

BASIC for PIC microcontrollers



The complete BASIC programming language manual for PIC microcontrollers!

Nebojsa Matic

Author

January / 2003

CHAPTER I THE FUNDAMENTS OF PIC BASIC

CHAPTER II BASIC ELEMENTS OF PIC BASIC LANGUAGE

CHAPTER III OPERATORS

CHAPTER IV INSTRUCTIONS

CHAPTER V SAMPLE PROGRAMS FOR SUBSYSTEMS WITHIN THE MICROS

CHAPTER VI SAMPLES WITH PIC16F84 MICROCONTROLLER

CHAPTER VII SAMPLES WITH PIC16F877 MICROCONTROLLER

APPENDIX A MPLAB

APPENDIX B MicroCode studio

In this book you can find:

- Practical connection samples for:
- Temperature sensors, AD and DA converters LCD and LED displays, relays. Every example is commented in details with detailed connection scheme
- Program writing
- Learn how to write your own program, correct mistakes and use it to start a microcontroller.
- Instruction Set
- Every instruction is explained in detail with the example how to use it.
- MicroCode studio
- How to install it, how to use it
- MPLAB program package
- How to install it, how to start the first program, how to connect BASIC and MPLAB etc

Preface

In order to simplify things and crash some prejudices, I will allow myself to give you some advice before reading this book. You should start reading it from the chapter that interests you the most, in order you find suitable. As the time goes by, read the parts you may need at that exact moment. If something starts functioning without you knowing exactly how, it shouldn't bother you too much. Anyway, it is better that your program works than that it doesn't. Always stick to the practical side of life. It is much better for the program to be finished on time, to be reliable and, of course, to be paid for it as well as possible.

In other words, it doesn't matter if the exact manner in which the electrons move within the PN junctions your microcontroller is composed of escapes your knowledge. You are not supposed to know the whole history of electronics in order to assure the income for you or your family. Do not expect that you will find everything you need in one single book. The information are dispersed literally everywhere around you, so it is necessary to collect them diligently and sort them out carefully. If you do so, success is inevitable. With all my hopes of having done something worthy investing your time in.

Yours **Nebojsa Matic**

Chapter 1

THE FUNDAMENTS OF PIC BASIC

[Introduction](#)

- [1.1 BASIC for PIC microcontrollers](#)
- [1.2 PIC microcontrollers](#)
- [1.3 First program written in PIC BASIC](#)
- [1.4 Writing and compilation of a BASIC program](#)
- [1.5 Loading a program into the microcontroller memory](#)
- [1.6 Running your program](#)
- [1.7 Problem with starting your program \(what if it doesn't work\)](#)

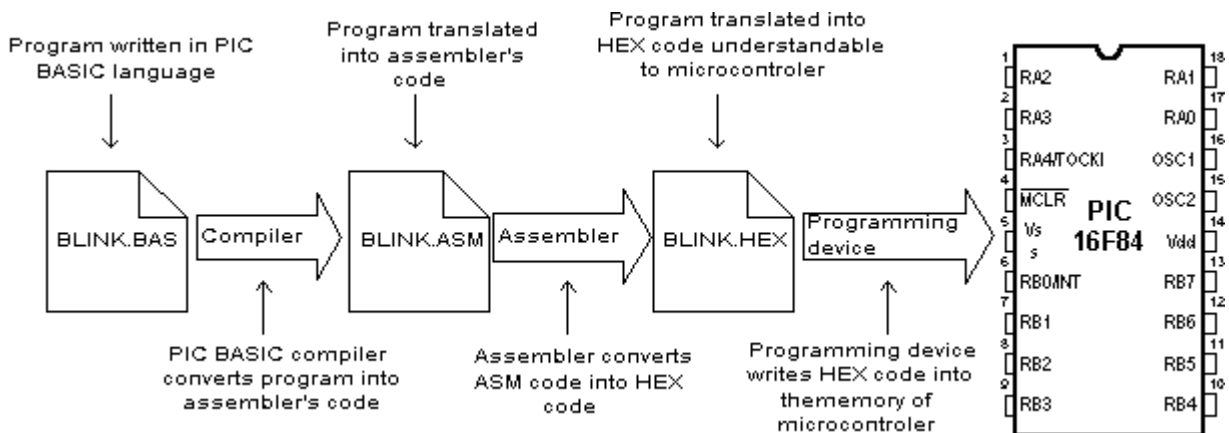
Introduction

Simplicity and ease, which the higher programming languages bring for program writing as well as broader application of the microcontrollers, was enough to incite some companies as Microengineering to embark on the development of BASIC programming language. What did we thereby get? Before all, the time of writing was shortened by employment of prepared functions that BASIC brings in (whose programming in assembler would have taken the biggest portion of time). In this way, the programmer can concentrate on solving the essential task without losing his time on writing the code for LCD display. To avoid any confusion in the further text, it is necessary to clarify three terms one encounters very often.

Programming language is understood as a set of commands and rules according to which we write the program and therefore we distinguish various programming languages such as BASIC, C, PASCAL etc. On the BASIC programming language the existing literature is pretty extensive so that most of the attention in this book will be dedicated to the part concretely dealing with the programming of microcontrollers.

Program consists of sequence of commands of language that our microcontroller executes one after another. The structure of BASIC program is explained with more detailed in the second chapter.

BASIC compiler is the program run on PC and it's task is to translate the original BASIC code into the language of 0 and 1 understandable to the microcontroller. The process of translation of a BASIC program into an executive HEX code is shown on the image below. The program written in PIC BASIC and registered as a file **Program.bas** is converted into an assembler code (**Program.asm**). So obtained assembler code is further translated into executive HEX code which is written to the microcontroller memory by a programmer. (programmer is a device used for transferring HEX files from PC to the microcontroller memory)



1.1 BASIC for PIC microcontrollers

As a programming language, BASIC is since long time ago known to the PC users to be the easiest and the most widespread one. Nowadays this reputation is more and more being transferred onto the world of microcontrollers. PIC

BASIC enables quicker and relatively easier program writing for PIC microcontrollers in comparison with the *Microchip's* assembling language MPASM. During the program writing, the programmer encounters always the same problems such as serial way of sending messages, writing of a variable on LCD display, generating of PWM signals etc. All for the purpose of facilitating programming, PIC BASIC contains its built-in commands intended for solving of the problems often encountered in praxis. As far as the speed of execution and the size of the program are concern, MPASM is in small advantage in respect with PIC BASIC (therefore exists the possibility of combining PIC BASIC and assembler). Usually, the part of the program in which the same commands are executed many times or time of the execution critical, are written in assembler. Modern microcontrollers such as PIC execute the instructions in a single cycle lasting for 4 tact of the oscillator. If the oscillator of the microcontroller is 4MHz, (one single tact lasts 250nS), then one assembler instruction requires $250\text{nS} \times 4 = 1\mu\text{S}$ for the execution. Each BASIC command is in effect the sequence of the assembler instructions and the exact time necessary for its execution may be obtained by simply summing up the times necessary for the execution of assembler instructions within one single BASIC command.

1.2 PIC microcontrollers

The creation of PIC BASIC followed the great success of Basic stamp (small plate with PIC16F84 and serial eeprom that compose the whole microcontroller system) as its modification. PIC BASIC enables the programs written for the original Basic stamp to be translated for the direct execution on the PIC16xxx, PIC17Cxxx and PIC18Cxxx members of the microcontrollers family. By means of PIC BASIC it is possible to write programs for the PIC microcontrollers of the following families PIC12C67x, PIC14C000, PIC16C55x, PIC16C6x, PIC16C7x, PIC16x84, PIC16C9xx, PIC16F62x, PIC16C87x, PIC17Cxxx and PIC 18Cxxx. On the contrary, the programs written in PIC BASIC language cannot be run on the microcontrollers possessing the hardware stack in two levels as is for example the case of PIC16C5x family (that implies that by using the *CALL* command any subroutine can be called not more than two times in a row).

For the controllers that are not able to work with PIC BASIC there is an adequate substitution. For example, instead of PIC16C54 or 58, we can use pin compatible chips PIC16C554, 558, 620 and 622 also operating with PIC BASIC without any difference in price.

