naming the project

Upon naming the project, click on NEXT to open the summary window.



The image shows a Windows-style dialog box titled "Project Wizard". The title bar has a close button (X) on the right. The main area has a header section with the text "Summary". Below this, there is a paragraph of text: "Click 'Finish' to create the project with these parameters." Below that, there is a box labeled "Project Parameters" containing the following text: "Device: PIC16F84A", "Toolsuite: Microchip MPASM Toolsuite", and "File: C:\PIC\Test.mcp". At the bottom of the dialog, there are four buttons: "< Back", "Finish", "Cancel", and "Help". On the left side of the dialog, there is a large graphic with the text "MICROCHIP MPLAB IDE" and a registered trademark symbol.

Project Wizard

Summary

Click 'Finish' to create the project with these parameters.

Project Parameters

Device: PIC16F84A
Toolsuite: Microchip MPASM Toolsuite
File: C:\PIC\Test.mcp

A new workspace will be created, and the new project added to that workspace.

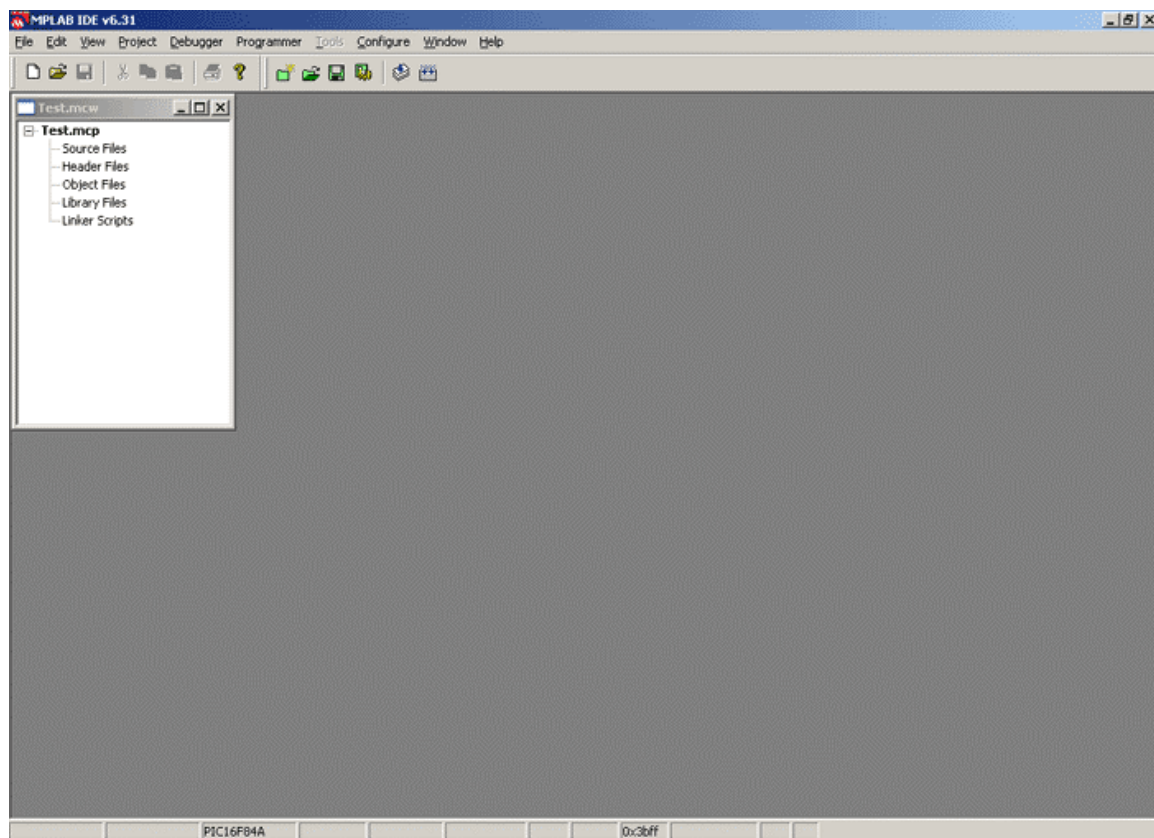
< Back Finish Cancel Help

Summary containing the defined parameters

Click on FINISH to create the project. The summary window contains the project parameters.

4.4 Creating a new assembler file

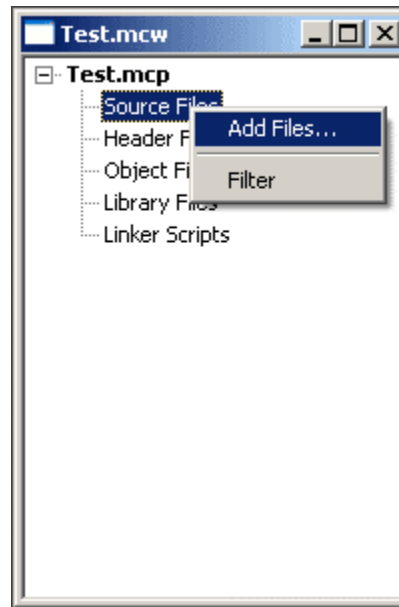
When 'project' part of the job is done, the following screen should appear.



New project opened

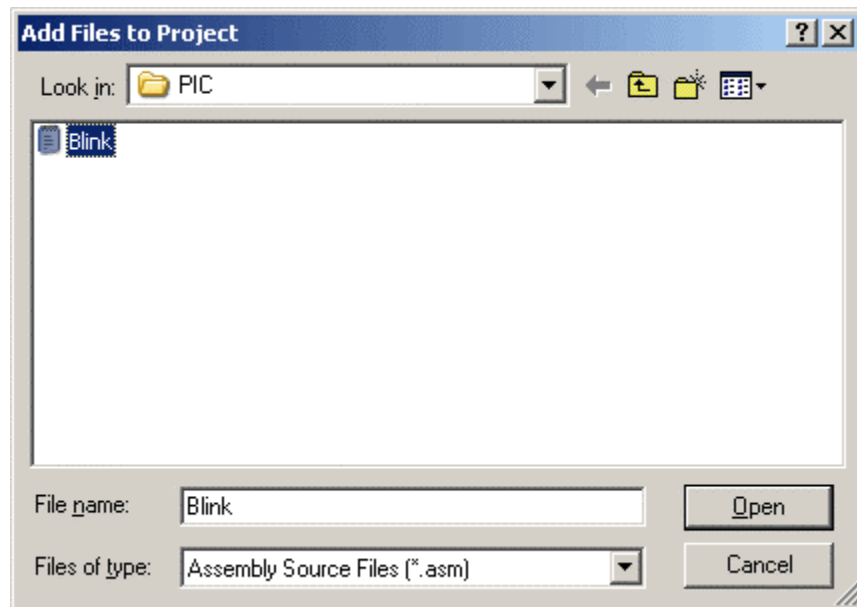
Next step is writing the program, which requires new file to be opened. Click on FILE > NEW, opening the text window within the MPLAB working area (new window represents the file program will be written to). Upon opening the new file, it should be saved to folder C:\PIC under a name "Blink.asm", to reflect the nature of the program (example for blinking diodes on microcontroller port B).

New file, "Blink.asm" should now be included into the project. Right-click on the source file in the window "Test.msw". This will open a small window with two available options - choose the first one, "Add Files".



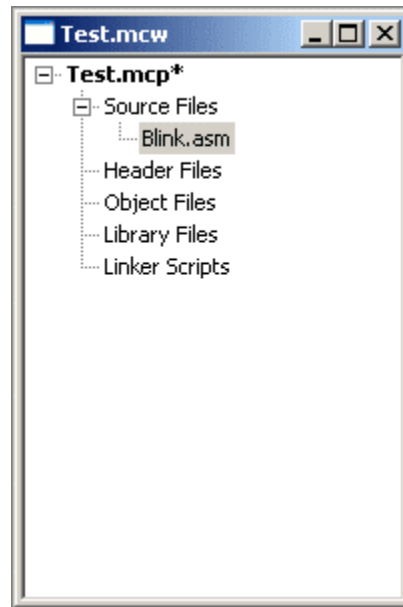
Inserting new Assembler file into project

In a newly opened browse window find our PIC folder and select the file "Blink.asm", as shown on the image below.



Selecting the desired file

The looks of the project window after the file blink.asm has been included is shown in the image below.



4.5 Writing a program

Only after all of the preceding operations have been completed we are able to start writing a program. This sample program is fairly simple and serves to illustrate creation of a project.



```
;Program for setting port B pins to logical one.
;Version: 1.0 Date: 25.04.2003 MCU:PIC16F84 Author:Petar Petrovic

;***** Deeclarations and microkontroller configuration *****

    PROCESSOR 16f84
    #include "pl6f84.inc"

    __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaration of variables *****

    Chlock    0x0C          ; Beginning of RAM
    endc              ; No variables

;***** Program memory structure *****

    ORG        0x00          ; Reset vector
    goto       Main          ; After reset jump to location

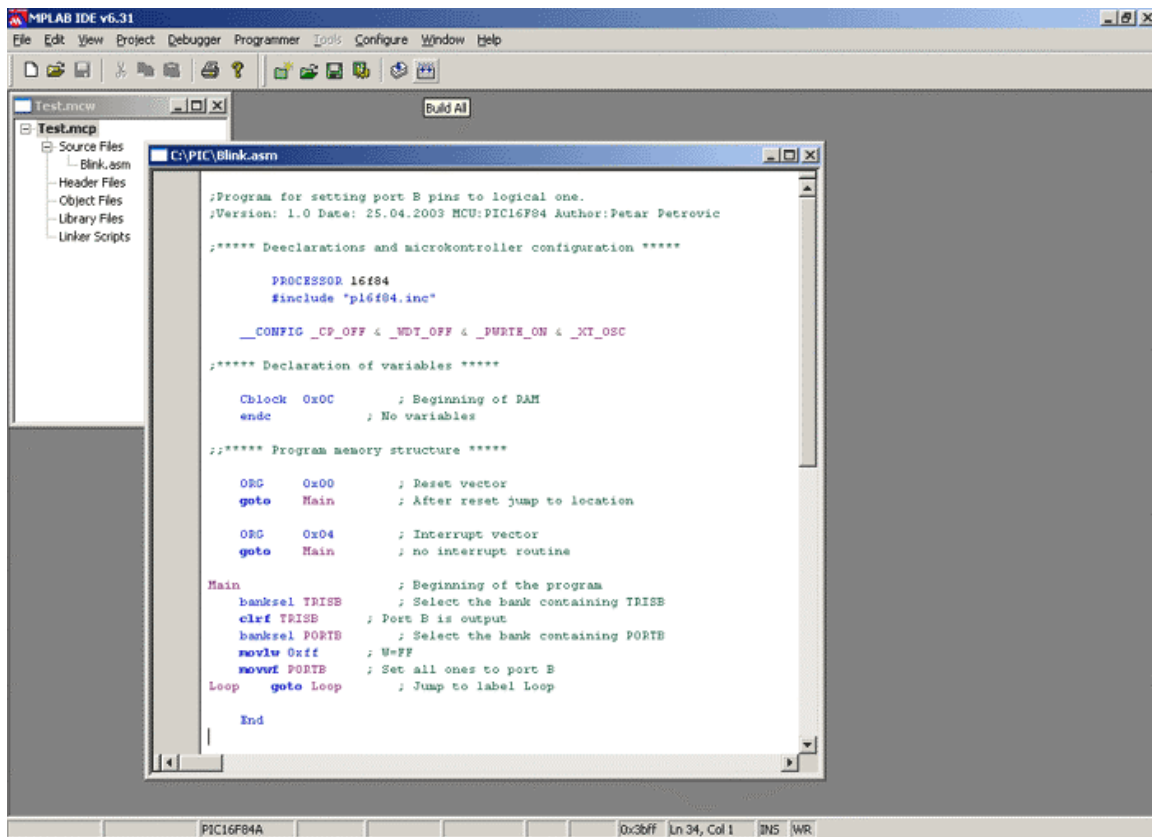
    ORG        0x04          ; Interrupt vector
    goto       Main          ; no interrupt routine

Main
    banksel    TRISB         ; Beginning of the program
    clrf       TRISB         ; Select the bank containing TRISB
    banksel    PORTB         ; Port B is output
    movlw      0xff          ; Select the bank containing PORTB
    movwf      PORTB         ; W=FF
    goto       Loop          ; Set all ones to port B

Loop
    goto       Loop          ; Jump to label Loop

End
```

You should re-write the program to the newly opened window or just copy/paste it from the disk.






Main window with blink.asm program












When the program is copied to "Blink.asm" window, we can use PROJECT -> BUILD ALL command to translate the program to executable HEX form. The last sentence in the window is the most important one, because it shows whether translation was successful or not. "BUILD SUCCEEDED" is a message stating that translation was successful and that there were no errors.

In case that error does show up, you need to double click on error message in 'Output' window. This will automatically take you to the assembler program, to the line where the error was encountered.

4.6 Toolbar icons

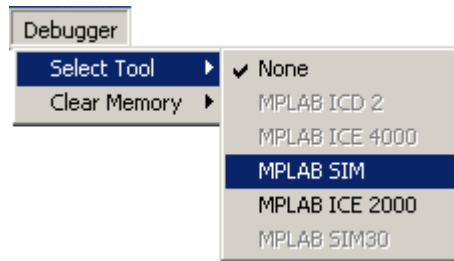
The following table contains detailed description of each toolbar icon.

	New Assembler file. It is commonly used for including an Assembler file into existing project.
	Open an existing Assembler file.
	Save program (Assembler file)

	Cut text. This and the following 3 icons are standard for all text processors. As every program code is actually a text file, these options are commonly used.
	Copy text. Unlike the previous one, this icon leaves the selected text on screen while storing it into clipboard.
	Paste text. When part of the text is copied or cut, it is stored to clipboard, where it can be called upon via this command.
	Print open file in which Assembler program is located.
	Help for the project section and the entire MPLAB.
	Create a new project. This command is straightforward as it avoids the use of Wizard - just name the project and type the path line to the appropriate folder.
	Open the existing project. Project opened in this way keeps all the previous settings.
	Save Workspace. Workspace saved in this way keeps all the parameter and window settings, so that when it's loaded you get the exact look of the work area it had at the moment of saving. Save your project frequently, especially if you work with multiple windows in simulator and you have them arranged to your liking.
	Build options.
	When eventual errors have been spotted during the simulation process, program needs to be repaired. As simulator uses HEX file as its input, program should be translated anew in order to take the changes to the simulator. This icon (Make) translates the project from the start and creates the latest version of the HEX file for the simulator.
	Similar to the previous icon, except that the whole project is translated and not just the Assembler file whose code has been changed.

4.7 MPSIM Simulator

Simulator is part of the MPLAB environment which provides a better insight into the workings of a microcontroller. Through a simulator, we can monitor current variable values, register values and status of port pins. Truthfully, simulator does not have the same value in all programs. If a program is simple (like the one given here as an example), simulation is not of great importance because setting port B pins to logic one is not a difficult task. However, simulator can be of great help with more complicated programs which include timers, different conditions where something happens and other similar requirements (especially with mathematical operations). Simulation, as the name indicates "simulates the work of a microcontroller". As microcontroller executes instructions one by one, simulator is conceived - programmer moves through a program step-by-step (line-by-line) and follows what goes on with data within the microcontroller. When writing is completed, it is a good trait for a programmer to first test his program in a simulator, and then run it in a real situation. Unfortunately, as with many other good habits, man tends to avoid this one too, more or less. Reasons for this are partly personality, and partly a lack of good simulators.

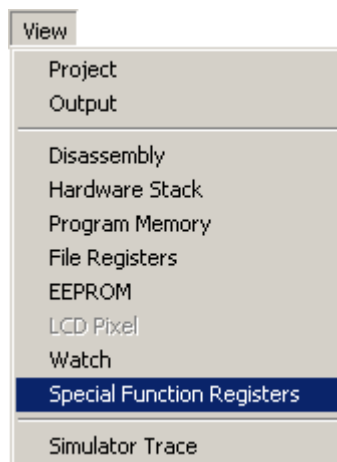


Starting the program simulation

Simulator is activated by clicking on DEBUGGER > SELECT TOOL > MPLAB SIM, as shown in the image above. Four new icons appear to the right. They are related to simulator only, and have the following meaning:

	Start the program execution at full speed. When started, simulator executes the program until "paused" by the icon below (just as with cassette or CD player).
	Stops full-speed program execution. After this icon has been clicked, program execution may be continued step-by-step or at full-speed.
	Step Into icon. Step-by-step program execution. Clicking on this icon executes the succeeding program line. It enters the macros and subroutines.
	Same as the previous icon, except it does not enter the macros and subroutines.
	Resets the microcontroller. Clicking on this icon positions the program counter to the beginning of program and simulation may begin.

First thing we need to do, as in a real situation, is to reset a microcontroller with DEBUGGER > RESET command or by clicking on the reset icon. This command results in green marker line positioned at the beginning of the program, and program counter PCL is positioned at zero which can also be seen in *Special Functions Registers* window.