Currently, the best choice for application development, using PIC BASIC are microcontrollers from the family : PIC16F87x, PIC16F62X and of course the famous PIC16F84. With this family of PIC microcontrollers, program memory is created using FLASH technology which provides fast erasing and reprogramming, thus allowing faster debugging. By a single mouse click in the programming software, microcontroller program can be instantly erased and then reloaded without removing chip from device. Also, program loaded in FLASH memory can be stored after power supply has been turned off. The older PIC microcontroller series (12C67x, 14C000, 16C55x, 16C6xx, 16C7xx and 16C92x) have program memory created using EPROM/ROM technology, so they can either be programmed only once (OTP version with ROM memory) or have glass window (JW version with EPROM memory), which allows erasing by few minutes exposure to UV light. OTP versions are usually cheaper and are used for manufacturing large series of products. Besides FLASH memory, microcontrollers of PIC16F87x and PIC16F84 series also contain 64-256 bytes of internal EEPROM memory, which can be used for storing program data and other parameters when power is off. PIC BASIC has built-in READ and WRITE instructions that can be used for loading and saving data to EEPROM. In order to have complete information about specific microcontroller in the application, you should get the appropriate Data Sheet or Microchip CD-ROM.



The program examples worked out throughout this book are mostly to be run on the microcontrollers PIC16F84 or PIC16F877, but could be, with small or almost no corrections, run on any other PIC microcontroller.

1.3 First program written in PIC BASIC

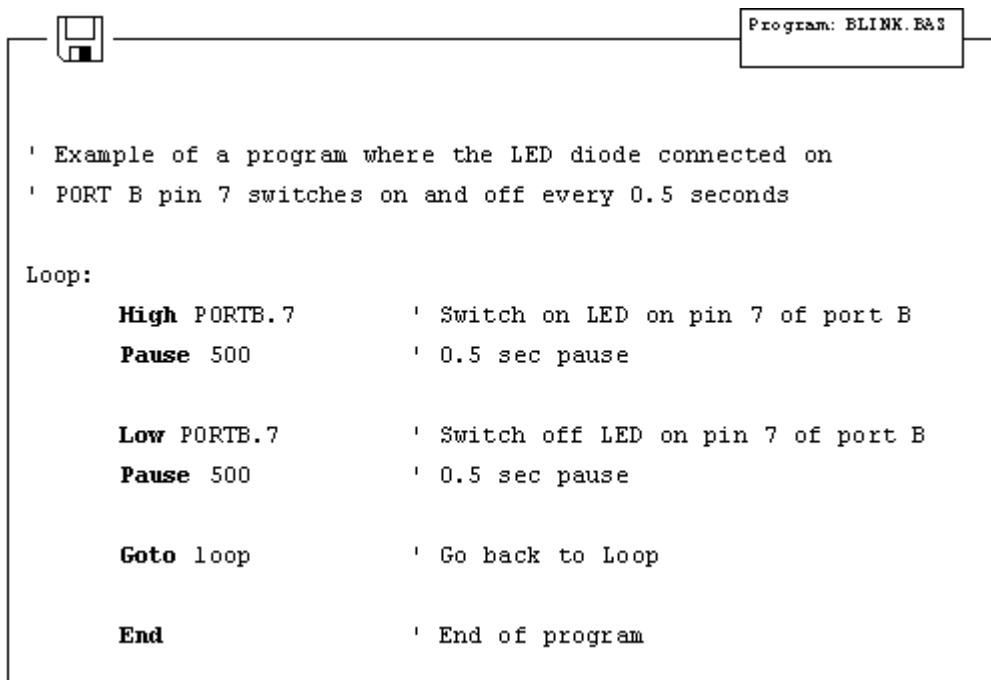
In order to start program writing and application development in BASIC programming language, it is necessary to have at least one text editor, PIC BASIC compiler and according to someone's wish - a system in development on which the program is supposed to be checked. For writing BASIC program code, any text editor that can save the program file as pure ASCII text (without special symbols for formatting) can be used. For this purpose editors like Notepad or WordPad are also good. Even better solution than the use of any classical text editor is the use of some of the editors specially devised for program code writing such as *Microchip's* MPLAB or *Mecanique's* Micro CODE

STUDIO.

The advantage of these program packages is that they take care of the code syntax, free memory and provide more comfortable environment when writing a program (appendices A and B describe MPLAB and MicroCODE STUDIO editors).

1.4 Writing and compilation of a BASIC program

The first step is the writing of a program code in some of enumerated text editors. Every written code must be saved on a single file with the ending .BAS exclusively as ASCII text. An example of one simple BASIC program - BLINK.BAS is given.



```
Program: BLINK.BAS

' Example of a program where the LED diode connected on
' PORT B pin 7 switches on and off every 0.5 seconds

Loop:
  High PORTB.7      ' Switch on LED on pin 7 of port B
  Pause 500         ' 0.5 sec pause

  Low PORTB.7      ' Switch off LED on pin 7 of port B
  Pause 500         ' 0.5 sec pause

  Goto loop        ' Go back to Loop

End                ' End of program
```

When the original BASIC program is finished and saved as a single file with .BAS ending it is necessary to start PIC BASIC compiler. The compiling procedure takes place in two consecutive steps.

Step 1. In the first step compiler will convert BAS file in assembler s code and save it as BLINK.ASM file.

Step 2. In the second step compiler automatically calls assembler, which converts ASM - type file into an executable HEX code ready for reading into the programming memory of a microcontroller.

The transition between first and second step is for a user - programmer an invisible one, as everything happens completely automatically and is thereby wrapped up as an indivisible process. In case of a syntax error of a program code, the compilation will not be successful and HEX file will not be created at all. Errors must be then corrected in original BAS file and repeat the whole compilation process. The best tactics is to write and test small parts of the program, than write one gigantic of 1000 lines or more and only then embark on error finding.

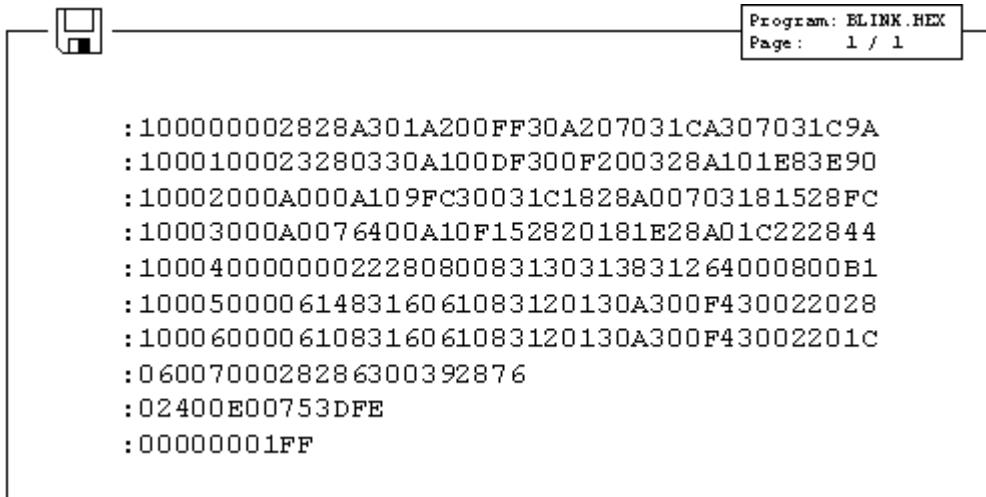
1.5 Loading a program into the microcontroller memory

As a result of a successful compilation of a PIC BASIC program the following files will be created.

- BLINK.ASM - assembler file
- BLINK.LST - program listing
- BLINK.MAC - file with macros

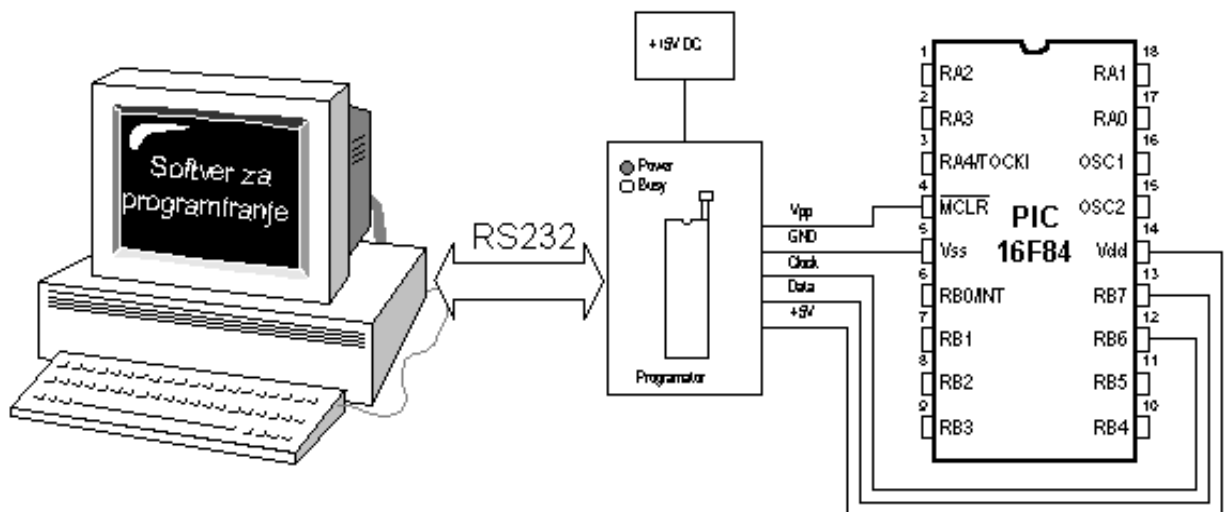
- BLINK.HEX - executable file which is written into the programming memory

File with the HEX ending is in effect the program that is written into the programming memory of a microcontroller. The programming device with accessory software installed on the PC is used for this operation. Programming device is a contrivance in charge of writing physical contents of a HEX file into the internal memory of a microcontroller. The PC software reads HEX file and sends to the programming device the information about an exact location onto which a certain value is to be inscribed in the programming memory. PIC BASIC creates HEX file in a standard **8-bit Merged Intel HEX** format accepted by the vast majority of the programming software. In the text below the contents of a file BLINK.HEX is given.



```
:100000002828A301A200FF30A207031CA307031C9A
:1000100023280330A100DF300F200328A101E83E90
:10002000A000A109FC30031C1828A00703181528FC
:10003000A0076400A10F152820181E28A01C222844
:1000400000002228080083130313831264000800B1
:1000500006148316061083120130A300F430022028
:1000600006108316061083120130A300F43002201C
:0600700028286300392876
:02400E00753DFE
:00000001FF
```

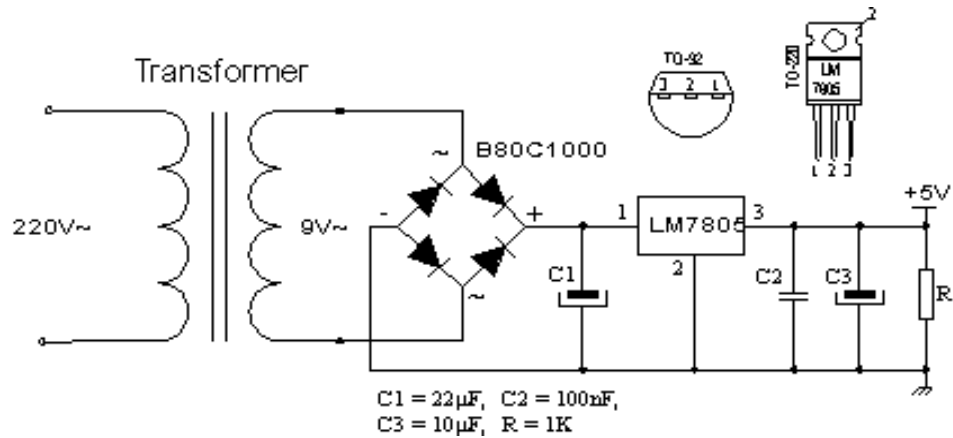
Besides reading of a program code into the programming memory, the programming device serves to set the configuration of a microcontroller. Here belongs the type of the oscillator, protection of the memory against reading, switching on of a watchdog timer etc. The connection between PC, programming device and the microcontroller is shown.



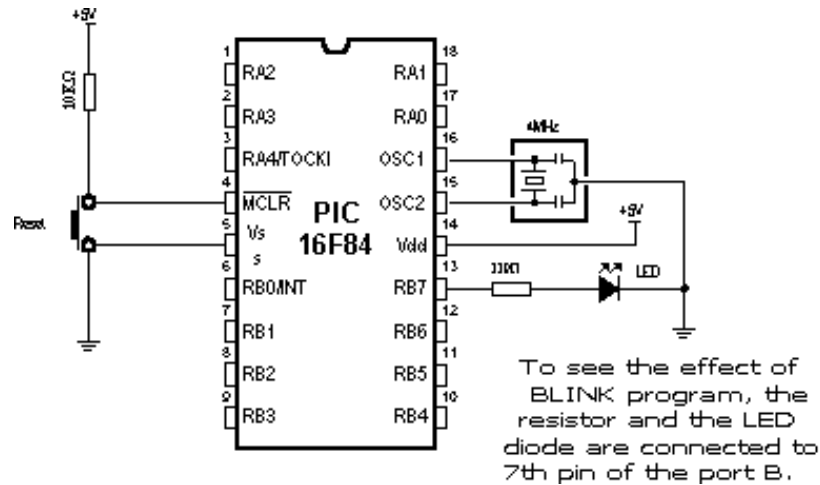
The programming software is used exclusively for the communication with the programming device and is not suitable for any code writing. The one comprising text editor, software for programming microcontroller and possibly the simulator as an entity bears the name IDE i.e. *Integrated Development Environment*. One such environment is a *Microchip's* software package MPLAB.

1.6 Running your program

For correct operating of a microcontroller, i.e. correct running of a program it is necessary to assure the **supply of the microcontroller, oscillator and the reset circuit**. The supply of the microcontroller can be organized with the simple rectifier with Gretz junction and LM7805 circuit as shown in the picture below.



The oscillator of the microcontroller can be a 4MHz crystal and either two 22pF capacitors or the ceramic resonator of the same frequency (ceramic resonator already contains the mentioned capacitors, but contrary to the oscillator has three termination instead of only two). The speed at which the microcontroller operates i.e. the speed at which the program runs depends heavily on this frequency of an oscillator. In the course of an application development the easiest to do is to use the internal reset circuit in a manner that MCLR pin is connected to +5V through a 10K resistor. In the sequence of text the scheme of a rectifier with circuit of LM7805 which gives the output of stable +5V, as well as the minimal configuration relevant for the operation of a PIC microcontroller.



Minimal hardware configuration necessary for the operation of PIC microcontroller

After the supply is brought to the circuit structured according to the previous pictures, PIC microcontroller should look animated, and its LED diode should be twinkling once each second. If the signal is completely missing (LED diode doesn't twinkle), the check is to be done to ascertain if the +5V is present at all the corresponding tentacles on PIC microcontroller.

1.7 Problem with starting your program (what if it doesn't work)

The usual problems of bringing the PIC microcontroller into the working conditions comprise the check of few external components and inquiry into the fact whether their values correspond to the wanted ones or whether all the connections with the microcontroller have been done properly. There are some suggestions that may be useful in order to help bringing to

Step 1. Check whether the MCLR pin is connected to 5V or over a certain reset circuit or simply with 10K resistor. If the pin remains disconnected, it's level will be "floating" and it may work sometimes, but usually it won't. Chip has power-on-reset circuit, so that appropriate external "pull-up" resistor on MCLR pin should be sufficient.

Step 2. Check whether the connection with the resonator is stable. For most PIC microcontrollers to begin with 4MHz resonator is well enough.