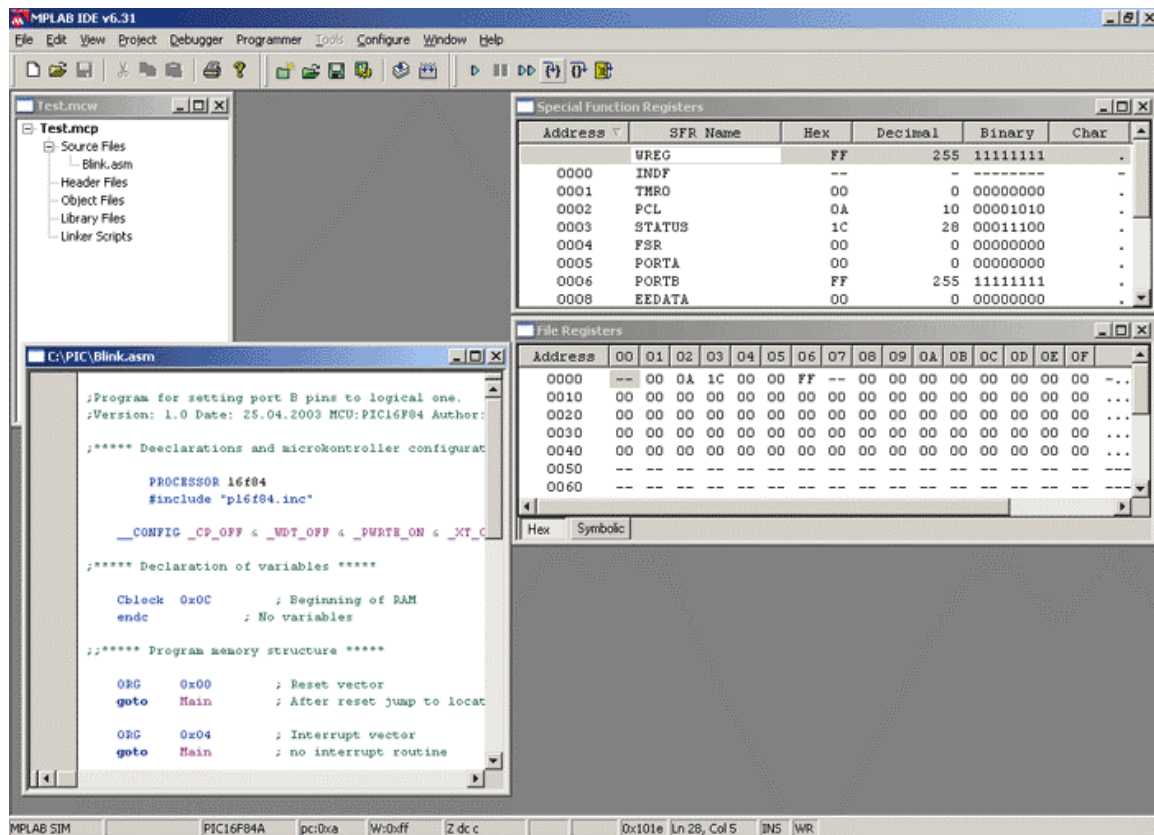


One of the main simulator features is the ability to view register status within a microcontroller. These registers are also called special function registers, or SFR

registers. We can get a window with SFR registers by clicking on VIEW > SPECIAL FUNCTION REGISTERS.

Beside SFR registers, it is useful to have an insight into file registers. Window with file registers can be opened by clicking on VIEW > FILE REGISTERS.

If there are variables in the program, it is good to monitor them, too. Each variable is assigned one window (*Watch Windows*) by clicking on VIEW > WATCH.



Simulator with open SFR registers and File registers windows

When all the variables and registers of interest are placed on the simulator working area, simulation may begin. Next command can be either *Step Into* or *Step Over*, as we may want to go into subroutines or not. Same commands can be issued via keyboard, by clicking F7 or F8.

In the SFR registers window, we can observe how register W receives value 0xFF and delivers it to port B.

By clicking on F7 key again, we don't achieve anything because program has arrived to an "infinite loop". Infinite loop is a term we will meet often. It represents a loop from which a microcontroller cannot get out until interrupt takes place (if it is used in a program), or until a microcontroller is reset.

CHAPTER 5

Macros and subprograms

[Introduction](#)

[5.1 Macros](#)

[5.2 Subprograms](#)

[5.3 Macros used in the examples](#)

Introduction

Same or similar sequence of instructions is frequently used during programming. Assembly language is very demanding. Programmer is required to take care of every single detail when writing a program, because just one incorrect instruction or label can bring about wrong results or make the program doesn't work at all. Solution to this problem is to use already tested program parts repeatedly. For this kind of programming logic, macros and subprograms are used.

5.1 Macros

Macro is defined with directive **macro** containing the name of macro and parameters if needed. In program, definition of macro has to be placed before the instruction line where macro is called upon. When during program execution macro is encountered, it is replaced with an appropriate set of instructions stated in the macro's definition.

```
macro_name      macro par1, par2,..  
                set of instructions  
                set of instructions  
                endm
```

The simplest use of macro could be naming a set of repetitive instructions to avoid errors during retyping. As an example, we could use a macro for selecting a bank of SFR registers or for a global permission of interrupts. It is much easier to have a macro BANK1 in a program than having to memorize which status bit defines the mentioned bank. This is illustrated below: banks 0 and 1 are selected by setting or clearing bit 5 (RP0) of status register, while interrupts are enabled by bit 7 of INTCON register. First two macros are used for selecting a bank, while other two enable and disable interrupts.

```
bank0      macro                ; Macro bank0  
           bcf STATUS, RP0      ; Reset RP0 bit = Bank0  
           endm                 ; End of macro
```

```

bank1      macro                ; Macro bank1
          bsf STATUS, RP0        ; Set RP0 bit = Bank1
          endm                  ; End of macro

enableint  macro                ; Interrupts are globally enabled
          bsf INTCON, 7          ; Set the bit
          endm                  ; End of macro

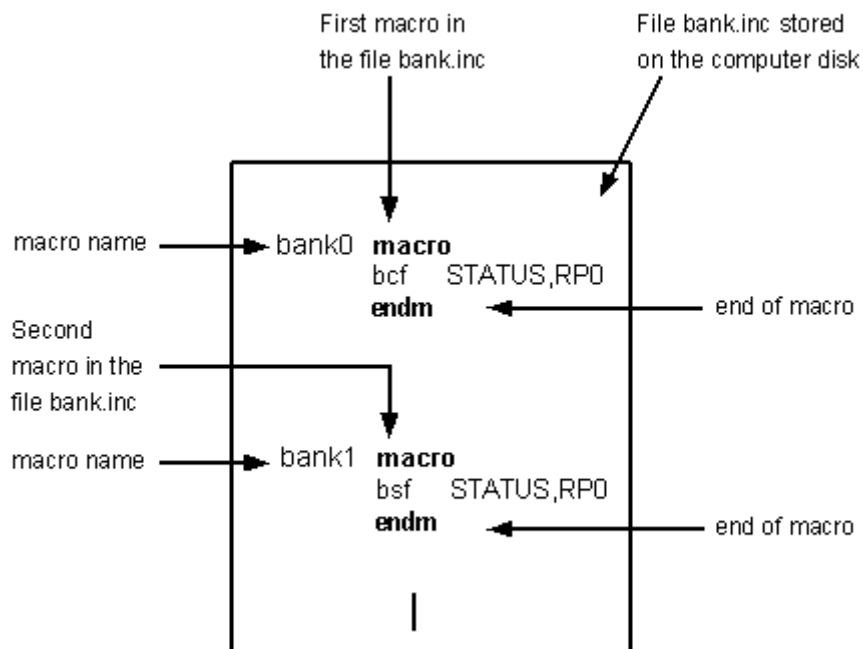
disableint macro                ; Interrupts are globally disabled
          bcf INTCON, 7          ; Reset the bit
          endm                  ; End of macro

```

These macros are to be saved in a special file with extension INC (abbrev. for INCLUDE file). The following image shows the file bank.inc which contains two macros, bank0 and bank1.



*Macros Bank0 and Bank1 are given for illustrational purposes more than practical, since directive **BANKSEL NameSFR** does the same job. Just write **BANKSEL TRISB** and the bank containing the TRISB register will be selected.*



As can be seen above, first four macros do not have parameters. However, parameters can be used if needed. This will be illustrated with the following macros, used for changing direction of pins on ports. Pin is designated as input if the appropriate bit is set (with the position matching the appropriate pin of TRISB register, bank1) , otherwise it's output.

```

input  macro par1, par2    ; Macro input
      bank1                ; In order to access TRIS registers
      bsf par1, par2       ; Set the given bit - 1 = input
      bank0                ; Macro for selecting bank0
      endm                 ; End of macro

output macro par1, par2    ; Macro output
      bank1                ; In order to access TRIS registers
      bcf par1, par2       ; Reset the given bit - 0 = output
      bank0                ; Macro for selecting bank0
      endm                 ; End of macro

```

Macro with parameters can be called upon in following way:

```
output TRISB, 7    ; pin RB7 is output
```

When calling macro first parameter TRISB takes place of the first parameter, *par1*, in macro's definition. Parameter 7 takes place of parameter par2, thus generating the following code:

```

output TRISB, 7          ; Macro output
      bsf STATUS, RP0     ; Set RP0 bit = BANK1
      bcf TRISB, 7        ; Designate RB7 as output
      bcf STATUS, RP0     ; Reset RP0 bit = BANK0
      endm                ; End of macro

```

Apparently, programs that use macros are much more legible and flexible. Main drawback of macros is the amount of memory used - every time macro name is encountered in the program, the appropriate code from the definition is inserted. This doesn't necessarily have to be a problem, but be warned if you plan to use sizeable macros frequently in your program.

In case that macro uses labels, they have to be defined as local using the directive **local**. As an example, below is the macro for calling certain function if *carry* bit in STATUS register is set. If this is not the case, next instruction in order is executed.

```

callc  macro label        ; Macro callc
local  Exit                ; Defining local label within macro

```

	bnc Exit	; If C=0 jump to Exit and exit macro
	call label	; If C=1 call subprogram at the
		; address label outside macro
Exit		; Local label within macro
	endm	; End of macro

5.2 Subprograms

Subprogram represents a set of instructions beginning with a label and ending with the instruction *return* or *retlw*. Its main advantage over macro is that this set of instructions is placed in only one location of program memory. These will be executed every time instruction *call subprogram_name* is encountered in program. Upon reaching *return* instruction, program execution continues at the line succeeding the one subprogram was called from. Definition of subprogram can be located anywhere in the program, regardless of the lines in which it is called.

Label	; subprogram is called with "call Label"
set of instructions	
set of instructions	
set of instructions	
return or retlw	

With macros, use of input and output parameters is very significant. With subprograms, it is not possible to define parameters within the subprogram as can be done with macros. Still, subprogram *can* use predefined variables from the main program as its parameters.

Common course of events would be: defining variables, calling the subprogram that uses them, and then reading the variables which may have been changed by the subprogram.

The following example, *addition.asm* adds two variables, PAR1 and PAR2, and stores the result to variable RES. As 2-byte variables are in question, lower and higher byte has to be defined for each of these. The program itself is quite simple; it first adds lower bytes of variables PAR1 and PAR2, then it adds higher bytes. If two lower bytes total exceeds 255 (maximum for a byte) carry is added to variable RESH.



Basic difference between macro and subprogram is that the macro stands for its definition code (sparing the programmer from additional typing) and can have its own parameters while subprogram saves memory, but cannot have its own parameters.



Program: addition.asm

```
;Program for adding two 16-bit numbers
;Version: 1.0 Date: 25.04.2003 MCU: PIC16F84

PROCESSOR 16f84A          ; Defining the processor
#include "p16f84A.inc"     ; Microchip's INC file
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

Cblock 0x0C               ;RAM starting address
PAR1H          ;Parameter 1 higher byte
PAR1L          ;Parameter 1 lower byte
PAR2H          ;Parameter 2 higher byte
PAR2L          ;Parameter 2 lower byte
RESH          ;higher byte of result
RESL          ;lower byte of result
endc               ;End of variables

ORG 0x00           ;Reset vector
goto Start

Start
    movlw 0x01      ;writing values to variables
    movwf PAR1H     ;PAR1=0x0104
    movlw 0x04
    movwf PAR1L

    movlw 0x07      ;PAR2=0x0705
    movwf PAR2H
    movlw 0x05
    movwf PAR2L

Main
    call Add16      ;main program
    goto Loop       ;Calling subprogram Add16
Loop
    ;remain at this line

Add16
    ;subprogram for adding 2 16-bit numbers
    clrf RESH       ;RESH=0
    movf PAR1L,w    ;w=PAR1L
    addwf PAR2L,w    ;w=w+PAR2L
    movwf RESL      ;RESL=w
    btfsc STATUS,C  ;does result exceed 255?
    incf RESH,f     ;if true increment RESH by one

    movf PAR1H,w    ;w=PAR1H
    addwf PAR2H,w    ;w=w+PAR2
    addwf RESH,f    ;RESH=w
    return          ;return from subprogram
end                ;end of program
```

5.3 Macros used in the examples

Examples given in chapter 6 frequently use macros *ifbit*, *ifnotbit*, *digbyte*, and *pausems*, so these will be explained in detail. The most important thing is to comprehend the function of the following macros and the way to use them, without

unnecessary bothering with the algorithms itself. All macros are included in the file *mikroel84.inc* for easier reference.

5.3.1 Jump to label if bit is set

```
ifbit    macro par1, par2, par3
          btfsc par1, par2
          goto par3
          endm
```

Macro is called with : ifbit Register, bit, label

5.3.2 Jump to label if bit is cleared

```
ifnotbit macro par1, par2, par3
          btfss par1, par2
          goto par3
          endm
```

Macro is called with : ifnotbit Register, bit, label

Next example shows how to use a macro. Pin 0 on port A is checked and if set, program jumps to label *ledoff*, otherwise macro *ifnotbit* executes, directing the program to label *ledon*.

Program: macrotest.asm

In this line we include the file with macros

Macros are called in these lines. The effect is same as if we had all the instructions from macro's definition at this point.

```

;Program for testing macros ifbit and ifnotbit
;Version: 1.0 Date: 25.04.2003 MCU:PIC16F84

    PROCESSOR 16f84A           ;Defining the processor
    #include "p16f84A.inc"      ;Microchip's INC file
    _CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

    Chlock 0x0C                ;RAM starting address
    endc                       ;No variables

    ORG 0x00                   ;Reset vector
    goto Start

Start bsf STATUS,RP0           ;Selecting the bank containing TRISA
      bsf TRISA,0              ;TRISA0=1, meaning the pin is input
      bcf TRISB,7              ;TRISB7=0, meaning the pin is output
      bcf STATUS,RP0          ;Selecting the bank containing PORTA

      include "mikroel84.inc"   ;Our INC file with 2 macros
Main  ifbit PORTA,0,Ledoff      ;if bit RA0=1 switch on LED
      ifnotbit PORTA,0,Ledon    ;if bit RA0=0 switch off LED
      goto Main                ;repeat all

Ledon  bsf PORTB,7              ;Switch on LED on RB7
       goto Main

Ledoff bcf PORTB,7              ;Switch off LED on RB7
       goto Main
      End

```

5.3.3 Extracting ones, tens and hundreds from variable

Typical use for this macro is displaying variables on LCD or 7seg display.

```

digbyte macro par0
        local Pon0
        local Exit1
        local Exit2
        local Positive
        local Negative
        clrf Dig1
        clrf Dig2
        clrf Dig3

Positive
        movf par0, w
        movwf Digtemp
        movlw .100

Pon0    incf Dig1                ;computing hundreds digit
        subwf Digtemp
        btfsc STATUS, C

```

```

                                goto Pon0
                                decf Dig1, w
                                addwf Digtemp, f
Exit1    movlw .10                ;computing tens digit
                                incf Dig2, f
                                subwf Digtemp, f
                                btfsc STATUS, C
                                goto Exit1
                                decf Dig2, f
                                addwf Digtemp, f
Exit2    movf Digtemp, w          ;computing ones digit
                                movwf Dig3
                                endm

```

Macro is called with :

```

movlw .156    ; w = 156
movwf RES     ; RES = w
digbyte RES   ; now Dec1<-1, Dec2<-5, Dec3<-6

```

The following example shows how to use macro *digbyte* in program. At the beginning, we have to define variables for storing the result, *Dig1*, *Dig2*, *Dig3*, as well as auxiliary variable *Digtemp*.