Step 3. Check the supply. PIC microcontroller spends very little energy but the supply must be pretty well filtrated. At the rectifier exit, the current is direct but pulsing and as such is by no means suitable for the supply of microcontroller. To avoid this pulsing, the electrolytic capacitor of high order of capacitance (say 470 μ F) is placed at the exit of a rectifier.

If PIC microcontroller supervises the devices that pull lot of energy from the energy source they can in their own rights provoke enough malfunctioning on the supply lines so that the microcontroller can stop working normally and start revealing somewhat strange behavior. Even seven-segmented LED display may well induce tension drops (the worst scenario is when all the digits are 8, for then LED display needs most power), if the source itself is not capable to procure enough current (for the case of 9V battery just for an example).

Some PIC microcontrollers have multi-functional entrance\exit pins, as it is the case with PIC16C62x family (PIC16C620, 621 and 622). The microcontrollers belonging to this family are provided with analogue comparators at port A. After putting those chips to work, port A is set onto an analogue mode, which brings about the unexpected behavior of the pin functions on this port. Any PIC microcontroller with analogue entrances will after reset show itself in an analogue mode (if the same pins are used as digital lines they must then be set into a digital mode).

One of the possible sources of troubles is that the fourth pin of the port A shows singular behavior when it is used as exit (because this pin has open collectors exit instead of usual bipolar state). That implies that the inscription of the logical zero on this pin will nevertheless set it on the low level, but the inscription of logical unit will let it float somewhere in between instead of setting it at high level. To coerce this pin react in a proper way the pull-up resistor is placed between RA4 and 5V. The magnitude of this resistor may be between 4.7K and 10K, depending on the intensity of the current necessary for the convected entrance. This pin functions as any other pin used as an entrance (all the pins are after reset procedure set as exits).

During the work with PIC microcontrollers more problems are to be expected. Sometimes what is being tried seems like going to work, but it doesn't happen to be the case regardless of how hard had we put an effort. Normally there is more than one way to solve something. A different angle approach may bring a solution with the same effort.

Chapter 2

BASIC ELEMENTS OF PIC BASIC LANGUAGE

[Introduction](#)

[2.1 Identifiers](#)

[2.2 Labels](#)

[2.3 Constants](#)

[2.4 Variables](#)

[2.5 Sequences](#)

[2.6 Modifiers](#)

[2.7 Symbols](#)

[2.8 Direction INCLUDE](#)

[2.9 Comments](#)

[2.10 Programming line with more instructions](#)

[2.11 Transfer of a instruction into another line](#)

[2.12 Define](#)

[2.13 DISABLE](#)

[2.14 ENABLE](#)

[2.15 ON INTERRUPT](#)

[2.16 RESUME](#)

Introduction

Next chapter describes the basic elements of a PIC BASIC language and the mode to use them in the efficient program writing. It is somewhat of an artistry to write a code that is both readable and easy to handle. Program is supposed to be understandable, before all, to the programmer himself and then later to his colleagues in charge of doing some corrections and adding as well. In the further text is given one example of the program written in a clear and manifest way.

		Program: PROBA.BAS Page: 1 / 1

		'* Ime prog.: PROBA.BAS *
		'* Copyright: Copyright (c) 2001 mikroElektronika *
		'* Datum : 11/20/01 *
		'* Verzija : 1.0 *
		'* Napomena : Efekat blinkanja dioda *

Program's header		
Define direction	DEFINE OSC 8	' Definisanje oscilatora
Symbols	symbol LEDDiode = PORTB	' Led diode su na portu B
Constants	Ugasi con \$00	' Konstanta
	Upali con \$FF	' Konstanta
Variable	i var byte	' Pomocna promenljiva
Command	TRISB = \$00	' svi pinovi porta B su izlazni
	i=0	' inicijalizacija promenljive i
Label	Main:	' Pocetak programa
Subroutine	for i=1 to 10	' Petlja koja ponavlja blinkanje
	gosub Blink	' 10 puta
	next i	
	goto Main	' ponovi celu petlju
Comment		
	Blink:	' pocetak podprograma
	LEDDiode=Upali	' PORTB=\$FF
	Pause 1000	' pauza od jedne sekunde
	LEDDiode=Ugasi	' PORTB=\$00
	Pause 1000	' pauza od jedne sekunde
	Return	' povratak iz podprograma
	End	' Zavrsetak programa

Extensive use of comments, symbols, labels and other elements supported by PIC BASIC, program can be rendered considerably clearer and more understandable what is in later corrections and enlargement of the program offering programmer a great deal of help. In order to make it even more understandable it is advisable to separate the program into logical entities as those parts to which a jump with the **goto** instruction can be performed or subprograms to be called with the **gosub** instruction. Labels indicating the beginning of the segments of programs should have meaning making some obvious sense. If it, say, exists such segment of a program that switches on and off LED diodes on some of the ports, the label indicating the beginning of that part of the program could well be for example "**Blink**" (LED diodes shine or go dark - therefore they blink) or the like.

Elements determining one BASIC program are the following:

- Identifiers
- Labels
- Constants
- Variables
- Sequences
- Modifiers
- Symbols
- Comments
- Include
- DEFINE
- _ (continuation of a instruction transferred into another line)
- On interrupt
- Disable
- Enable
- Resume

Although they are many at first glance only but a few of them is fair enough for writing approximately 90% of all programs. Nevertheless for the sake of completeness on all the elements will be treated on the following pages.

2.1 Identifiers

Identifier represents the name of some PIC BASIC element. Identifiers are used in PIC BASIC in order to sign program lines and the names of various symbols. Identifier itself could be any string of letters, numbers or even dashes with the limit that it is not allowed to begin with a number. Identifiers don't distinguish small and capital letters, so that the strings TASTER and Taster are treated the same way. The maximum length for such strings is 32 characters.

```
symbol Taster = PORTA.0      ' RAO se identifikuje kao "Taster"  
symbol LED_0 = PORTB.0     ' RBO se identifikuje kao "LED_0"
```

2.2 Labels

Label represents textual sign for some programming line or respectively some of its fragments on which the program can jump through some of the instructions used to change the program flow. It is obligatory to end the label with. Contrary to many old BASIC versions, PIC BASIC doesn't allow numerical values as labels.

```
symbol Taster = PORTA.0  
symbol LED_0 = PORTB.0  
  
B0 var byte  
  
Main:           ' Label Main  
    B0 = 0  
    button Set,0,255,0,B0,1,LED_toggle  
    goto Main  
  
LED_toggle:   ' Label LED_toggle  
    toggle LED_0  
    goto Main  
end
```

2.3 Constants

Name_constants **con** value_constants

With this declaration is to some chosen name assigned the value that is constant. For example the constant minute has the value of 60 seconds, bearing the recollection to the number of seconds in a minute. Written at whatever program position, minute will be interpreted by compiler as if it had been written 60. There are two very important reasons for such habit in program writing. The first one is the programmers wish to be more manifest. Good visibility is achieved by giving to the variables and constants those names that could be associated with the very function they assume within the program. On the other hand, the bigger flexibility of the program is obtained as well. It is for an example so that if it becomes necessary in some future work to use the same code but with a change value of the constant, it is enough make a change in the part for declaration instead performing search and replace throughout the program.

```
minute con 60          ' No. of seconds in a minute  
if seconds < minute then minute = minute + 1 ' If the number of seconds is different  
                                           ' from 60, raise the variable minutes
```

Constants can be equally written in decimal, hexadecimal and binary form. Decimal constants are written without any prefix. Hexadecimal constants start all with a sign \$ and binary with %. To make the programming easier, single letters are converted into their ASCII counterparts. The sign constants must be placed into the inverted comas and they

contain only one letter as a rule (in adverse case they are string constants).

56	' 56 decimal
\$0F	' 15 hexadecimal
% 10001100	' 140 binary
"A"	' ASCII value for decimal 65
"d"	' ASCII value for decimal 100

2.4 Variables

Name_variable **var** Type_variable

Variables serve for temporary storing of data and results of various arithmetic and logical operations. Variables are stored on the microcontrollers RAM locations, which means that the total number of the variables that can be used depend on the size of RAM.

Accordingly for the 36-byte microcontroller, 22 bytes are reserved for variables.