Program: extractingdigits.asm

```

;Program for extracting digits from numerical value
;Version: 1.0 Date: 25.04.2003 MCU:PIC16F84

    PROCESSOR 16f84A      ;Defining the processor
    #include "p16f84A.inc" ; Microchip's INC file
    _CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

    Cblock 0x0C           ;RAM starting address
    RES                                ;Numerical value
    Digtemp                ;auxiliary variable
    Dig1                   ;First digit
    Dig2                   ;Second digit
    Dig3                   ;Third digit
    endc                   ;end of variables

    ORG 0x00              ;Reset vector
    goto Start
    include "digbyte.inc"  ;including file with macro

Start movlw 0xff          ;w=255
      movwf RES           ;RES=255
Main  digbyte RES         ;main program
      digbyte RES         ;calling a macro

Loop  goto Loop           ;remain at this line
      end                ;end of program

```

5.3.4 Generating pause in miliseconds (1~65535ms)

Purpose of this macro is to provide exact time delays in program.

```

pausems macro par1
    local Loop1
    local dechi
    local Delay1ms
    local Loop2
    local End
    movlw high par1      ; Higher byte of parameter 1 goes to HIcnt
    movwf HIcnt
    movlw low par1       ; Lower byte of parameter 1 goes to LOcnt
    movwf LOcnt

Loop1
    movf LOcnt, f        ; Decrease HIcnt and LOcnt necessary
    btfsc STATUS, Z      ; number of times and call subprogram Delay1ms
    goto dechi
    call Delay1ms

```

```

        decf LOcnt, f
        goto Loop1
dechi
        movf HIcnt, f
        btfsc STATUS, Z
        goto End
        call Delay1ms
        decf HIcnt, f
        decf LOcnt, f
        goto Loop1
Delay1ms:                ; Delay1ms produces a one millisecond delay
        movlw .100        ; 100*10us=1ms
        movwf LOOPcnt    ; LOOPcnt<-100
Loop2:
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        decfsz LOOPcnt, f
        goto Loop2        ; Time period necessary to execute loop Loop2
        return            ; equals 10us
End
        endm

```

This macro is written for an 4MHz oscillator. For instance, with 8MHz oscillator, pause will be halved. It has very wide range of applications, from simple code such as blinking diodes to highly complicated programs that demand accurate timing. Following example demonstrates use of macro *pausems* in a program. At the beginning of the program we have to define auxiliary variables *HIcnt*, *LOcnt*, and *LOPcnt*.



Program: ledblink.asm

;Program for blinking LED diodes on port B
;Version: 1.0 Date: 25.04.2003 MCU:PIC16F84

```
PROCESSOR 16f84A      ;Defining the processor
#include "p16f84A.inc" ;Microchip's INC file
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
Cblock 0x0C            ;RAM starting address
Hicnt                 ;higher byte of macro par.
LOcnt                 ;lower byte of macro par.
LOOPcnt               ;macro variable
endc                  ;end of variables

ORG 0x00              ;Reset vector
goto Main
ORG 0x04              ;Interrupt vector
goto Main              ;no interrupt routine
include "pause.inc"   ;including file with macro

Main                  ;Main program
    banksel TRISB      ;Sel. bank containing TRISB
    clrf TRISB         ;Port B is output
    banksel PORTB      ;Sel. bank containing PORTB

Loop
    movlw 0x00          ;Switch off diodes on port B
    movwf PORTB
    pausems .500        ;500 milisecond delay (0.5sec)
    movlw 0xff          ;Switch on diodes on port B
    movwf PORTB
    pausems .500        ;500 milisecond delay (0.5sec)
    goto Loop           ;jump to label Loop
End
```

CHAPTER 6

Examples for subsystems within microcontroller

[Introduction](#)

[6.1 Writing to and reading from EEPROM](#)

[6.2 Processing interrupt caused by changes on pins RB4-RB7](#)

[6.3 Processing interrupt caused by change on pin RB0](#)

[6.4 Processing interrupt caused by overflow on timer TMR0](#)

[6.5 Processing interrupt caused by overflow on TMR0 connected to external input \(TOCKI\)](#)


Introduction

Every microcontroller comprises a number of subsystems allowing for flexibility and wide range of applications. These include internal EEPROM memory, AD converters, serial or other form of communication, timers, interrupts, etc. Two most commonly utilized elements are interrupts and timers. One of these or several in combination can create a basis for useful and practical programs.

6.1 Writing to and reading from EEPROM

Program "eeprom.asm" uses EEPROM memory for storing certain microcontroller parameters. Transfer of data between RAM and EEPROM has two steps - calling macros *eewrite* and *eeread*. Macro *eewrite* writes certain variable to a given address, while *eeread* reads the given address of EEPROM and stores the value to a variable.

Macro *eewrite* writes the address to EEADR register and the variable to EEDATA register. It then calls the subprogram which executes the standard procedure for initialization of writing data (setting WREN bit in EECON1 register and writing control bytes 0x55 and 0xAA to EECON2).



Program: EEPROM.INC

```

eewrite macro addr,prom
    movlw addr                ;Pass the parameters to subprogram E2write
    movwf EEADR
    movf prom,w
    movwf EEDATA
    call E2write              ;Call E2write which initializes writing data
    pausems .10               ;10ms delay
endm

E2write
    bcf INTCON,GIE            ;disable all interrupts
    bsf STATUS,RPO            ;select bank 1
    bsf EECON1,WREN           ;set WREN bit in EECON1
    movlw 0x55                ;write 0x55 to EECON2
    movwf EECON2
    movlw 0xaa                ;write 0xaa to EECON2, as
    movwf EECON2              ;writing confirmation
    bsf EECON1,WR             ;set bit WR to initialize writing
    bcf STATUS,RPO
    return

eeread macro addr,prom        ;Pass the parameters to subprogram E2read
    movlw addr
    movwf EEADR
    call E2read               ;Call E2write which initializes reading data
    movwf prom
endm

E2read
    bsf STATUS,RPO
    bsf EECON1,RD
    bcf STATUS,RPO
    movf EEDATA,w
    return
  
```

For data to be actually stored in EEPROM, 10ms delay is necessary. This is achieved by using macro *pausems*. In case that this pause is unacceptable for any reason, problem can be solved by using an interrupt for signaling that data is written to EEPROM.

- | | |
|----------------|--|
| eewrite | macro addr, var |
| addr | Destination address. With PIC16F84, there are 68 bytes of EEPROM for a total address range of 0x00 - 0x44. |
| var | Name of the variable to be stored to EPROM |
| eeread | macro addr, var |
| addr | Destination address. With PIC16F84, there are 68 bytes of EEPROM for a total address range of 0x00 - 0x44. |
| var | Name of the variable into which data read from EPROM will be stored. |

Example: Variable *volume*, which is set via buttons RA0 and RA1, will be stored to the address 0 of EEPROM. After reboot, when the program is started, it first loads the last known value of variable *volume* from EEPROM.



;***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "p16f84.inc"

_CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaring variables *****

Cblock 0x0c ; RAM starting address
Hicnt
Locnt
LOOPcnt
volume
endc

;***** Structure of program memory *****

ORG 0x00 ;Reset vector
goto Main

ORG 0x04 ;Interrupt vector
goto Main ;no interrupt routine

include "mikroel84.inc"
include "button.inc"
include "eeprom.inc"

Main ;Main program

banksel TRISB
clrf TRISB
banksel PORTB

eeread 0x00,volume ;Read the previous value of volume
movwf PORTB

Loop

button PORTA,0,0,Increase
button PORTA,1,0,Decrease
goto Loop

Increase

incf volume,f ;Increase volume by 1
movf volume, W ;display on port B
movwf PORTB
eewrite 0x00,volume ;write to eeprom, address 0x00
goto Loop

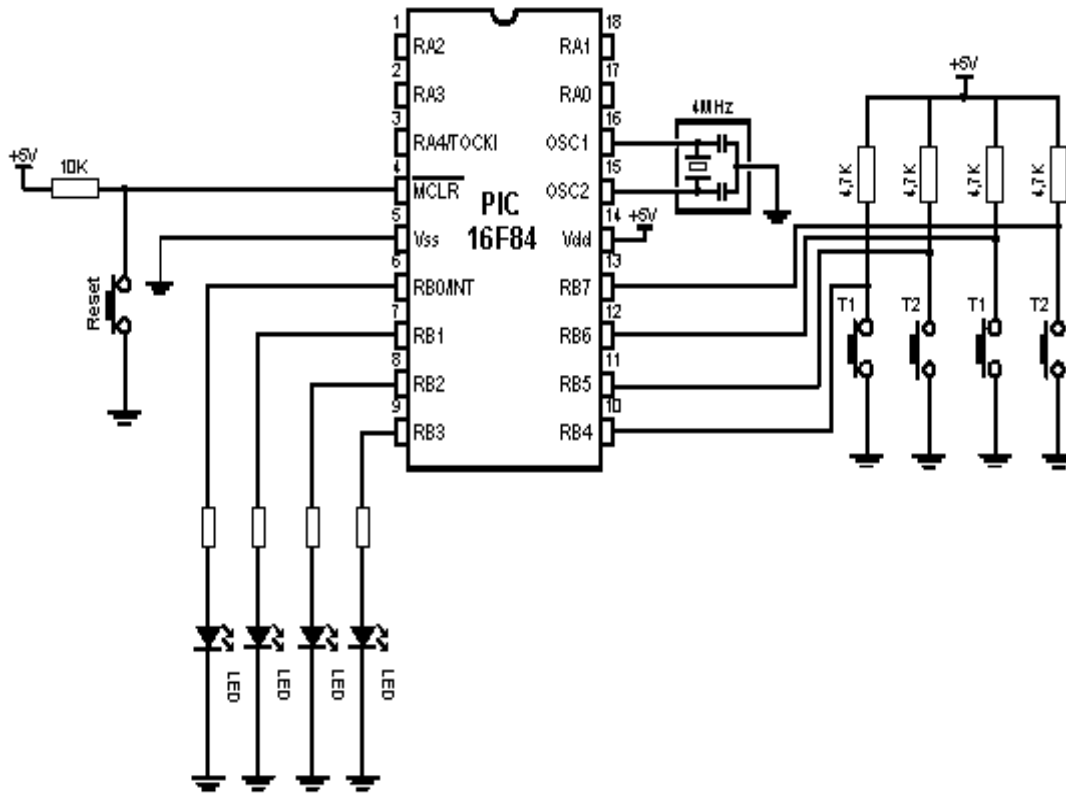
Decrease

decf volume,f ;Decrease volume by 1
movf volume, W ;display on port B
movwf PORTB
eewrite 0x00,volume ;write to eeprom, address 0x00
goto Loop

org 0x2100 ;Starting value of EEPROM address zero
de .5 ;after the microcontroller is programmed is 0x05
End

6.2 Processing interrupt caused by changes on pins RB4-RB7

Program "intportb.asm" illustrates how interrupt can be employed for indicating changes on pins RB4-RB7. Upon pushing any of the buttons, program enters the interrupt routine and determines which pin caused an interrupt. This program could be utilized in systems with battery power supply, where power consumption plays an important role. It is useful to set microcontroller to low consumption mode with a *sleep* instruction. Microcontroller is practically on stand-by, saving energy until the occurrence of interrupt.



Example of processing interrupt caused by changes on pins RB4-RB7



```
;***** Declaring and configuring a microcontroller *****
```

```
PROCESSOR 16f84
#include "p16f84.inc"
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC
```

```
;***** Structure of program memory *****
```

```
org 0x00
goto Main
org 0x04
goto ISR
```

Main

```
banksel TRISB
movlw 0xf0           ;Higher four LED diodes are on
movwf TRISB
banksel PORTB
movlw 0xff
movwf PORTB
bsf INTCON,RBIE      ;interrupt upon pin change enabled
bsf INTCON,GIE       ;all interrupts are enabled
```

Loop

```
goto Loop           ;Main loop
```

ISR

```
bcf INTCON,RBIF      ;Clears the flag that indicates RB interrupt
                     ;took place thus enabling detection of
                     ;new interrupts in main program
                     ;Determining which button caused the interrupt

btfss PORTB,7
goto Led0
btfss PORTB,6
goto Led1
btfss PORTB,5
goto Led2
btfss PORTB,4
goto Led3
retfie
```

Led0

```
bcf PORTB,0          ;Switch off diode LD0
retfie
```

Led1

```
bcf PORTB,1          ;Switch off diode LD1
retfie
```

Led2

```
bcf PORTB,2          ;Switch off diode LD2
retfie
```

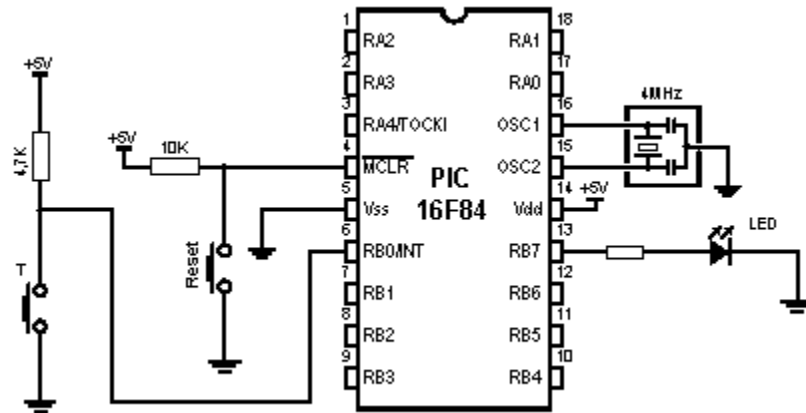
Led3

```
bcf PORTB,3          ;Switch off diode LD3
retfie
```

```
End
```

6.3 Processing interrupt caused by change on pin RB0

Example "intrb0.asm" demonstrates use of interrupt RB0/INT. Upon falling edge of the impulse coming to RB0/INT pin, program jumps to subprogram for processing interrupt. This routine then performs a certain operation, in our case it blinks the LED diode on PORTB, 7.



Example of processing interrupt caused by changes on pin RB0



```
;***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "p16f84.inc"
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Structure of program memory *****

org 0x00
goto Main
org 0x04
goto ISR

Main
    banksel TRISB
    movlw b'00000001'           ;RB0 is input, the rest are output
    movwf TRISB
    banksel OPTION_REG
    bcf OPTION_REG,INTEDG       ;interrupt occurs at falling edge
    bsf OPTION_REG,NOT_RBPU     ;internal pull-up resistors are off
    banksel PORTB
    clrf PORTB
    bsf PORTB,7                 ;Only LED diode PORTB,7 is on
    bsf INTCON,INTE             ;interrupt RB0 enabled
    bsf INTCON,GIE              ;all interrupts enabled

Loop
    goto Loop                   ;Main loop

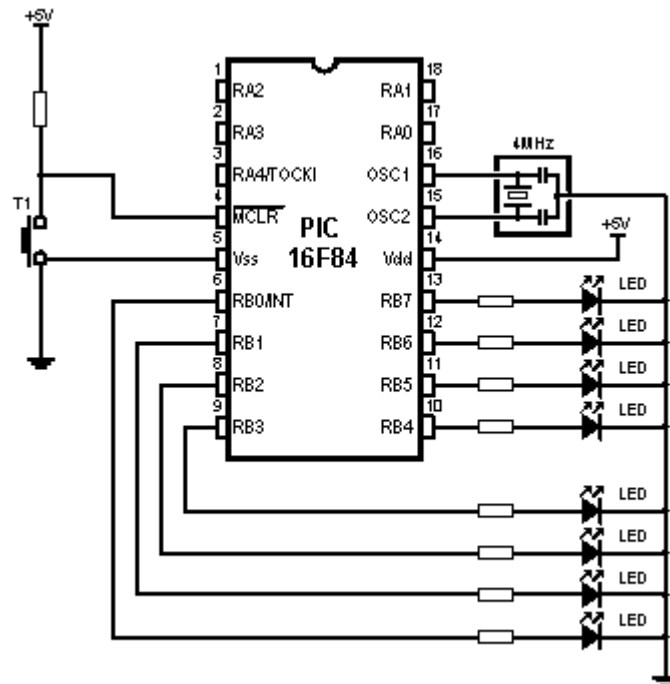
ISR
    bcf INTCON,INTF             ;Clears the flag that indicates RB interrupt
                                ;took place thus enabling detection of
                                ;new interrupts in main program
    btfss PORTB,7               ;Is LED7 on?
    goto Lab1
    bcf PORTB,7                 ;If true switch off LED7
    retfie

Lab1
    bsf PORTB,7                 ;If false switch on LED7
    retfie
End
```

6.4 Processing interrupt caused by overflow on timer TMR0

Program "inttmr0.asm" illustrates how interrupt TMR0 can be employed for generating specific periods of time. Diodes on port B are switched on and off

alternately every second. Interrupt is generated every 5.088ms; in interrupt routine variable *cnt* is incremented to the cap of 196, thus generating approx. 1 second pause ($5.088\text{ms} \times 196$ is actually 0.99248s). Pay attention to initialization of OPTION register which enables this mode of work for timer TMR0.



Example of processing interrupt caused by overflow on timer TMR0



```
;***** Declaring and configuring a microcontroller *****

    PROCESSOR 16f84
    #include "pl6f84.inc"

    _CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaring variables *****

    cnt      equ 0x0c

;***** Structure of program memory *****

    ORG      0x00          ;Reset vector
    goto     Main

    ORG      0x04          ;Interrupt vector
    goto     ISR

Main
    banksel  TRISB
    clrf     TRISB         ;Port B is output
    movlw    TRISA         ;Port A is input
    movlw    B'10000100'   ;Set prescaler to TMRO
    banksel  OPTION_REG
    movwf    OPTION_REG    ;ps = 32=> TMRO is incremented every 32us
    banksel  PORTB
    clrf     PORTB         ;All the diodes are off by default
    bsf      INTCON,TOIE    ;Enable TMRO interrupt
    movlw    .96           ;Initialize TMRO
                                ;Overflow occurs
                                ;every (255-96)*32us=5.088ms
    movwf    TMRO          ;Start the counter

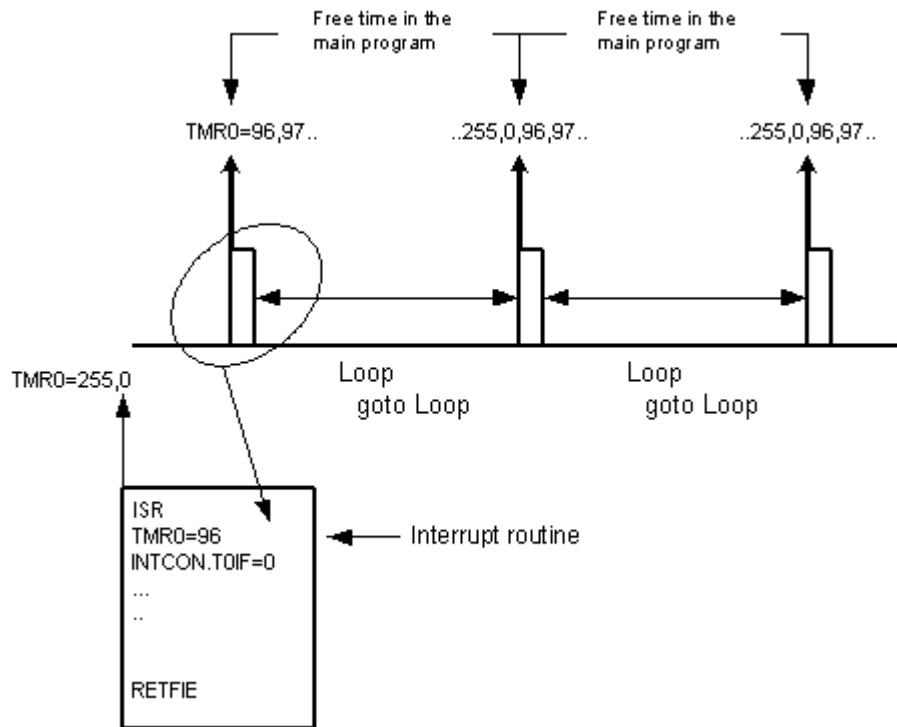
    bsf      INTCON,GIE     ;Interrupts are globally enabled
    clrf     cnt

loop
    goto     loop          ;Remain at this line

ISR
    movlw    .96           ;Initialize TMRO to ensure next interrupt
                                ;in 5ms
    movwf    TMRO
    bcf      INTCON,TOIF    ;clear int. flag

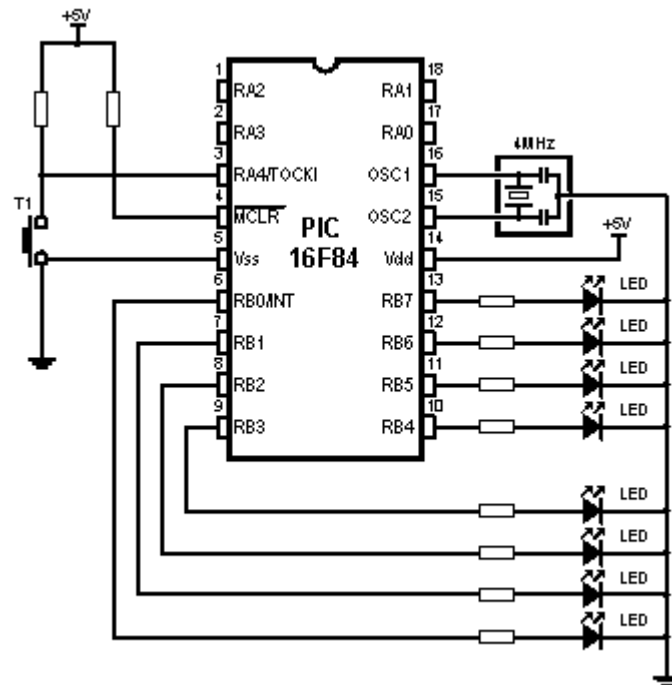
    incf     cnt,F
    movlw    .196          ;Has one second elapsed?(1s~196*5.088 ms)
    subwf    cnt,W
    bt fss   STATUS,Z
    retfie
    comf     PORTB,f       ;If true, complement the values of port B
    clrf     cnt           ;and set the initial value of variable cnt
    retfie                ;If false, exit the interrupt routine

End                      ;End of program
```



6.5 Processing interrupt caused by overflow on TMR0 connected to external input (TOCKI)

Counter TMR0 increments upon signal change on pin RA4/TOCKI. Prescaler is set to 4, meaning that TMR0 will be incremented on every fourth impulse. Pay attention to initialization of OPTION register which enables this mode of work for timer TMR0 (this mode is common for devices such as counters).



Example of processing interrupt caused by overflow on timer TMR0 connected to T0CKI



***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "pl6f84.inc"

_CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

***** Declaring constants *****

num_rev equ .100; (1-256)

***** Structure of program memory *****

ORG 0x00 ; Reset vector
goto Main

ORG 0x04 ; Interrupt vector
goto ISR

Main

banksel TRISB
clrf TRISB ; Port B is input
movlw TRISA ; Port A is output
movlw B'10100001' ; External impulses on TOCKI increment
; TMRO
movwf OPTION_REG ; ps = 4 TMRO increments every 4 impulses
banksel PORTB ; i.e. 1 revolution = 4 impulses
clrf PORTB ; All diodes are off by default
bsf INTCON,TOIE ; Enable TMRO interrupt
movlw .256 ; Initialize TMRO so that overflow
sublw num_rev ; occurs every 100 revolutions,
; i.e. TMRO is set to 156
; 256-156=100)
movwf TMRO ; Start the counter

bsf INTCON,GIE ; Interrupts are globally enabled

loop

; Display number of remaining revolutions
; on port B
movf TMRO,W
sublw .256 ; number of remaining revolutions=256-TMRO
movwf PORTB
goto loop ; Remain at this line

ISR

movlw .256 ; Initialize TMRO so it can count next
; 100 revolutions
sublw broj_obrtaja
movwf TMRO

bcf INTCON,TOIF ; clear int. fleg
retfie ; return to main program

End ; End of program

CHAPTER 7

Examples

[Introduction](#)

[7.1 Supplying the microcontroller](#)

[7.2 LED diodes](#)

[7.3 Push buttons](#)

[7.4 Optocoupler](#)

[7.4.1 Optocoupler on input line](#)

[7.4.2 Optocoupler on output line](#)

[7.5 Relay](#)

[7.6 Generating sound](#)

[7.7 Shift registers](#)

[7.7.1 Input shift register](#)

[7.7.2 Output shift register](#)

[7.8 7-seg display \(multiplexing\)](#)

[7.9 LCD display](#)

[7.10 Software SCI communication](#)

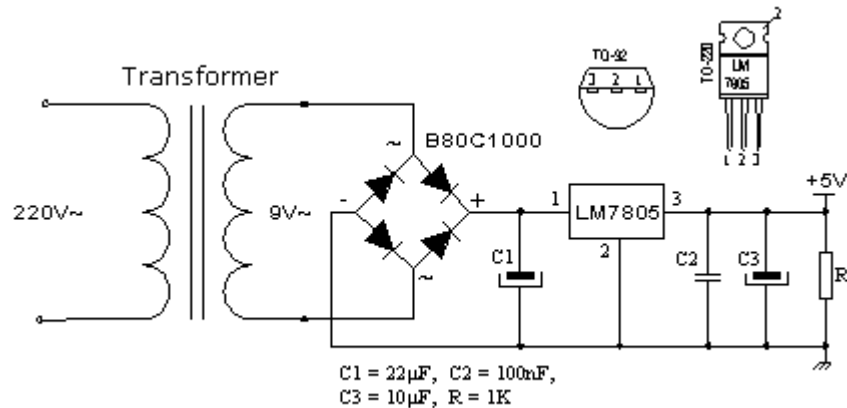
Introduction

Examples given in this chapter will show you how to connect the PIC microcontroller with other peripheral components or devices when developing your own microcontroller system. Each example contains detailed description of hardware with electrical outline and comments on the program. All programs can be taken directly from the 'MikroElektronika' Internet presentation.

7.1 Supplying the microcontroller

Generally speaking, the correct voltage supply is of utmost importance for the proper functioning of the microcontroller system. It can easily be compared to a man breathing in the air. It is more likely that a man who is breathing in fresh air will live longer than a man who's living in a polluted environment.

For a proper function of any microcontroller, it is necessary to provide a stable source of supply, a sure reset when you turn it on and an oscillator. According to technical specifications by the manufacturer of PIC microcontroller, supply voltage should move between 2.0V to 6.0V in all versions. The simplest solution to the source of supply is using the voltage stabilizer LM7805 which gives stable +5V on its output. One such source is shown in the picture below.



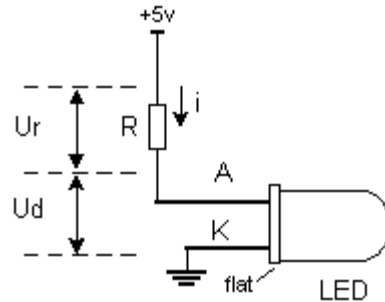
In order to function properly, or in order to have stable 5V at the output (pin 3), input voltage on pin 1 of LM7805 should be between 7V through 24V. Depending on current consumption of device we will use the appropriate type of voltage stabilizer LM7805. There are several versions of LM7805. For current consumption of up to 1A we should use the version in TO-220 case with the capability of additional cooling. If the total consumption is 50mA, we can use 78L05 (stabilizer version in small TO - 92 packaging for current of up to 100mA).

7.2 LED diodes

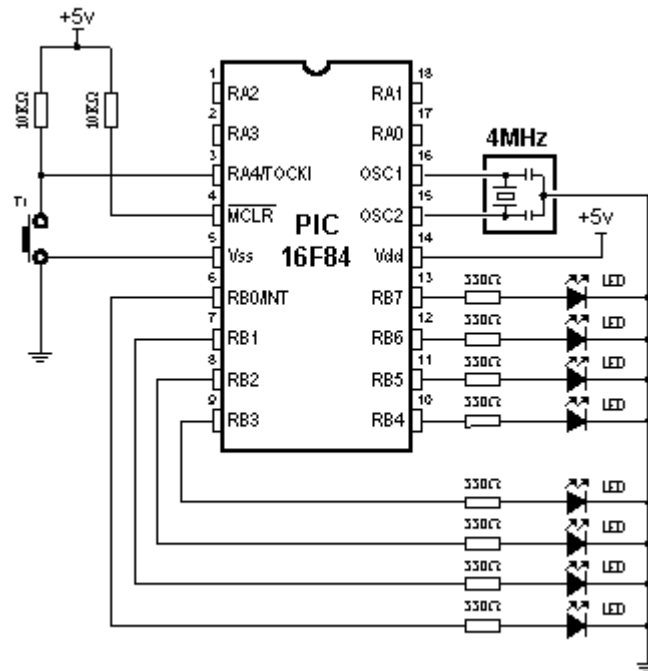
LEDs are surely one of the most commonly used elements in electronics. LED is an abbreviation for 'Light Emitting Diode'. When choosing a LED, several parameters should be looked at: diameter, which is usually 3 or 5 mm (millimeters), working current which is usually about 10mA (It can be as low as 2mA for LEDs with high efficiency - high light output), and color of course, which can be red or green though there are also orange, blue, yellow....

LEDs must be connected around the correct way, in order to emit light and the current-limiting resistor must be the correct value so that the LED is not damaged or burn out (overheated). The positive of the supply is taken to the anode, and the cathode goes to the negative or ground of the project (circuit). In order to identify each lead, the cathode is the shorter lead and the LED "bulb" usually has a cut or "flat" on the cathode side. Diodes will emit light only if current is flowing from anode to cathode. Otherwise, its PN junction is reverse biased and current won't flow. In order to connect a LED correctly, a resistor must be added in series that to limit the amount of current through the diode, so that it does not burn out. The value of the resistor is determined by the amount of current you want to flow through the LED. Maximum current flow through LED was defined by manufacturer.

To determine the value of the dropper-resistor, we need to know the value of the supply voltage. From this we subtract the characteristic voltage drop of a LED. This value will range from 1.2v to 1.6v depending on the color of the LED. The answer is the value of **Ur**. Using this value and the current we want to flow through the LED (0.002A to 0.01A) we can work out the value of the resistor from the formula **$R = U_r / I$** .



LEDs are connected to a microcontroller in two ways. One is to switch them on with logic zero, and other to switch them on with logic one. The first is called NEGATIVE logic and the other is called POSITIVE logic. The next diagram shows how to connect POSITIVE logic. Since POSITIVE logic provides a voltage of +5V to the diode and dropper resistor, it will emit light each time a pin of port B is provided with a logic 1. The other way is to connect all anodes to +5V and to deliver logical zero to cathodes.



Connecting LED diodes to PORTB microcontroller

The following example initializes port B as output and alternately switches on and off LED diodes every 0.5sec. For pause we used macro *pausems*, which is defined in the file mikroel84.inc.



```
;***** Declaring and configuring a microcontroller *****

    PROCESSOR 16f84
    #include "pl6f84.inc"

    __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaring variables *****

    Cblock 0x0C      ;RAM starting address
    H1cnt           ;Auxiliary variables for macro pauses
    L0cnt
    L00Pcnt
    endc

;***** Structure of program memory *****

    ORG 0x00        ;Reset vector
    goto Main

    ORG 0x04        ;Interrupt vector
    goto Main       ;no interrupt routine

    include "mikroel84.inc"

Main                                ;Main program

    banksel TRISB      ;Select bank containing TRISB
    clrf TRISB         ;Port B is output
    banksel PORTB      ;Select bank containing PORTB

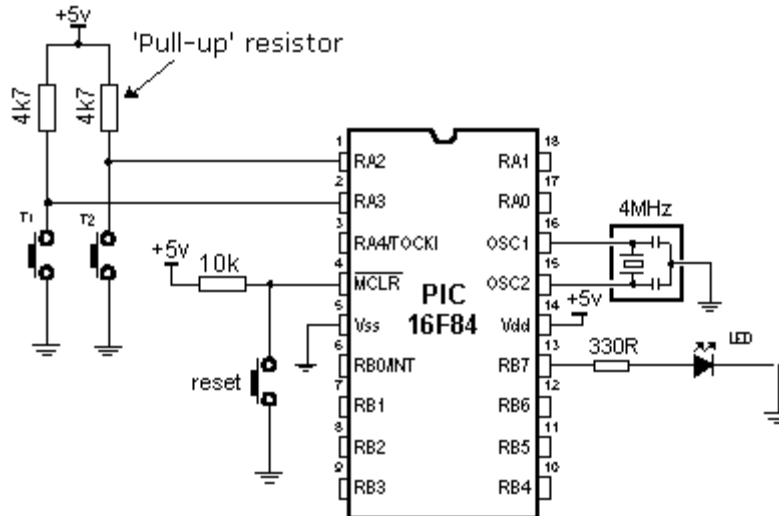
Loop

    movlw 0x00         ;Switch off diodes on port B
    movwf PORTB
    pausems .500       ;500ms delay (0.5sec)
    movlw 0xff         ;Switch on diodes on portu B
    movwf PORTB
    pausems .500       ;500ms delay (0.5sec)
    goto Loop          ;Jump to label Loop

End
```

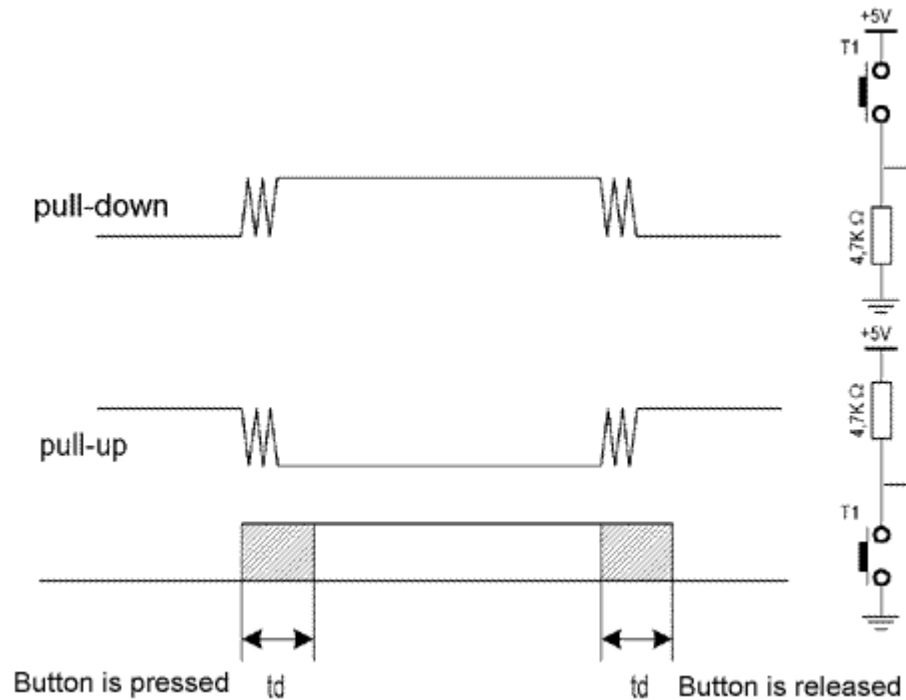
7.3 Push buttons

Buttons are mechanical devices used to execute a break or make connection between two points. They come in different sizes and with different purposes. Buttons that are used here are also called "dip-buttons". They are soldered directly onto a printed board and are common in electronics. They have four pins (two for each contact) which give them mechanical stability.