Variable defining is achieved with the formal word **var** at the beginning of the program. PIC BASIC supports variables like *bit*, *byte* and *word*. Variable type is selected with reference to the expected value that this same variable can assume in the course of the program run. Therefore the variable of the *bit* type can take value of 0 or 1, the variable of the *byte* values from 0 to 256 and finally, *word* from 0 to 65535.

Fleg	var	bit	' Fleg is a variable of the type bit
B0	var	byte	' B0 is a variable of the type byte
W0	var	word	' W0 is a variable of the type word
B0	var	W0.byte0	' B0 is a first byte of the word W0
B1	var	W0.byte1	' B1 is a second byte of the word W0

2.5 Sequences

Name_sequence **var** type_element [number of the elements]

Sequences of the variables are defined in a similar way as we have done with the variables. "Type_element" represents the value of every element of the sequence, and can be *bit*, *byte* or *word*.

The number of the elements of the sequence is given through value between "[]". Each element of the sequence is accessible by an index. Index starts with zero. When we come to define the number of the elements of the sequence one must always have in mind that the number of locations in RAM memory on which we intend to store variables finite. The next table shows the maximal number of the elements of various types.

The size of the sequence	
Element of the sequence	Maximal number of elements
BIT	256
BYTE	96*
WORD	48*

* Depends on microcontroller

Sequence1 **var** byte[10] ' the sequence of 10 elements of the type byte

Sequence1 [0] represents the first element of the sequence and sequence1 [9] the last element of the sequence "sequence1".

Sequence2 **var** byte[8] ' the sequence of 8 elements of the type byte

Sequence2 [0] represents the first element of the sequence and sequence2 [7] the last element of the sequence "sequence2".

2.6 Modifiers

new_name **var** old_name

By means of modifier it is possible to introduce a new name for the variable already defined. This direction is used relatively rarely but it ought to be mentioned for the sake of completeness. It is used in an identical way as a direction for the definition of the variables. Introduction of a new name is effectuated through the official word **var**.

ADCRestult var word HigherByte var ADCresult.byte0	' The new name for the higher byte of the ' word ADCresult
--	---

2.7 Symbols

symbol old_name = new_name

Symbols are granted the function exactly the same as direction for modifying variables, i.e. they serve for assigning the new names to the variables and constants. Symbols are introduced for the compatibility of the programs written for Basic Stamp and cannot be used for introducing variables.

symbol Taster = PORTA.0	' Taster is a new name for RA0
symbol LED_0 = PORTB.0	' LED_0 is a new name for RB0

2.8 Direction INCLUDE

INCLUDE "the name of the file"

Direction INCLUDE serves for inserting of a segment of a BASIC file. In this manner is rendered possible to store some general definitions of variables or subroutines that are being executed as parts of several different programs. The effect achieved is the same as if at the location on which is placed the direction INCLUDE simultaneously copied the contents of whole file.

```

Include "modedefs.bas"           ' The transfer modes that use the
                                   ' commands SERIN and SEROUT

symbol SO = PORTA.3
symbol SI = PORTB.0
B0 var byte

Loop:
    serin SI,T2400,B0
    serout SO,T2400,[B0]
    goto Loop
end

```

2.9 Comments

' Comment.... '

In the course of program writing there's a space for lot of comments even if it may be self-evident what is the main purpose of the program. Although it may well seem as a sheer waste of time, it may play later a crucial role (comments don't occupy an additional memory space in the memory of a microcontroller). Comments should give useful instructions about all that the program is doing. Comment as Set Pin0 to 1 simply explains the syntax of the language but fails to pinpoint the purpose of the act. Something of a sort Turn the Relay on may prove itself to be much more useful.

At the beginning of the program it should be described what is the program used for, who were the authors and when was it written. Stipulating the information concerning revision and the exact date may be useful too. Even every concrete statement about connection to each pin can be crucial in an effort to memorize the very hardware for which this program was designed to operate.

```

symbol LED = PORTB.0           ' LED diode is connected to RB0

Main:                           ' The beginning of the program

    LED = 1                     ' Turn on LED
    Pause 500                   ' Pause 500 mS
    LED = 0                     ' Turn off LED
    Pause 500                   ' Pause 500 mS
    goto Main                   ' Jump to the beginning of program

end                             ' End of the program

```

2.10 Programming line with more instructions

Compactness and better visuality of a program can be achieved by logically grouping instructions by using ":". In that way the block of instructions can be placed all in a single line, while instruction remain mutually separated with ":".

```

B2 = B0
B0 = B1
B1 = B2

```

The three upper instructions can be written in a single row as:

```

B2 = B0 : B0 = B1 : B1 = B2

```

2.11 Transfer of a instruction into another line

In case that instruction has big number of parameters so that they cannot stay all into another programming line, there is a possibility that the intake of parameters continue in the next row what is done by means of "_" at the end of line. The typical examples are the instructions *lookup*, *branch* and *sound*.

```
lookup KeyPress,["1","4","7","*","2","5","8","0","3","6","9","#","N"]
```

2.12 Define

DEFINE the value parameter

Instructions of the PIC BASIC language can have some parameters from which depends the exact way the instructions are executed. Those parameters assume some predefined values that appear in the most of the cases. A frequency of an oscillator is a good example for that. If not otherwise stated the tact of the oscillator is taken by default as 4MHz. In case that the used oscillator is of a different frequency from 4MHz it is necessary using the DEFINE direction to specify that frequency and communicate it to all the programs that contain within instructions depending on the tact of the microcontroller. One such instruction is for the serial transfer. In case that the instruction DEFINE is omitted and in gear is 8Mhz instead of 4Mhz oscillator, all the instructions that depend on the tact of microcontroller will be executed 2 times quicker. For instance, if the parameter of the speed of transfer amounts to 9600 bauds by using SERIN instruction, the data transfer would be effectuated at the speed 19200. In the same way the instruction pause 1000 the delay realized would be 0.5s instead 1.0s. It is also possible similarly to upgrade the resolution of the instructions. What is next is the review of the usage for DEFINE direction in case of adjusting of parameters explained within each particular instruction.

The use of a direction DEFINE		
parameter	description	instruction on which it acts
I2C_HOLD 1	pause 12C transfer while the tact is on a low level	I2COUT, I2COUT
I2C_INTERNAL 1	internal EEPROM in series 16Cexxx and 12Cxxx of the PIC microcontroller	I2COUT, I2COUT
I2C_SCLOUT 1	serial tact is a bipolar at the place of an open collector	I2CWRITE, I2CREAD
I2C_SLOW 1	for the tact > BMHz OSC with the devices of a standard velocity	I2CWRITE, I2CREAD
LCD_DREG PORTD	LCD data port	LCDOUT, LCDIN
LCD_DBIT 0	Initial bit of a data 0 or 4	LCDOUT, LCDIN
LCD_RSREG PORTD	RS (Register select) port	LCDOUT, LCDIN
LCD_RSBIT 4	RS (Register select) pin	LCDOUT, LCDIN

LCD_INSTRUCTIONUS 2000	the time of delay of instruction in microseconds (us)	LCDOUT, LCDIN
LCD_DATAUS 50	the time of delay of data in microseconds	LCDOUT, LCDIN
OSC 4	tact of the oscillator in MHz: 3(3.58) 4 8 10 12 16 20 25 32 33 40	all instructions of the serial transfer and next pause
OSCCAL_1K 1	setting of OSCCAL for PIC12C671/CE673 microcontrollers	
OSCCAL_2K 1	the number of data bits	
SER2_BITS 8	the slowing of the tact of transfer	SHIFTOUT, SHIFIN
SHIFT_PAUSEUS 50	instruction LFSR in 18Cxxx microcontrollers	LFSR
BUTTON_PAUSE 10		BUTTON
CHAR_PACING 1000		SEROUT, SERIN
HSER_BAUD 2400		HSEROUT, HSERIN
HSER_SPBRG 25		HSEROUT, HSERIN
HSER_RCSTA 90h		HSEROUT, HSERIN
HSRE_TXSTA 20h		HSEROUT, HSERIN
HSER_EVEN 1		HSEROUT, HSERIN
HSER_ODD 1		HSEROUT, HSERIN

Example:

2.13 DISABLE

DISABLE

Before entering the interrupt routine, it is necessary to switch off the interrupts in order to avoid any new interruption in the course of data processing. The interruptions are forbidden in a manner that the instruction "DISABLE" reset the bit GIE in the register INTCON.

Disable	‘ Forbid the interruptions
ISR:	‘ Start of an interruption routine
.	
.	
Resume	‘ The end of the interruption routine
Enable	

2.14 ENABLE

ENABLE

In the course of execution of the interruption routine, the interrupts must be forbidden by resetting the bit GIE in the INTCON register. When the interruption processing is finished, the interruptions must be allowed once again with the instruction "ENABLE".

```
        Disable
ISR:    ^ Start of the interruption routine
        .
        .
        ^ The end of the interruption routine
        Resume
Enable ^ Allow interruptions
```

2.15 ON INTERRUPT

On interrupt LABEL

With instruction "On interrupt" is indicated the label on which the program will "jump" when the interruption happened, i.e. from which label the interruption routine starts.

```
On interrupt ISR ^ The interruption routine starts from the label ISR
Main:    ^ Main program
        goto Main
        Disable
ISR:    ^ Start of the interruption routine
        .
        .
        ^ The end of the interruption routine
        Resume
        Enable
```

2.16 RESUME

RESUME

Return from the interruption routine to the main program.

```
        Disable
ISR:    ^ Start of the interruption routine
        |
        |
        |
        ^ End of the interruption routine
Resume ^ Exit from the interruption routine
        Enable
```

Chapter 3

OPERATORS

[Introduction](#)

[3.1 Expressions](#)

[3.2 Instructions](#)

[3.3 Arithmetical operators](#)

[3.3.1 Multiplication](#)

[3.3.2 Division](#)

[3.3.3 Shift](#)

[3.3.4 ABS](#)

[3.3.5 COS](#)

[3.3.7 DIG](#)

[3.3.8 MAX and MIN](#)

[3.3.9 NCD](#)

[3.3.10 REV](#)

[3.3.11 SIN](#)

[3.3.12 SQR](#)

[3.4 Bit operators](#)

[3.5 The operators of comparison](#)

[3.6 Logical operators](#)

Introduction

The PIC BASIC language possesses the operator set used to assign the values, compare objects and perform multitude of other operations. The objects manipulated for that purposes are called operands (which themselves can be variables or constants). The operators of PIC BASIC language must have at least two operands. They serve to create instructions and expressions that together with variables, constants and comments in effect compose the program.

3.1 Expressions

Combinations of operators and operands are called expressions. The expression does the computation and furnishes the result or starts some other activity.

A = ' The expression that sums up the values of the variables B and C and
B + ' stores the result into the variable A
C

In application of any expression the attention must be paid that the result of the computation must be within the range of variable A in order to avoid the overflow and therefore the evident computational error. If the result of expression amounts to 428, and the variable A is of BYTE type having range between 0 and 255, the result accordingly obtained will be 172 - obviously the wrong one.

3.2 Instructions

Each instruction determines an action to be performed. As a rule, the instructions are being executed in an exact order in which they are written in the program. However, the order of their execution can be changed as well employing the instructions for the change of the flow of a program to another segment of the program such as the instructions of the ramification, jump or interrupt.

IF Time = 60 **THEN**
GOTO Minute

' if A = 23 jump to label Minute

Instruction IF...THEN contains the conducting expression Time=60 composed in its own rights of two operands, the variable Time, constant 60 and the operator of comparison (=). The instructions of PIC BASIC language can be distinguished as the instructions of **choice** (decision making) **repeating** (loops), **jump** and specific **instruction for an access to the peripheries of the microcontrollers**. Each of these instructions is explained in detail in Chapter 4.



Operators are numerous, but for almost 90% of all the programs it is necessary to know only few of them. It suffices to look how many operators are used in the examples in Chapter 5, 6 and 7.

After the activities they perform, the operators can be classified into the following categories:

- Arithmetic operators
- Bit operators?
- The operators of comparison
- Logical operators

3.3 Arithmetic operators

All arithmetic operators work in 16-bit precision with the unsigned values what means that the range of the operand is from 0 to 65535. In order to group operations, one may use brackets.

$$A = (B + C) * (D - E)$$

In the following table all the supported arithmetic operators are listed.

Operator Description			
Operator	Description	Operator	Description
+	summation	ABS	absolute value of a number
-	subtraction	COS	cosine of an angle
*	multiplication	DCD	bit decoding
**	the result is in higher 16 bits	DIG	value of the digit for a decimal number
*/	the result is in middle 16 bits	MAX	maximum of a number
/	division	MIN	minimum of a number
//	remainder	NCD	priority coding
<<	left shift	REV	bit reversing
>>	right shift	SIN	sine of an angle
=	assignment of value	SQR	square root of a number

3.3.1 Multiplication

Syntax:	<pre>L0 = W1 * 100 L1 = W1 ** W2 L2 = W1 */ W2</pre>
Description:	<p>PIC BASIC pro does not support directly the work with the 32-bit numbers. It is usual to present a 32-bit variable as a two 16-bit variables. Operator '*' reverts lower 16 bits of a 32-bit result. Operator '**' reverts higher 16 bits of a 32-bit result. These two operators can be used in a combined way for computing 16x16 multiplications in order to produce 32-bit results.</p>
Example:	<pre>L0 var long W1 var word W2 var word Main: L0 = W1 * 100 ' Multiplies value W1 with 100 and ' stores the result in lower 16 bits of L0 L0 = W1 ** 100 ' Multiplies value W1 with 100 and ' stores result in 16 higher bits of L0 L0 = W1 */ W2 ' Reverts the 16 middle bits of the result Loop: goto Loop END</pre>

3.3.2 Division

Syntax:	<pre>W0 = W1 / 100 W2 = W1 // 100</pre>
Description:	<p>As it is the case with multiplication, the operation of division is done over the 16 bit operands. Operator '/' reverts 16-bit integer result while the operator '//' reverts the remainder.</p>
Example:	<pre>W0 var word W1 var word W2 var word Main: W0 = W1 / 100 ' Divide the value W0 with 100 and ' store the integer result in W1 W2 = W1 // 100 ' Remainder store in W2 Loop: goto Loop END</pre>

3.3.3 Shift

Syntax:	W0 = W0 << 3 W0 = W0 >> 1
Description:	Operators of the shift perform the shift towards left or right from 0 to 15 times. All the new bits that enter from the side have value 0. These two operators belong to the operators over the bits.
Example:	<pre> Main: W0 = W0 << 3 ' Shift W0 three places to the left ' (same as multiplication with 8) W0 = W0 >> 1 ' Shift W0 one place to the right ' (same as division with 2) Loop: goto Loop END </pre>

3.3.4 Absolute value of a number

Syntax:	B0 = ABS B1
Description:	ABS gives the absolute value of a number. If ABS gets applied to the variable of the BYTE type greater than 127 (set MSB) the result is 256. If the ABS gets applied to the variable of WORD type greater than 32767 (the bit set is of the biggest weight - MSB) result is 65536.
Example:	<pre> B0 var byte B1 var byte Main: B0 = ABS B1 ' Absolute value of B1 store in B0 Loop: goto Loop END </pre>

3.3.5 Cosine of an angle

Syntax:	B0 = COS B1
Description:	COS reverts the 8-bit value of the cosine. The result is in the second complement (i.e. within the range -127 to 127). For that reason it is necessary to use the lookup table in order to determine the result (cosine of an angle goes in the binary range between 0 and 255 in contrast with usual 0 to 359 degrees).
Example:	<pre> B0 var byte B1 var byte B2 var byte Main: B0 = COS B1 ' 8-bit value of cosine B1 store in B0 ' (index of Lookup table) Lookup B0, [constant to determine_cosine], B2 ' After this instruction the true value of ' cosine is stored in B2 Loop: goto Loop END </pre>