Example of connecting buttons to microcontroller pins

Button function is simple. When we push a button, two contacts are joined together and connection is made. Still, it isn't all that simple. The problem lies in the nature of voltage as an electrical dimension, and in the imperfection of mechanical contacts. That is to say, before contact is made or cut off, there is a short time period when vibration (oscillation) can occur as a result of unevenness of mechanical contacts, or as a result of the different speed in pushing a button (this depends on person who pushes the button). The term given to this phenomena is called SWITCH (CONTACT) DEBOUNCE. If this is overlooked when program is written, an error can occur, or the program can produce more than one output pulse for a single button push. In order to avoid this, we can introduce a small delay when we detect the closing of a contact. This will ensure that the push of a button is interpreted as a single pulse. The debounce delay is produced in software and the length of the delay depends on the button, and the purpose of the button. The problem can be partially solved by adding a capacitor across the button, but a well-designed program is a much-better answer. The program can be adjusted until false detection is completely eliminated. Image below shows what actually happens when button is pushed.



As buttons are very common element in electronics, it would be smart to have a macro for detecting the button is pushed. Macro will be called *button*. *Button* has several parameters that deserve additional explanation.

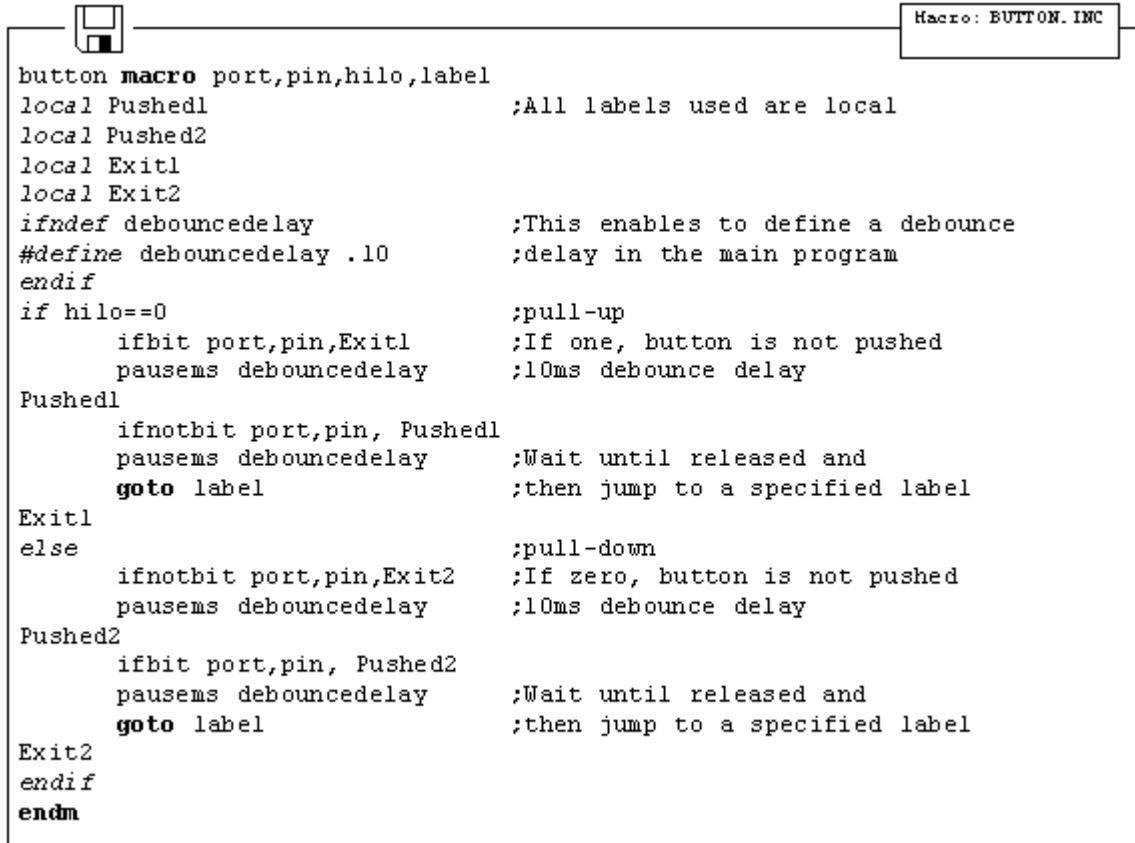
button **macro** port, pin, hilo, label

Port is a microcontroller's port to which a button is connected. In case of a PIC16F84 microcontroller, it can be PORTA or PORTB.

Pin is port's pin to which the button is connected.

HiLo can be '0' or '1' which represents the state when the button is pushed.

Label is a destination address for jump to a service subprogram which will handle the event (button pushed).



Example 1:

```
button PORTA, 3, 1, Button1
```

Button T1 is connected to pin RA3 and to the mass across a pull-down resistor, so it generates logical one upon push. When the button is released, program jumps to the label Button1.

Example 2:

```
button PORTA, 2, 0, Button2
```

Button T1 is connected to pin RA1 and to the mass across a pull-up resistor, so it generates logical zero upon push. When the button is released, program jumps to the label Button2.

The following example illustrates use of macro *button* in a program. Buttons are connected to the supply across pull-up resistors and connect to the mass when pushed. Variable *cnt* is displayed on port B LEDs; *cnt* is incremented by pushing the button RA0, and is decremented by pushing the button RA1.

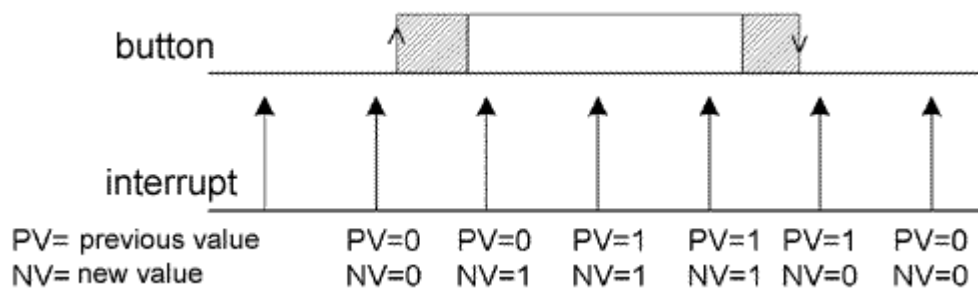


```
;***** Declaring and configuring a microcontroller *****  
  
    PROCESSOR 16f84  
    #include "p16f84.inc"  
  
    __CONFIG __CP_OFF & __WDT_OFF & __PWRTE_ON & __XT_OSC  
;***** Declaring variables *****  
  
    Cblock 0x0C      ;RAM starting address  
    H1cnt  
    L0cnt  
    LOOPcnt  
    cnt  
    endc  
;***** Structure of program memory *****  
  
    ORG 0x00          ;Reset vector  
    goto Main  
  
    ORG 0x04          ;Interrupt vector  
    goto Main          ;no interrupt routine  
  
    include "mikroel84.inc"  
    include "button.inc"  
Main                      ;Main program  
    input PORTA,0  
    input PORTA,0  
    banksel TRISB  
    clrf TRISB  
    banksel PORTB  
    clrf cnt  
Loop  
    button PORTA,0,0,Increase  
    button PORTA,1,0,Decrease  
    goto Loop  
Increase  
    incf cnt,f  
    movf cnt,w  
    movwf PORTB  
    goto Loop  
Decrease  
    decf cnt,f  
    movf cnt,w  
    movwf PORTB  
    goto Loop  
  
End
```

It is important to note that this kind of debouncing has certain drawbacks, mainly concerning the idle periods of microcontroller. Namely, microcontroller is in the state of waiting from the moment the button is pushed until it is released, which can be a very long time period in certain applications. If you want the program to be attending to a number of things at the same time, different approach should be used from the

start. Solution is to use the interrupt routine for each push of a button, which will occur periodically with pause adequate to compensate for repeated pushes of button.

The idea is simple. Every 10ms, button state will be checked upon and compared to the previous input state. This comparison can detect rising or falling edge of the signal. In case that states are same, there were apparently no changes. In case of change from 0 to a 1, rising edge occurred. If succeeding 3 or 4 checks yield the same result (logical one), we can be positive that the button is pushed.



7.4 Optocouplers

Optocouplers were discovered right after photo-transistors (like any other transistor, except it is stimulated by light), by combining a LED and photo-transistor in the same case. The purpose of an optocoupler is to separate two parts of a circuit.

This is done for a number of reasons:

- Interference. Typical examples are industrial units with lots of interferences which affect signals in the wires. If these interferences affected the function of control section, errors would occur and the unit would stop working.
- Simultaneous separation and intensification of a signal. Typical examples are relays which require higher current than microcontroller pin can provide. Usually, optocoupler is used for separating microcontroller supply and relay supply.
- In case of a breakdown, optocoupled part of device stays safe in its casing, reducing the repair costs.

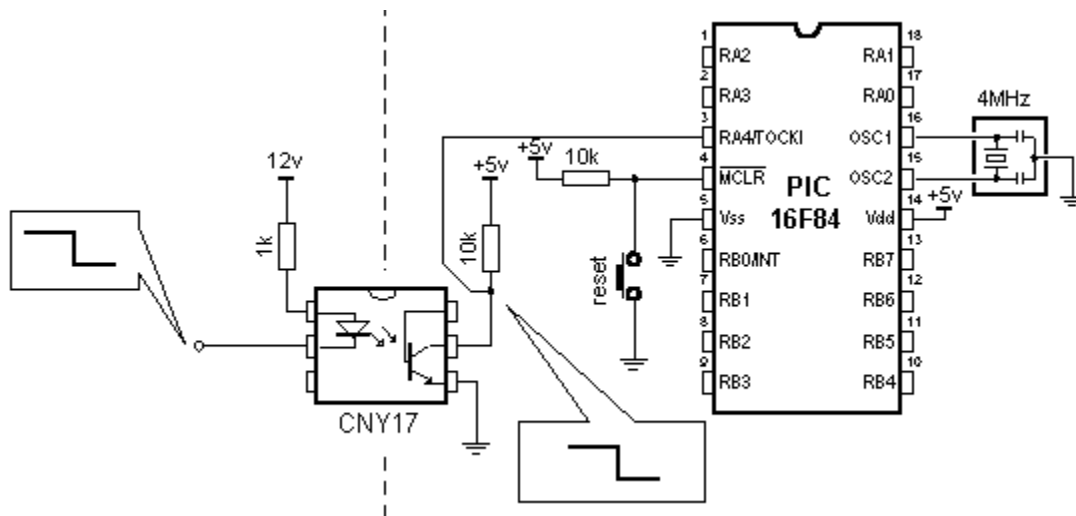
Optocouplers can be used as either input or output devices. They can have additional functions such as intensification of a signal or Schmitt triggering (the output of a Schmitt trigger is either 0 or 1 - it changes slow rising and falling waveforms into definite low or high values). Optocouplers come as a single unit or in groups of two or more in one casing.

Each optocoupler needs two supplies in order to function. They can be used with one supply, but the voltage isolation feature, which is their primary purpose, is lost.


7.4.1 Optocoupler on an input line

The way it works is simple: when a signal arrives, the LED within the optocoupler is turned on, and it illuminates the base of a photo-transistor within the same case. When the transistor is activated, the voltage between collector and emitter falls to 0.7V or less and the microcontroller sees this as a logic zero on its RA4 pin.

The example below is a simplified model of a counter, element commonly utilized in industry (it is used for counting products on a production line, determining motor speed, counting the number of revolutions of an axis, etc). We will have sensor set off the LED every time axis makes a full revolution. LED in turn will 'send' a signal by means of photo-transistor to a microcontroller input RA4 (TOCKI). As prescaler is set to 1:2 in this example, every second signal will increment TMR0. Current status of the counter is displayed on PORTB LEDs.



Example of optocoupler on an input line



Macro: OPTOIN.ASM

```

;***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "p16f84.inc"

__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Structure of program memory *****

ORG    0x00           ;Reset vector
goto   Main

ORG    0x04           ;Interrupt vector
goto   Main           ;no interrupt routine

Main
;Main program
banksel TRISA
movlw  0xef           ;Initialization of port A
movwf  TRISA          ;TRISA <- 0xff
movlw  0x00           ;Initialization of port B
movwf  TRISB          ;TRISB <- 0x00
movlw  b'00110000'    ;RA4 -> TMRO, PS=1:2
banksel OPTION
movwf  OPTION_REG     ;Increment TMRO upon falling edge
banksel PORTB

clrf   PORTB          ;PORTB <- 0
clrf   TMRO           ;TMRO <- 0

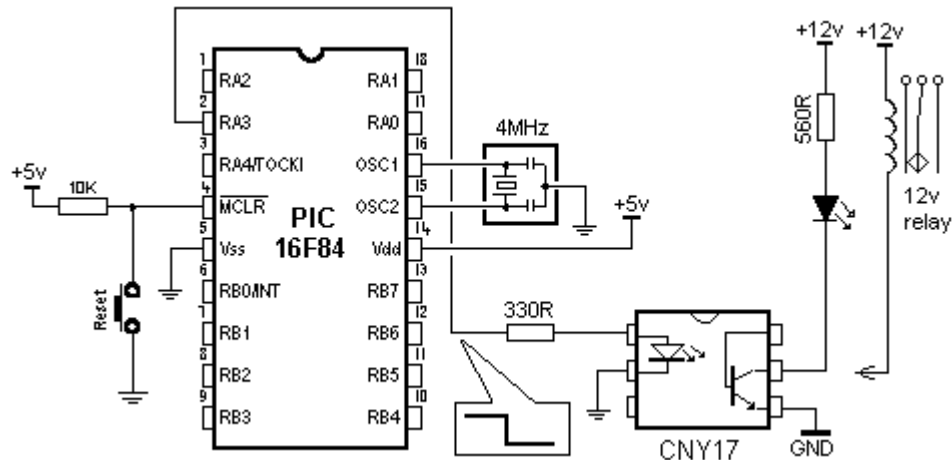
Loop   movf  TMRO,w     ;Send value of the counter
        movwf PORTB     ;to PORTB
        goto Loop       ;Remain at this line

End    ;End of program

```

7.4.2 Optocoupler on an output line

An Optocoupler can be also used to separate the output signals. If optocoupler LED is connected to microcontroller pin, logical zero on pin will activate optocoupler LED, thus activating the transistor. This will consequently switch on LED in the part of device working on 12V. Layout of this connection is shown below.

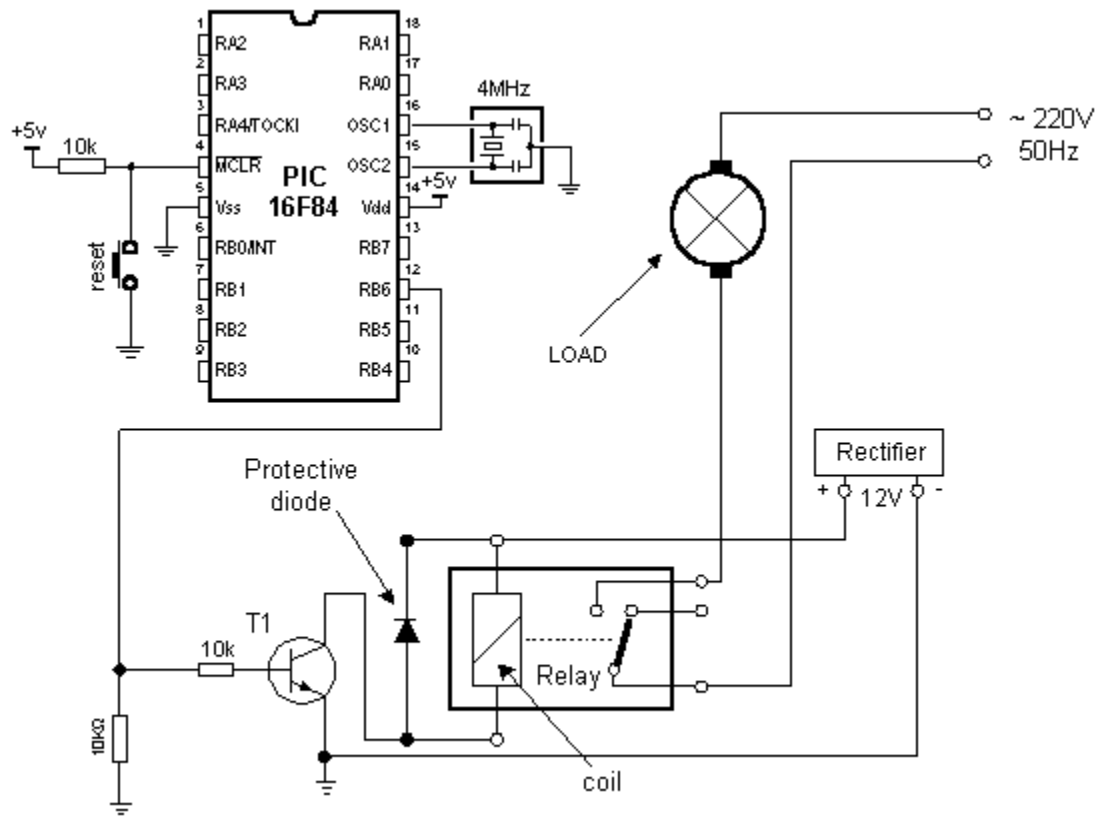


Example of optocoupler on output line

The program for this example is simple. By delivering a logical one to the third pin of port A, the transistor will be activated in the optocoupler, switching on the LED in the part of device working on 12V.