3.3.6 The decoded bit value

Syntax:	B0 = DCD N
Description:	DCD gives the decoded bit value of the operand whose value is in the range within 0-15. If the operand is 0 then the zeroth bit of the result is 1, and if the operand reads as 7, the seventh bit of the result is 1.
Example:	<pre> B0 var byte Main: B0 = DCD 2 ' Contents B0 is %00000100 Loop: goto Loop END </pre>

3.3.7 DIG The value of the digit for a decimal number

Syntax:	W = W1 DIG N
Description:	DIG furnishes the value of the digit of a decimal number. The number whose digits are looked for is 0-3 where 0 is a last right digit i.e. digit of the smallest weight (it is most often used for the work with seven-segment digits for extraction of the digits to be displayed).
Example:	<pre> B0 var byte B1 var byte Main: B1 = 5843 B0 = B1 DIG 0 ' Contents B0 is 3 B0 = B1 DIG 1 ' Contents B0 is 4 B0 = B1 DIG 2 ' Contents B0 is 8 B0 = B1 DIG 3 ' Contents B0 is 5 Loop: goto Loop END </pre>

3.3.8 MAX and MIN Maximum and Minimum of a number

Syntax:	<pre> B0 = B1 MAX 100 B0 = B1 MIN 100 </pre>
Description:	The operator's maximum and minimum are used whenever it is necessary to revert one out of two values that are being compared. If those numbers are for example 100 and 200 operator Max will revert the value 200 and operator Min, value 100. To the difference from the operators "bigger then" and "less then" they revert the entire value and not only the quantification whether some value is smaller or bigger then the other.
Example:	<pre> B0 var byte B1 var byte Main: B0 = B1 MAX 100 ' B0 is either 100 or B1 unless B1 ' contains the value bigger then 100 B0 = B1 MIN 100 ' B0 is either 100 or B1 unless B1 ' contains the value smaller then 100 Loop: goto Loop END </pre>

3.3.9 NCD Priority coding

Syntax:	B0 = NCD %01001000 B0 = NCD %00001111
Description:	NCD furnishes the value that is coded with the priority code. That gives the position of the first unit, which it encounters from the left side. If the operand is 0 the result is 0 as well.
Example:	<pre>B0 var byte Main: B0 = NCD %01001000 ` Contents B0 is 7 B0 = NCD %00001111 ` Contents B0 is 4 Loop: goto Loop END</pre>

3.3.10 REV Reverting of the lowest bits of the operand

Syntax:	B0 = %10101100 REV 4
Description:	REV reverts the order of the lowest bits of the operand. The number of the bits that can be reverted goes from 1 to 16.
Example:	<pre>B0 var byte Main: B0 = %10101100 REV 4 ` Contents B0 is %10100011 Loop: goto Loop END</pre>

3.3.11 SIN Sine of an angle

Syntax:	B0 = SIN B1
Description:	SIN reverts the 8-bit value of the sine. The result is in the second Complement (i.e. within the range -127 to 127). For that reason it is necessary to use the lookup table in order to determine the result (sine of an angle goes in the binary range between 0 and 255 in contrast with usual 0 to 359 degrees).
Example:	<pre>B0 var byte B1 var byte B2 var byte Main: B0 = SIN B1 ` 8-bit value of sine B1 store in B0 ` (index of Lookup table) Lookup B0, [constant to determine_sine], B2 ` After this instruction the true value of sine is ` stored in B2 Loop: goto Loop END</pre>

3.3.12 SQR Square root

Syntax:	B0 = SQR W1
Description:	SQR reverts a value of a square root. Result is stored into the variable of BYTE type.
Example:	<pre> B0 var byte W1 var word Main: B0 = SQR W1 ' Square root of W1 store into B0 Loop: goto Loop END </pre>

3.4 Bit operators

One of the more important properties of higher programming languages is their capacity to go down to the lower level i.e. the level of the assembler. Bit operators furnish the access to the registers and memory of a microcontrollers at the level of a single bit. Operators supported by the language PIC BASIC are given in the table below:

Bit operators	
Operator	Description
&	Logical AND over the bits
	Logical OR over the bits
^	Logical XOR over the bits
~	Logical NOT over the bits
&/	Logical NAND over the bits
/	Logical NOR over the bits
^/	Logical NXOR over the bits

The value result of the expression depends on the fact which of the listed logical operations is executed over the bits of the operand. In that way, it is possible to extract, delete, set or invert the certain bit of the operand.

Example1:

```
B0 = B0 & %00000001
```

The upper instruction extracts the value of the lowest bit of the variable B0. When the logical "AND" is performed with the zero, there will be 0 at the position of a corresponding bit (so that all the bits 1-7 will be zeroes). The value will depend on bit 0 in the variable B0 and if it is "0", the value of variable B0 will be "0" and if it is "1" the value of B0 will accordingly be "1".

Example2:

```
B0 = B0 & %00000100
```

The upper instruction sets bit2 in the variable B0. When the logical "or" is performed with the unity the result is always equal to "1" regardless of the state of the corresponding bit from B0.

Example 3:

```
B0 = B0 & %00000010
```

The upper instruction inverts the bit 1 in variable B0. If the bit was "1" then it turns into "0" and vice versa. The other logical operators are used only rarely so there's no need for their detailed explanation.

3.5 The operators of comparison

The expressions that contain the operators of comparison give after having compared the two operands the result true or false. If the expression of comparison is true then the instruction to be executed is the one on the left side, otherwise the execution of the program continues with the next instruction. The operators of comparison are shown in the table below:

Operators of comparison	
Operator	Description
= or ==	equal
<> or !=	not equal
<	less then
>	bigger then
<=	less then or equal
>=	bigger then or equal

These operators are most often used in examination of the conditions by the instructions such as IF...THEN.

Example:

```
If Seconds = 60 then minutes = minutes + 1  
Seconds = Seconds + 1
```

If the variable " Seconds" equals 60 the condition of the comparison is true and the instruction "Minutes=Minutes+1" will be executed then. Unless the expression is not true the instruction "Seconds=Seconds+1" will be executed instead.

3.6 Logical operators

Logical operators serve for the operations over the variables, which take two possible values 0 or 1. These values may well be interpreted as "condition is fulfilled" what corresponds to state "1" and "condition is not fulfilled" which corresponds to the state "0". They are used in the very same way as the operators of comparison within the frame of the instruction IF...THEN. The list of the logical operators is shown in the table below.

Logical operators	
Operator	Description
AND or &&	Logical AND
OR or	Logical OR
XOR or ^^	Logical XOR
NOT	Logical NOT
NOT AND	Logical NAND
NOT OR	Logical NOR
NOT XOR	Logical NXOR

Example1:

If A Or B THEN GOTO Lab

If the condition is fulfilled, i.e. if at least one of the operands A or B equal to one, then the program jumps to the label Lab.

Example2:

IF (Seconds>59) And (Minutes>59) THEN Hours=Hours+1

The conditions may be complex as well. Separating into the brackets is obligatory otherwise the result can be very unpredictable.

Chapter 4

INSTRUCTIONS (1/4)

[Introduction](#)

4.1 @	4.17 GOSUB	4.33 LOOKUP2	4.49 RETURN
4.2 ASM..ENDASM	4.18 GOTO	4.34 LOW	4.50 REVERSE
4.3 ADCIN	4.19 HIGH	4.35 NAP	4.51 SELECT-CASE
4.4 BRANCH	4.20 HSERIN	4.36 OUTPUT	4.52 SERIN
4.5 BRANCHL	4.21 HPWM	4.37 OWIN	4.53 SERIN2
4.6 BUTTON	4.22 HSEROUT	4.38 OWOUT	4.54 SEROUT
4.7 CALL	4.23 I2CREAD	4.39 PAUSE	4.55 SEROUT2
4.8 CLEAR	4.24 I2CWRITE	4.40 PAUSEUS	4.56 SHIFIN
4.9 CLEARWDT	4.25 INPUT	4.41 POT	4.57 SHIFOUT
4.10 COUNT	4.26 IF-THEN-ELSE	4.42 PULSIN	4.58 SLEEP
4.11 DATA	4.27 LCDOUT	4.43 PULSOUT	4.59 SOUND
4.12 DTMFOUT	4.28 LCDIN	4.44 PWM	4.60 STOP
4.13 EEPROM	4.29 {LET}	4.45 RANDOM	4.61 SWAP
4.14 END	4.30 LOOKDOWN	4.46 RCTIME	4.62 TOGGLE
4.15 FREQOUT	4.31 LOOKDOWN2	4.47 READ	4.63 WRITE
4.16 FOR-NEXT	4.32 LOOKUP	4.48 READCODE	4.64 WRITECODE
			4.65 WHILE-WEND

Introduction

All the programs regardless of the fact how complicated or simple they may be are nothing else but a strict flow of the executions of instructions.