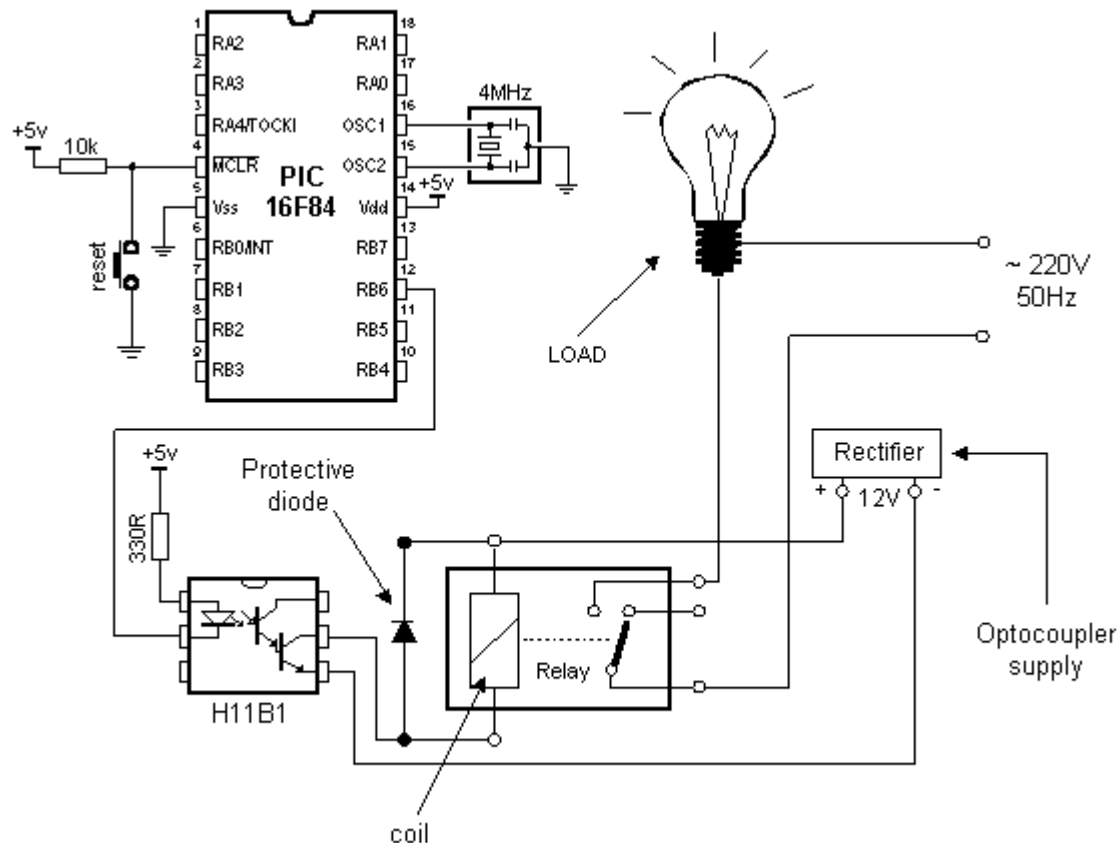
7.5 Relay

The relay is an electromechanical device, which transforms an electrical signal into mechanical movement. It consists of a coil of insulated wire on a metal core, and a metal armature with one or more contacts. When a supply voltage was delivered to the coil, current would flow and a magnetic field would be produced that moves the armature to close one set of contacts and/or open another set. When power is removed from the relay, the magnetic flux in the coil collapses and produces a fairly high voltage in the opposite direction. This voltage can damage the driver transistor and thus a reverse-biased diode is connected across the coil to "short-out" the spike when it occurs.



Connecting a relay to the microcontroller via transistor

Since microcontroller cannot provide sufficient supply for a relay coil (approx. 100+mA is required; microcontroller pin can provide up to 25mA), a transistor is used for adjustment purposes, its collector circuit containing the relay coil. When a logical one is delivered to transistor base, transistor activates the relay, which then, using its contacts, connects other elements in the circuit. Purpose of the resistor at the transistor base is to keep a logical zero on base to prevent the relay from activating by mistake. This ensures that only a clean logical one on RA3 activates the relay.



Connecting the optocoupler and relay to a microcontroller

A relay can also be activated via an optocoupler which at the same time amplifies the current related to the output of the microcontroller and provides a high degree of isolation. High current optocouplers usually contain a 'Darlington' output transistor to provide high output current.

Connecting via an optocoupler is recommended especially for microcontroller applications, where relays are used for starting high power load, such as motors or heaters, whose voltage instability can put the microcontroller at risk. In our example, when LED is activated on some of the output port pins, the relay is started. Below is the program needed to activate the relay, and includes some of the already discussed macros.



```
;***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "p16f84.inc"
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaring variables *****

Cblock 0x0C                ;RAM starting address
Hicnt
LOcnt
LOOPcnt
endc

;***** Declaring hardware *****

#define RELAY PORTA,3      ;Relay is located on the third pin
                           ;of port A

;***** Structure of program memory *****

ORG    0x00                ;Reset vector
goto   Main

ORG    0x04                ;Interrupt vector
goto   Main                ;no interrupt routine

#include "mikroel84.inc"    ;Macros
#include "button.inc"

Main                                ;Beginning of the program
    banksel TRISA
    movlw b'00010111'       ;Initializing port A
    movwf TRISA             ;TRISA <- 0x17
    movlw 0x00              ;Initializing port B
    movwf TRISB             ;TRISB <- 0x00
    banksel PORTB

    clrf PORTB              ;PORTB <- 0x00

Loop
    button PORTA, 0, 0, On   ;Button 1
    button PORTA, 1, 0, Off  ;Button 2

    goto Loop

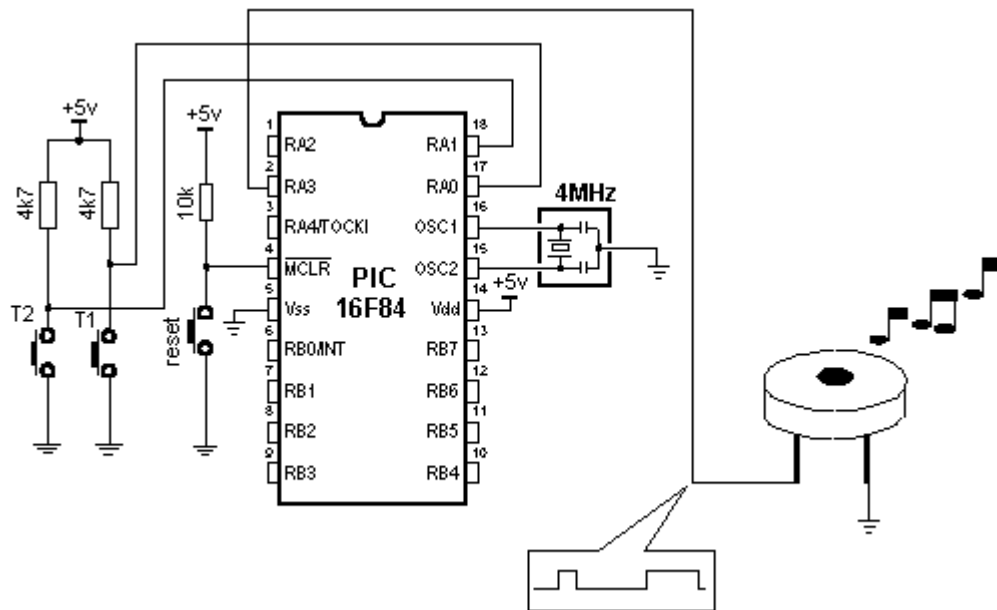
On
    bsf RELAY               ;Turn on relay
    goto Loop

Off
    bcf RELAY               ;Turn off relay
    goto Loop

End                          ;End of program
```

7.6 Generating sound

In microcontroller systems, beeper is used for indicating certain occurrences, such as push of a button or an error. To have the beeper started, it needs to be delivered a string in binary code - in this way, you can create sounds according to your needs. Connecting the beeper is fairly simple: one pin is connected to the mass, and the other to the microcontroller pin through a capacitor, as shown on the following image.



As with a button, you can employ a macro that will deliver a BEEP ROUTINE into a program when needed. Macro BEEP has two arguments:

BEEP **macro** freq , duration:

freq: frequency of the sound. The higher number produces higher frequency

duration: sound duration. Higher the number, longer the sound.

Example 1: BEEP 0xFF, 0x02

The output has the highest frequency and duration at 2 cycles per 65.3mS which gives 130.6 mS

Example2: BEEP 0x90, 0x05

The output has a frequency of 0x90 and duration of 5 cycles per 65.3mS. It is best to determine these macro parameters through experimentation and select the sound that best suits the application.

The following is the BEEP Macro listing:



Macro: BEEP.INC

```
;***** Declaring constants *****

        CONSTANT PRESCbeep = b'00000111' ; 65,3 ms per cycle

;***** Macros *****

BEEP    macro    freq,duration
        movlw    freq
        movwf    Beep_TEMP1
        movlw    duration
        call     BEEPsub
        endm

BEEPinit macro
        bcf      BEEPport
        bsf      STATUS,RPO
        bcf      BEEPtris
        bcf      STATUS,RPO
        endm

;***** Podprogrami *****

BEEPsub movwf Beep_TEMP2          ;set the sound duration
        clrf     TMRO             ;initialize the counter
        bcf      BEEPport
        bsf      STATUS,RPO
        bcf      BEEPport
        movlw    PRESCbeep        ;set prescaler for TMRO
        movwf    OPTION_REG      ;OPTION <- W
        bcf      STATUS,RPO
BEEPa   bcf      INTCON,TOIF      ;clear TMRO Overflow Flag
BEEPb   bsf      BEEPport
        call     B_Wait           ;duration of logical "1"
        bcf      BEEPport
        call     B_Wait           ;duration of logical "0"
        btfss    INTCON,TOIF      ;check TMRO Overflow Flag,
        goto     BEEPb            ;skip if set
        decfsz   Beep_TEMP2,1     ;Is Beep_TEMP2 = 0 ?
        goto     BEEPa            ;If not, jump back to BEEP
        RETURN

B_Wait  movfw    Beep_TEMP1
        movwf    Beep_TEMP3
B_Waita decfsz   Beep_TEMP3,1
        goto     B_Waita
        RETURN
```

The following example shows the use of a macro in a program. The program produces two melodies which are obtained by pressing T1 or T2. Some of the previously discussed macros are included in the program.



```
;***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "p16f84.inc"

__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaring variables *****

Cblock 0x0C                ;RAM strating address
PRESCwait
Beep_TEMP1                ;Belongs to macro "BEEP"
Beep_TEMP2
Beep_TEMP3
Hicnt                    ;Auxiliary variable for macro pausesms
Locnt
LOOPcnt
endc

;***** Declaring hardware *****

#define BEEPport PORTA,3    ;Port and pin beeper is located at
#define BEEPtris TRISA,3

;***** Structure of program memory *****

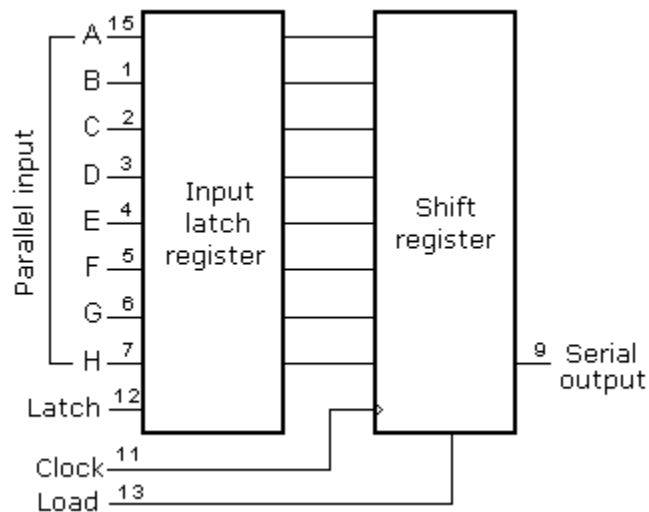
ORG 0x00                  ;Reset vector
goto Main
ORG 0x04                  ;Interrupt vector
goto Main                 ;no intertupt routine
include "mikroel84.inc"
include "button.inc"
include "beep.inc"
Main                      ;Beginning of the program
    banksel TRISA
    movlw b'00010111      ;Initializing port A
    movwf TRISA           ;TRISA <- 0x17
    banksel PORTE
    BEEPinit              ;Initializing Beeper
Loop
    button PORTA, 0,0, Play1 ;Button 1
    button PORTA, 1,0, Play2 ;Button 2
    goto Loop
Play1
    BEEP 0xFF, 0x02
    BEEP 0x90, 0x05
    BEEP 0xC0, 0x03
    BEEP 0xFF, 0x03        ;First tune
    goto Loop
Play2
    BEEP 0xbb, 0x02
    BEEP 0x87, 0x05
    BEEP 0xa2, 0x03
    BEEP 0x98, 0x03        ;Second tune
    goto Loop
End                        ;End of program
```

7.7 Shift registers

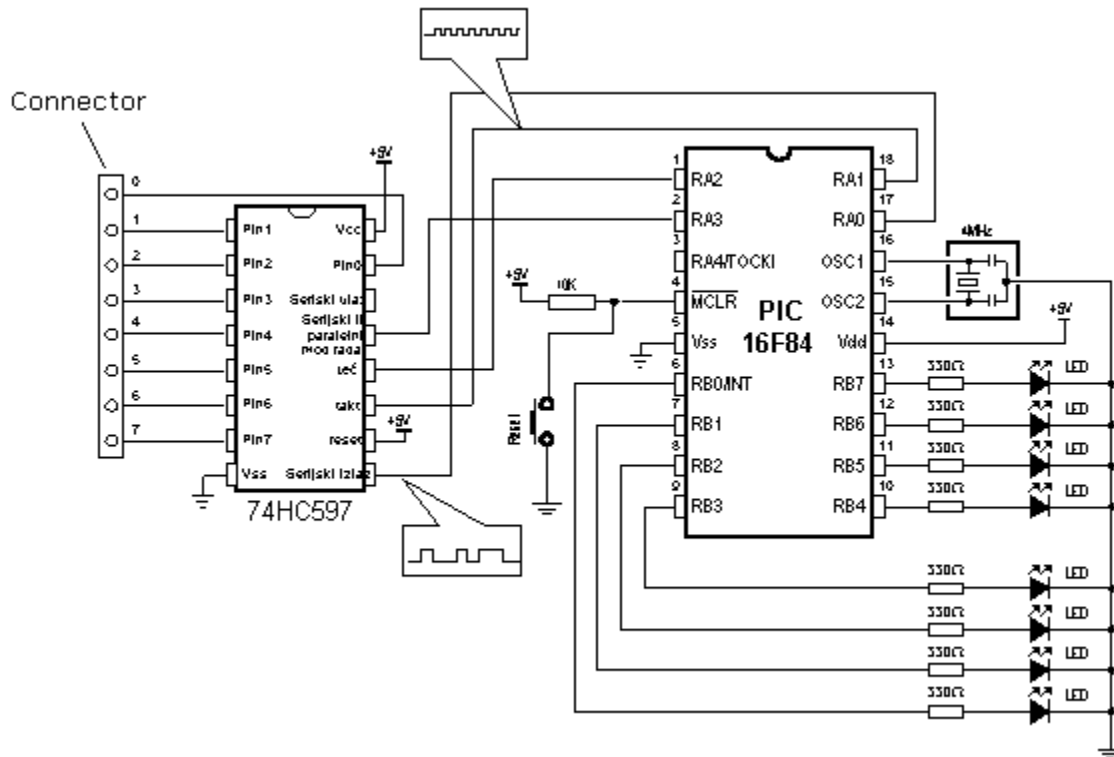
There are two types of shift registers: **input and output**. **Input shift registers** receive data in parallel, through 8 lines and then send it serially through two lines to a microcontroller. **Output shift registers** work in the opposite direction; they receive serial data and on a "latch" line signal, they turn it into parallel data. Shift registers are generally used to expand the number of input-output lines of a microcontroller. They are not so much in use any more though, because most modern microcontrollers have a large number of pins. However, their use with microcontrollers such as PIC16F84 is very important.

7.7.1 Input shift register 74HC597

Input shift registers transform parallel data into serial data and transfers it to a microcontroller. Their working is quite simple. There are four lines for the transfer of data: **Clock, Latch, Load and Data**. Data is first read from the input pins by an internal register through a 'latch' signal. Then, with a 'load' signal, data is transferred from the input latch register to the shift register, and from there it is serially transferred to a microcontroller via 'data' and 'clock' lines.



An outline of the connection of the shift register 74HC597 to a micro, is shown below.



In order to simplify the main program, a macro can be used for the input shift register. Macro HC597 has two parameters:

HC597 macro Var, Var1

Var variable where data from shift register input pins is transferred

Var1 loop counter

Example: HC597 data, counter

Data from the input pins of the shift register is stored in data variable. Timer/counter variable is used as a loop counter.

Macro listing:



Macro: HC597.INC

```
HC597 macro Var,Var1

    Local Loop          ;Local label

    movlw .8            ;Transfer eight bits
    movwf Var1          ;Initializing counter

    bsf Latch           ;Receive pin states into input latch
    nop
    bcf Latch

    bcf Load           ;Transfer the contents of input latch to
                        ;SHIFT reg.
    nop
    bsf Load

Loop    rlf Var,f       ;Rotate "Var" one place to the left

    btfss Data          ;Is Data line = "1" ?
    bcf Var,0           ;If not, clear bit '0' in Var
    btfsc Data          ;Is Data line = "0" ?
    bsf Var,0           ;If not, set bit '0'

    bsf Clock           ;Generate one clock
    nop
    bcf Clock

    decfsz Var1,f       ;Has 8 bits been received ?
    goto Loop          ;If not, repeat

endm
```

Example of how to use the HC597 macro is given in the following program. Program receives data from a parallel input of the shift register and moves it serially into the RX variable of the microcontroller. LEDs connected to port B will indicate the result of the data input.



```
***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "p16f84.inc"

__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

***** Declaring variables *****

Cblock 0x0C      ;RAM starting address
RX
CountSPI
endc

***** Declaring the hardware *****

#define Data PORTA,0      ;Any other I/O pin could be used
#define Clock PORTA,1
#define Latch PORTA,2
#define Load PORTA,3

***** Structure of program memory *****

ORG 0x00          ;Reset vector
goto Main

ORG 0x04          ;Interrupt vector
goto Main         ;no interrupt routine

#include "hc597.inc"

Main              ;Beginning of the program
    banksel TRISA
    movlw b'00010001' ;Initializing port A
    movwf TRISA      ;TRISA <- 0x11
    clrf TRISB        ;Pins of porta B are output
    banksel PORTA

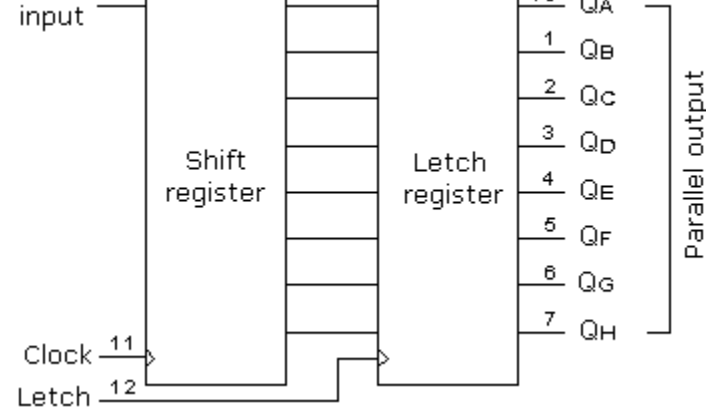
    clrf PORTA        ;PORTA <- 0x00
    bsf Load         ;Enable SHIFT register

Loop HC597 RX, CountSPI ;States of input pins of SHIFT reg.
    movf RX,W         ;are stored in variable RX
    movwf PORTB        ;Set the content of RX reg. to port B

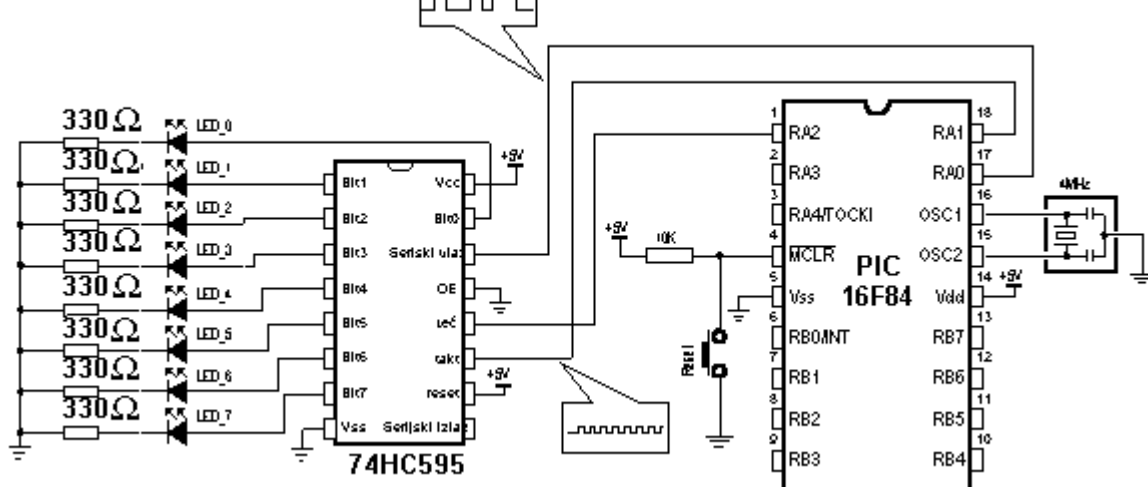
    goto Loop         ;Repeat the loop
End                   ;End of program
```

7.7.2 Output shift register

Serial 14 15



Downloaded from <http://ajph.org/> on November 10, 2015



Macro HC595 has two parameters:

HC595 macro Var, Var1

Var variable whose contents is transferred to outputs of shift register.

Example: HC595 Data, counter

Example: HC595 Data, counter

The data we want to transfer is stored in data variable, and counter variable is used as a loop counter.

Macro: HC595.INC

```

HC595  macro  Var,Var1

        Local  Loop           ;Local label
        movlw  .8              ;Transfer eight bits
        movwf  Var1            ;Initializing counter

Loop    rlf     Var,f           ;Rotate "Var" on place to the left

        bt fss  STATUS,C       ;Is carry = "1" ?
        bcf     Data          ;If not, set Data line to "0"
        bt fsc  STATUS,C       ;Is carry = "0" ?
        bsf     Data          ;If not, set Data line to "1"

        bsf     Clock          ;Generate one clock
        nop
        bcf     Clock

        decfsz  Var1,f         ;Has eight bits been sent ?
        goto   Loop           ;If not, repeat

        bsf     Latch          ;If all 8 bits have been sent, send the
        nop                          ;contents of SHIFT register to output latch
        bcf     Latch

        endm

```

An example of how to use the HC595 macro is given in the following program. Data from variable TX is serially transferred to shift register. LEDs connected to the parallel output of the shift register will indicate the state of the lines. In this example value 0xCB (1100 1011) is sent so that the seventh, sixth, third, first, and zero LEDs are illuminated.



```
;***** Declaring and configuring a microcontroller *****

PROCESSOR 16f84
#include "p16f84.inc"

__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaring variables *****

Cblock 0x0C                      ;RAM starting address
TX                               ;Belongs to the function "HC595"
CountSPI
endc

;***** Declaring the hardware *****

#define Data PORTA,0
#define Clock PORTA,1
#define Latch PORTA,2

;***** Structure of program memory *****

ORG 0x00                        ;Reset vector
goto Main
ORG 0x04                        ;Interrupt vector
goto Main                       ;no interrupt routine

include "hc595.inc"

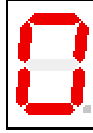
Main                             ;Beginning of the program
    banksel TRISA
    movlw b'00011000'           ;Initializing port A
    movwf TRISA                 ;TRISA <- 0x18
    banksel PORTA
    clrf PORTA                  ;PORTA <- 0x00

    movlw 0xcb                  ;Fill TX buffer
    movwf TX                    ;TX <- '11001011'
    HC595 TX, CountSPI
Loop    goto Loop               ;Remain at this line

End                             ;End of program
```

7.8 Seven-Segment Display (multiplexing)


The segments in a 7-segment display are arranged to form a single digit from 0 to F as shown in the animation:



We can display a multi-digit number by connecting additional displays. Even though LCD displays are more comfortable to work with, 7-segment displays are still standard in the industry. This is due to their temperature robustness, visibility and wide viewing angle. Segments are marked with non-capital letters: a, b, c, d, e, f, g and dp, where dp is the decimal point. The 8 LEDs inside each display can be arranged with a common cathode or common anode. With a common cathode display, the common cathode must be connected to the 0V rail and the LEDs are turned on with a logic one. Common anode displays must have the common anode connected to the +5V rail. The segments are turned on with a logic zero. The size of a display is measured in millimeters, the height of the digit itself (not the housing, but the digit!). Displays are available with a digit height of 7, 10, 13.5, 20, or 25 millimeters. They come in different colors, including: red, orange, and green.

The simplest way to drive a display is via a display driver. These are available for up to 4 displays. Alternatively displays can be driven by a microcontroller and if more than one display is required, the method of driving them is called "multiplexing."

The main difference between the two methods is the number of "drive lines." A special driver may need only a single "clock" line and the driver chip will access all the segments and increment the display. If a single display is to be driven from a microcontroller, 7 lines will be needed plus one for the decimal point. For each additional display, only one extra line is needed. To produce a 4, 5 or 6 digit display, all the 7-segment displays are connected in parallel. The common line (the common-cathode line) is taken out separately and this line is taken low for a short period of time to turn on the display. Each display is turned on at a rate above 100 times per second, and it will appear that all the displays are turned on at the same time. As each display is turned on, the appropriate information must be delivered to it so that it will give the correct reading. Up to 6 displays can be accessed like this without the brightness of each display being affected. Each display is turned on very hard for one-sixth the time and the POV (persistence of vision) of our eye thinks the display is turned on the whole time. Therefore, the program has to ensure the proper timing, else the unpleasant blinking of display will occur.



Program: DIGIT.INC

```
digbyte macro par0
    local Pon0
    local Exit1
    local Exit2
    local Pozitiv

    clrf Dig1
    clrf Dig2
    clrf Dig3
Pozitiv
    movf par0,w
    movwf Digtemp
    movlw .100
Pon0
    incf Dig1,f
    subwf Digtemp,f
    bt fsc STATUS,C
    goto Pon0
    decf Dig1,f
    addwf Digtemp,f
Exit1 movlw .10
    incf Dig2,f
    subwf Digtemp,f
    bt fsc STATUS,C
    goto Exit1
    decf Dig2,f
    addwf Digtemp,f
Exit2
    movf Digtemp,w
    movwf Dig3
    endm
```

The following example shows the use of the macro in a program. Program prints a specified 2-digit number on a 7seg display in multiplex mode.



```
PROCESSOR P16F84
#include "p16f84.inc"
__CONFIG __CP_OFF & __WDT_OFF & __PWRTE_ON & __XT_OSC

org 0x00
goto Main           ;Beginning of the program

org 0x04
goto ISR            ;Beginning of the interrupt routine

Cblock 0x0c
Dig1                ;Variable for storing the hundreds digit of number D
Dig2                ;Variable for storing the tens digit of number D
Dig3                ;Variable for storing the ones digit of number D
Digtemp
D                   ;D stores the number to be displayed
One                 ;Auxiliary variable for multiplex disp.
W_temp              ;Auxiliary variable
endc

include "mikroel84.inc"

Main
    banksel TRISA
    movlw b'11111100' ;RA0 and RA1 are output pins used for
                        ;multiplexing
    movwf TRISA
    clrf TRISB         ;Port B is output
    movlw b'10000100' ;Prescaler is 32 meaning that TMR 0 is
                        ;incremented every 32ms
    movwf OPTION_REG   ;Supposition that 4MHz oscillator is used
    banksel PORTA
    movlw .96           ;Starting value of TMRO is 96, thus
                        ;interrupt occurs every (255-97)*32us=5.088ms
    movwf TMRO
    movlw b'10100000' ;TMRO interrupt enabled
    movwf INTCON

    movlw .21           ;Print decimal value 21
    movwf D
    clrf One
    clrf PORTA          ;Turn off both displays at the start

Loop
    goto Loop           ;Main loop

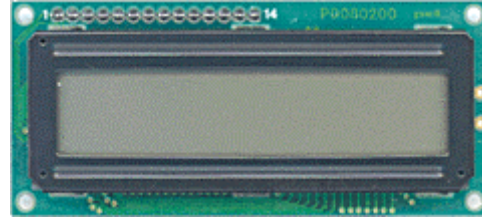
ISR
    movwf W_temp        ;Store the contents of W register
    movlw .96
    movwf TMRO          ;Initialize TMRO to have the next interrupt in
                        ;approximately 5ms
    bcf INTCON,TOIF     ;Clear flag overflow TMRO to be able
                        ;to react to the next interrupt
    bcf PORTA,0         ;Turn off both displays
    bcf PORTA,1
    movf One,f
    btfsc STATUS,Z      ;Which display should be on?
    goto Msdon

Lsdon
    ;If MSD was previously on,
    bcf One,0           ;One stores that LSD is on

    movlw high Bcdto7seg;Prior to jump to lookup table, PCLATH
    movwf PCLATH        ;register should be initialized with the higher
                        ;byte of the address lookup (HIGH lookup),
```

7.9 LCD Display

More microcontroller devices are using 'smart LCD' displays to output visual information. The following discussion covers the connection of a **Hitachi LCD display** to a PIC microcontroller. LCD displays designed around Hitachi's LCD HD44780 module, are inexpensive, easy to use, and it is even possible to produce a readout using the 8 x 80 pixels of the display. Hitachi LCD displays have a standard ASCII set of characters plus Japanese, Greek and mathematical symbols.



A 16x2 line Hitachi HD44780 display

For a 8-bit data bus, the display requires a +5V supply plus 11 I/O lines. For a 4-bit data bus it only requires the supply lines plus seven extra lines. When the LCD display is not enabled, data lines are tri-state which means they are in a state of high impedance (as though they are disconnected) and this means they do not interfere with the operation of the microcontroller when the display is not being addressed.

The LCD also requires 3 "control" lines from the microcontroller.

Enable (E)	This line allows access to the display through R/W and RS lines. When this line is low, the LCD is disabled and ignores signals from R/W and RS. When (E) line is high, the LCD checks the state of the two control lines and responds accordingly.
Read/Write (R/W)	This line determines the direction of data between the LCD and microcontroller. When it is low, data is written to the LCD. When it is high, data is read from the LCD.
Register select (RS)	With the help of this line, the LCD interprets the type of data on data lines. When it is low, an instruction is being written to the LCD. When it is high, a character is being written to the LCD.

Logic status on control lines:

E 0 Access to LCD disabled
 1 Access to LCD enabled

R/W 0 Writing data to LCD
 1 Reading data from LCD

RS 0 Instruction
 1 Character









Writing data to the LCD is done in several steps:

Set R/W bit to low
Set RS bit to logic 0 or 1 (instruction or character)
Set data to data lines (if it is writing)

Set E line to high
Set E line to low
Read data from data lines (if it is reading)

Reading data from the LCD is done in the same way, but control line R/W has to be high. When we send a high to the LCD, it will reset and wait for instructions. Typical instructions sent to LCD display after a reset are: turning on a display, turning on a cursor and writing characters from left to right. When the LCD is initialized, it is ready to continue receiving data or instructions. If it receives a character, it will write it on the display and move the cursor one space to the right. The Cursor marks the next location where a character will be written. When we want to write a string of characters, first we need to set up the starting address, and then send one character at a time. Characters that can be shown on the display are stored in data display (DD) RAM. The size of DDRAM is 80 bytes.

The LCD display also possesses 64 bytes of Character-Generator (CG) RAM. This memory is used for characters defined by the user. Data in CG RAM is represented as an 8-bit character bit-map. Each character takes up 8 bytes of CG RAM, so the total number of characters, which the user can define is eight. In order to read in the character bit-map to the LCD display, we must first set the CG RAM address to starting point (usually 0), and then write data to the display. The definition of a 'special' character is given in the picture.

CG RAM address	Bit map	Data
0000		01010
0001		00100
0010		01110
0011		10001
0100		10000
0101		10001
0110		01110
0111		00000

Before we access DD RAM after defining a special character, the program must set the DD RAM address. Writing and reading data from any LCD memory is done from the last address which was set up using set-address instruction. Once the address of DD RAM is set, a new written character will be displayed at the appropriate place on the screen. Until now we discussed the operation of writing and reading to an LCD as if it were an ordinary memory. But this is not so. The LCD controller needs 40 to 120 microseconds (uS) for writing and reading. Other operations can take up to 5 mS. During that time, the microcontroller can not access the LCD, so a program needs to know when the LCD is busy. We can solve this in two ways.

Set DD RAM address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A	A	A	A	A	A	A

Set CG RAM address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	A	A	A	A	A	A

Write in data to RAM

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	D	D	D	D	D	D	D	D

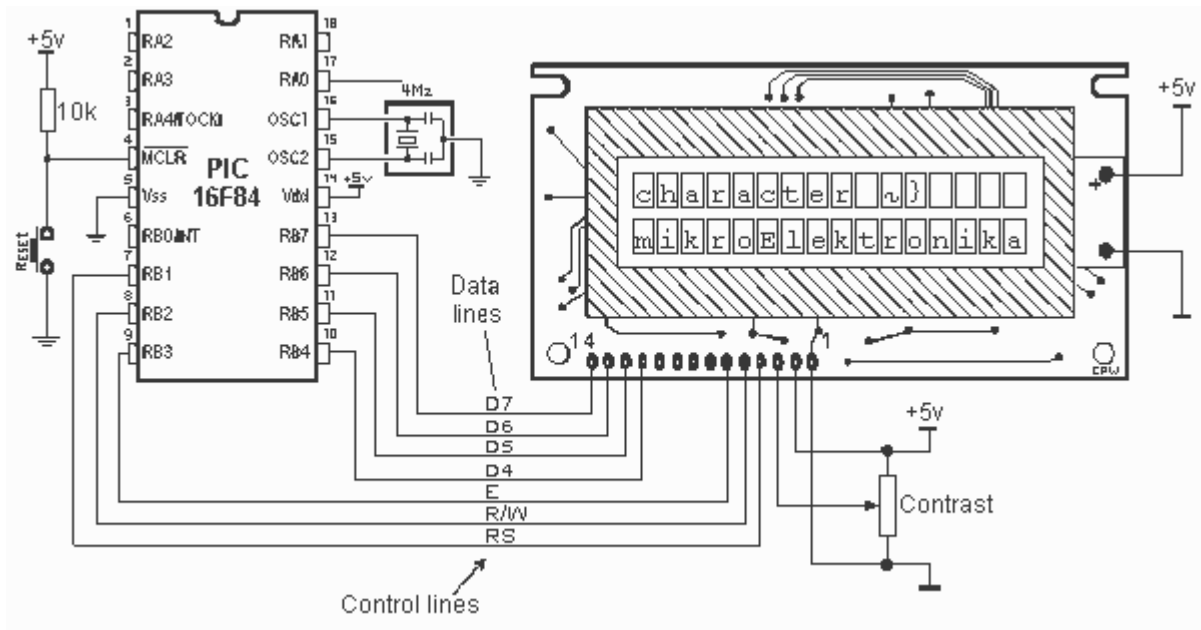
Read data from RAM

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D	D	D	D	D	D	D	D

A=address**D=data**

One way is to check the BUSY bit found on data line D7. This is not the best method because LCD's can get stuck, and program will then stay forever in a loop checking the BUSY bit. The other way is to introduce a delay in the program. The delay has to be long enough for the LCD to finish the operation in process. Instructions for writing to and reading from an LCD memory are shown in the previous table.

At the beginning we mentioned that we needed 11 I/O lines to communicate with an LCD. However, we can communicate with an LCD through a 4-bit data bus. Thus we can reduce the total number of communication lines to seven. The wiring for connection via a 4-bit data bus is shown in the diagram below. In this example we use an LCD display with 2x16 characters, labeled LM16X212 by Japanese maker SHARP. The message 'character' is written in the first row: and two special characters '~' and '}' are displayed. In the second row we have produced the word 'mikroElektronika'.



Connecting an LCD display to a microcontroller

File **lcd.inc** contains a group of macros for use when working with LCD displays.



Macro: LCD.INC

```
;*****
;
;
;*****
    CONSTANT  FUNCTSET8 = b'00110000' ; 8-bit mode, 2 lines
    CONSTANT  FUNCTSET4 = b'00100000' ; 4-bit mode, 2 lines
    CONSTANT  DDZERO    = b'10000000' ; Write 0 to DDRAM

    CONSTANT  LCD2L     = b'00101000'
    CONSTANT  LCDCONT   = b'00001100'

    CONSTANT  LCDSH     = b'00101000'

;Commands for working with LCD display
    CONSTANT  LCDCLR    = b'00000001' ;clear display, cursor home
    CONSTANT  LCDCH     = b'00000010' ;cursor home
    CONSTANT  LCDCL     = b'00000100' ;move cursor to the left
    CONSTANT  LCDCR     = b'00000110' ;move cursor to the right
    CONSTANT  LCDSL     = b'00011000' ;move the content of display
                                   ;to the left
    CONSTANT  LCDSR     = b'00011100' ;move the content of display
                                   ;to the right
    CONSTANT  LCDL1     = b'10000000' ;select line 1
    CONSTANT  LCDL2     = b'11000000' ;select line 2

;*****
;   before accessing the LCD, macro lcdinit has to
;   be initialized
;*****

lcdinit macro

    bankl
    clrf    LCDdsport          ;LCDdsport where LCD is an output
    bank0

    call    Delaylms           ;4 ms pause
    call    Delaylms
    call    Delaylms
    call    Delaylms
    movlw   FUNCTSET8          ;Begin initialization in
    call    SendW               ;8-bit mode
    call    Delaylms           ;2 ms pause
    call    Delaylms
    movlw   DDZERO             ;Write 0 to DDRAM
    call    SendW

    movlw   FUNCTSET4          ;From this line, LCD works in 4-bit mode
    call    SendW
;Commands for initializing LCD
    lcdcmd  LCD2L              ;lcd has 2 lines
    lcdcmd  LCDCONT            ;
    lcdcmd  LCDSH              ;
    lcdcmd  LCDCLR            ;Clear LCD
endm

;*****
;   lcdcmd sends the command to LCD (see the table above)
;   lcdclr has the same meaning as lcdcmd 0x01
;*****
lcdcmd macro    LCDcommand          ;Send a command to lcd
    movlw    LCDcommand
```

Using the macro for LCD support

lcdinit Macro used to initialize port connected to LCD. LCD is configured to work in 4-bit mode.

Example: lcdinit

lcdtext lcdtext prints the text of up to 16 characters, which is specified as a macro parameter. First parameter selects the line in which to start printing. If select is zero, text is printed from the current cursor position.

Example: lcdtext 1, "mikroelektronika"

lcdtext 1, "Temperature1" ;Print the text starting from line 1, character 1

lcdtext 2, "temp=" ;Print the text starting from line 2, character 1

lcdtext 0, " C" ;Print C in the rest of the line 2

lcdcmd Sends command instructions

LCDCLR = ;Clear display, cursor home
b'00000001'

LCDCH = ;Cursor home
b'00000010'

LCDCL = ;Move the cursor to the left
b'00000100'

LCDCR = ;Move the cursor to the right
b'00000110'

LCDSL = ;Move the content of display to the left
b'00011000'

LCDSR = ;Move the content of display to the right
b'00011100'

LCDL1 = ;Select line 1
b'10000000'

LCDL2 = ;Select line 2
b'11000000'

Example: lcdcmd LCDCH

lcdbyte Prints one byte variable and omits leading zeros

Example: lcdbyte Temperature

When working with a microcontroller the numbers are presented in a binary form. As such, they cannot be displayed on a display. That's why it is necessary to change the numbers from a binary system into a decimal system so they can be easily understood. For printing the variables lcdbyte and lcdword we have used the macros *digbyte* and *digword* which convert the numbers from binary system into a decimal system and print the result on LCD. Main program has the purpose of demonstrating use of LCD display. At the start it's necessary to declare variables LCDbuf, LCDtemp, Digtemp, Dig1, Dig2, and Dig3 used by the macros for LCD support. It is also necessary to state the port of microcontroller that LCD is connected to. Program initializes the LCD and demonstrates printing text and 8-bit variable temp.



```
;***** Declaring and configuring a microcontroller *****

    PROCESSOR 16f84
    #include "p16f84.inc"

    __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Declaring variables *****

Cblock 0x0C      ;RAM starting address
Hicnt           ;Belongs to macro "pausems"
LOcnt
LOOPcnt

LCDbuf          ;Belongs to functions "LCDxxx"
LCDtemp

Digtemp         ;Belongs to macro "digbyte"
Dig1
Dig2
Dig3

temp
endc

ORG 0x00        ;Reset vector
goto Main

ORG 0x04        ;Intertupt vector
goto Main       ;no interrupt routine

include "mikroel84.inc"
include "lcd.inc"

LCDdsport equ PORTB ;LCD is on port B(4 data lines on RB4-RB7)
RS equ 1        ;RS line RB1
RW equ 2        ;RW line RB2
EN equ 3        ;EN line RB3

Main
    movlw .23
    movwf temp           ;Put any value to variable temp
                           ;for printing on LCD

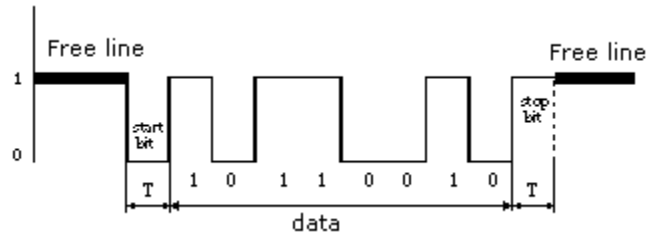
    lcdinit              ;Incitializing LCD

Loop
    lcdcmd 0x01           ;Clear LCD
    lcdtext 1,"mikroelektronika" ;Print text from line 1, char 1
    lcdtext 2,"Proba LCD"  ;Print text from line 2, char 1
    pausems .2000         ;2 sec pause
    lcdcmd 0x01           ;Clear LCD
    lcdtext 1,"Temperatura" ;Print text from line 1, char 1
    lcdtext 2,"temp="      ;Print text from line 2, char 1
    lcdbyte temp          ;Print decimal value of variable
    lcdtext 0,"C"         ;Print text from the current
    pausems .2000         ;cursor position

    goto Loop
```

7.10 Serial Communication

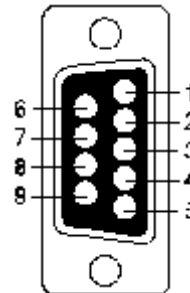
SCI is an abbreviation for Serial Communication Interface and, as a special subsystem, it exists on most microcontrollers. When it is not available, as is the case with PIC16F84, it can be created in software.



As with hardware communication, we use standard NRZ (Non Return to Zero) format also known as 8 (9)-N-1, or 8 or 9 data bits, without parity bit and with one stop bit. **Free line** is defined as the status of **logic one**. Start of transmission - **Start Bit**, has the status of **logic zero**. The data bits follow the start bit (the first bit is the low significant bit), and after the bits we place the **Stop Bit** of **logic one**. The duration of the stop bit 'T' depends on the transmission rate and is adjusted according to the needs of the transmission. For the transmission speed of 9600 baud, T is 104 μ S.

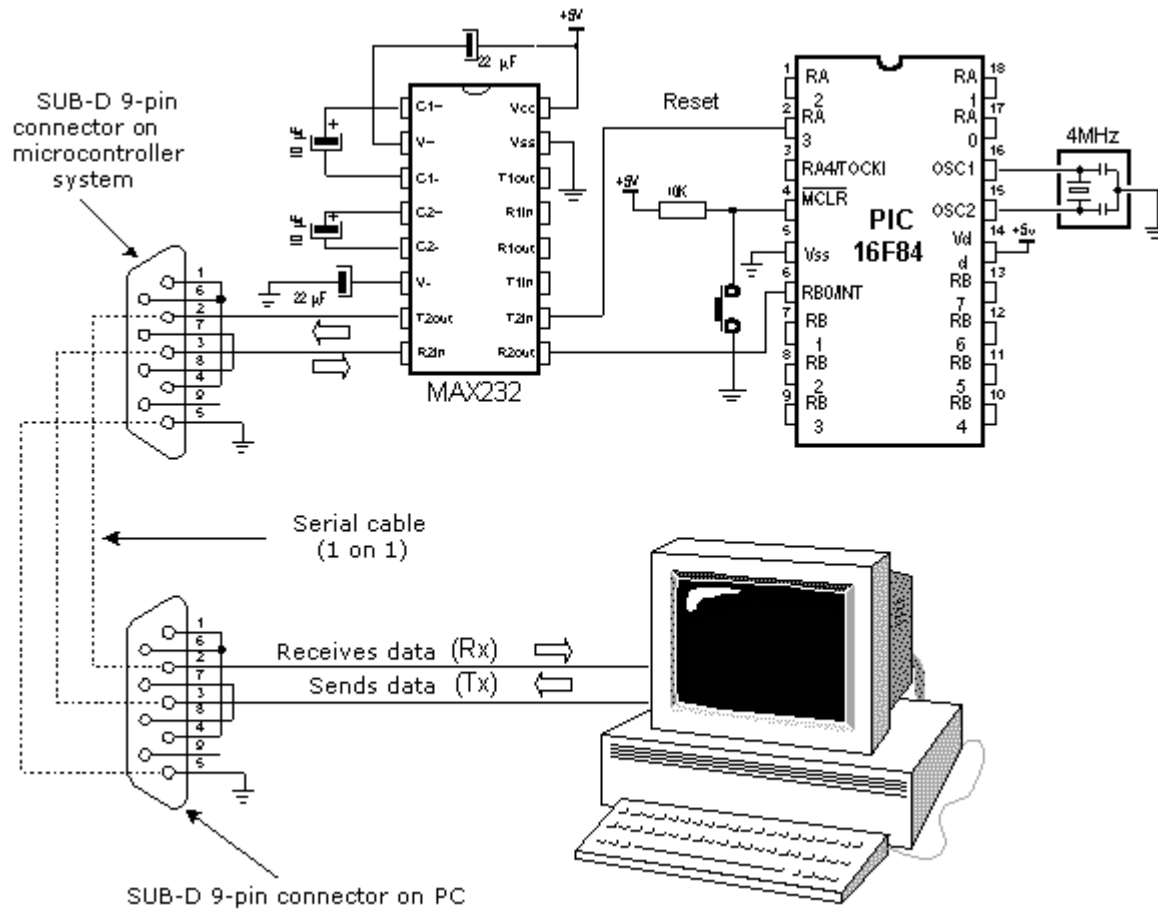
Pin designations on RS232 connector

- | | |
|--------|-----------------------|
| 1. CD | (Carrier Detect) |
| 2. RXD | (Receive Data) |
| 3. TXD | (Transmit Data) |
| 4. DTR | (Data terminal Ready) |
| 5. GND | (Ground) |
| 6. DSR | (Data Set Ready) |
| 7. RTS | (Request To Send) |
| 8. CTS | (Clear To Send) |
| 9. RI | (Ring Indicator) |



In order to connect a microcontroller to a serial port on a PC computer, we need to adjust the level of the signals so communicating can take place. The signal level on a PC is -10V for logic zero, and +10V for logic one. Since the signal level on the microcontroller is +5V for logic one, and 0V for logic zero, we need an intermediary stage that will convert the levels. One chip specially designed for this task is MAX232. This chip receives signals from -10 to +10V and converts them into 0 and 5V.

The circuit for this interface is shown in the diagram below:



Connecting a microcontroller to a PC via a MAX232 line interface chip

File RS232.inc contains a group of macros used for serial communication.



```
***** Declaring constants *****
CONSTANT LF = .10      ;Line Feed
CONSTANT CR = .13      ;Carriage Return
CONSTANT TAB = .9      ;Tabulator
CONSTANT BS = .8       ;Backspace

***** Macros *****

rs232init macro
    bsf    STATUS,RP0      ;bank 1
    bcf    TXtris          ;TX pin is output
    bcf    STATUS,RP0      ;bank 0
    bsf    TXport          ;Initial state on TX : logical "1"
    endm

***** Subprograms *****

Sendw                                ;sendw sends value of W register
    movwf  TXD              ;TX Data-register
    bcf    TXport          ;start bit
    call   Bitdelay

    local  i=0
    while  i<8              ;.. 8 times starting from LSB bit
        bt fsc  TXD,i
        call   SEND1        ;If i bit equals 1, call SEND1
        bt fss  TXD,i
        call   SEND0        ;If i bit equals 0, call SEND0
        i=i+1
    endw

    bsf    TXport          ;stop bit
    call   Bitdelay
    call   Bitdelay        ;Re-Synchronization
    return

SEND1
    bsf    TXport          ;Set line to 1
    call   Bitdelay        ;Delay for duration of sending 1 bit
    return

SEND0
    bcf    TXport          ;Set line to 0
    call   Bitdelay        ;Delay for duration of sending 1 bit
    return

Bitdelay                            ;104us pause necessary for sending 1 bit
    movlw  0x1E            ;Total delay between 2 bits
    movwf  Rs232temp       ;9600 baud ==> 104 us

Waitloop
    decfsz Rs232temp,f
    goto   Waitloop
    return

rs232text macro text                ;This macro prints the text from the
                                      ;parameter, of up to 16 characters

    local  Message
    local  Start
    local  Exit
    local  i=0
    goto   Start
    Poruka  dt  text
            dt  0                ;0 marks the end of the text
    Start
```

Using the macro for serial communication:

rs232init Example:	Macro for initializing the pin for transmitting data (TX-pin). RS232init
Sendw Example:	Sending ASCII value of data found in W register. movlw 't' call Sendw
rs232text Example:	Sending ASCII value of a specified text rs232 "mikroelektronika"
rs232byte Example:	Sending ASCII value of decimal digits of 8-bit variable movlw .123 movwf TXdata rs232byte TXdata ;Send '1', '2', '3'

When rs232.inc file is used, it is necessary to declare variables Rstemp and TXD at the beginning of the main program.

Example:

As a demonstration of a serial communication, we have an example which sends text and the content of variable *cnt*. Text is sent via macro rs232text, while variable *cnt* is sent via macro rs232byte. This operation repeats itself after 500ms, with incrementing *cnt* in the process. This example can be easily modified to have button(s) for sending specified data.