Instructions of branching are used in program for the decision-making (in which one of two or more program paths is being chosen). The basic instruction of branching in PIC BASIC language is instruction *if*. This instruction has several variations that furnish necessary flexibility required for the realization of the logic of the decision-making (these variations comprise the use of term *else* and insertion of the instructions).

Instructions of repeating give the possibility of repeating one or more single instructions. The conducting expression determines how many times the repetition will be performed. The set of those instructions is composed of WHILE ... WEND and FOR ... NEXT.

Instructions of jump serve to change the flow of the program execution. The basic instruction of jump, *GOTO*, transfers the execution of the program to a signed instruction in a main program or inside subroutines. Other instructions of jump are *BRANCH*, *BRANCHL*, *CALL*, *GOSUB*, *RETURN* (these instructions are unavoidable in programs but their use is subject to certain restrictions).

Instructions of access to the peripheral devices facilitate the programmer's job. Now programmer can concentrate on the essence of the program he set out to solve, avoiding unnecessary waste of time in writing routine for LCD display or some other peripheral device he uses in his set. The set of instructions is such to satisfy the large part of needs in the design of even the most complicated microcontrollers systems.

4.1 @ Inserts one programming line of assembler code

Syntax:	<code>@ assembler's instruction</code>
Description:	<p>If used at the beginning of the line @ enables free-style combining of the assemblers code and PIC BASIC code. Instruction @ can be used for insertion of the libraries written in assembler as well.</p> <p>It should be taken notice that the further access from assembler towards variables works through the lower dash added to the variables name. In an example below, the variable B0 is used as _B0 in assembler programming line.</p>
Example:	<pre> @include "some_asm_program.asm" ' inserts an assembler code library B0 var byte Main : @ bsf _B0, 7 ' sets the seventh bit of variable B0 Loop : goto Loop end </pre>

4.2 ASM..ENDASM Inserts the block of assembler instructions

Syntax:	<pre> ASM / assembler / ENDASM </pre> <p style="text-align: right;"><i>instructions</i></p>
Description:	<p>ASM and ENDASM instructions give the information that the code between ASM and ENDASM assembler type. Maximal size of the assembler code depends on the size of the programming memory of a microcontroller. In case of a PIC16F877 microcontroller the maximal value of an assembler code is 8K.</p>
Example:	<pre> Main : asm ' Beginning of asm part of the program bsf PORTA, 0 ' set RA0 to "1" bcf PORTB, 3 ' set RB3 to "0" endasm ' End of asm part of the program Loop : goto Loop end </pre>

4.3 ADCIN Write the values from the input of the internal AD converter

Syntax:	ADCIN <i>channel, variable</i>
Description:	<p>ADCIN performs A/D conversion of an input analogue signal in microcontrollers that have A/D converter built in chip (i.e. PIC16F877). The value read in is stored into a designated variable. Before use of ADCIN instruction the appropriate TRIS register must be initiated so that the given is designated input one. Beside that in ADCON1 register one has to set the input pins for analogue working regime, format of the results and tact of A/D converter.</p>
Example:	<pre> DEFINE ADC_BITS 8 ' Converted result will have 8, 10 or 12 bits DEFINE ADC_CLOCK 3 ' Clock for A/D converter DEFINE ADC_SAMPLEUS 10 ' Sampling time expressed in us B0 var byte Main : TRISA = \$FF ' All pins of port A are input ADCON1 = 0 ' PORTA is analog adcin 0 B0 ' Read the channel 0 and store the result into variable B0 </pre>

Example:	<pre> B0 var byte Main : branch B0, [lab1, lab2, lab3] Loop : goto Main lab1 : ' Labels where the program execution resumes after lab2 : ' the jump initiated by instruction BRANCH lab3 : end </pre>
-----------------	---

4.5 BRANCHL Jump to the label in second code segment

Syntax:	BRANCHL <i>index</i> , [<i>label1</i> { <i>label...</i> }]
Description:	BRANCHL (BRANCH long) is a instruction quite similar to BRANCH. The only difference is that BRANCHL can realize jump onto the location situated on the second code segment. BRANCHL instruction creates the code approximately two times greater than one created by BRANCH, so that in case that the whole code of a program is in one single code segment or occupies less then 2K of memory - use of BRANCH is recommended.
Example:	<pre> W0 var word Main : branchl W0, [lab1, lab2, lab3] Loop : goto Loop lab1 : ' Labels where the program execution resumes after lab2 : ' the jump initiated by instruction BRANCHL lab3 : end </pre>

4.6 BUTTON Reads the state of button on input pin

Syntax:	BUTTON <i>Pin, State, Delay, Speed, Variable, Action, Label</i>
Description:	<p>The Button instruction eliminates the influence of contact flickering due to the pressing on the button (debouncing), what could be interpreted by the program as the pressing of the button more than one time instead of only once. Beside this function, instruction Button secures the function of auto-repeat which enables execution of determinate instruction as long as we keep pressing the button. The time between consecutive execution of two instructions is specified with the argument <i>Speed</i>.</p> <p>Pin - Pin on which we have button.</p> <p>State - State of the pin when the button is pressed (0...1).</p> <p>Delay - Countdown time before we initiate auto-repeat (0...255). At value 0, there will be no auto-repeat. At value 255, the debouncing will be effectuated but without auto-repeat.</p> <p>Speed - Time of auto-repeat (0..255).</p> <p>Variable - Auxiliary variable of byte type (which must be defined at the very beginning of program)</p>
Example:	<p>The example below will at each pressing of the button, which is connected to RA0, change the state of pin. If the diode is tied to the same pin the effect of the twinkling of the diode will be manifested.</p>

	<pre> DEFINE BUTTON_PAUSE 50 TRISA = 0 TRISB = 255 B0 var byte ` Auxiliary variable Main: B0 = 0 ` Initialization of B0 button PORTB.0,0,100,10,B0,1,led goto Main ` Repeat the loop led: toggle PORTA.0 ` Change the pin state goto Main end </pre>
--	--

4.7 CALL It calls assemblers subroutine

Syntax:	CALL <i>label</i>
Description:	It executes the subprogram under the name <i>Label</i> in the language of assembler.
Example:	<pre> @include "init.asm" Main call init_sys Loop: goto Loop end </pre>

4.8 CLEAR Sets the value of every variable to 0

Syntax:	CLEAR
Description:	CLEAR sets the entire RAM registers in all databanks to zero. It also means that all the variables will simultaneously be set to zero.
Example:	<pre> clear ` Clear all the variables in RAM Main: goto Main end </pre>

4.9 CLEARWDT Resets the watchdog timer

Syntax:	CLEARWDT
Description:	Resets the watchdog timer
Example:	<pre>clearwdt ` Clear WDT Main: goto Main end</pre>

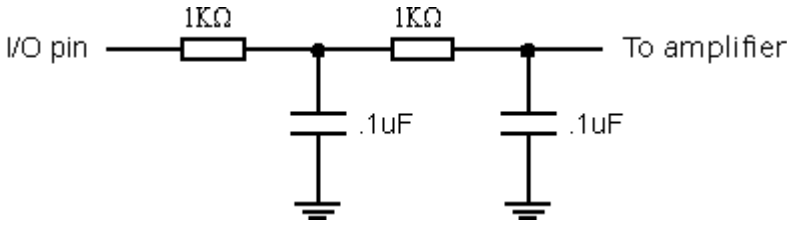
4.10 COUNT Counts the impulses on input pin

Syntax:	COUNT <i>Pin, Period, No_Impulses</i>
Description:	Counts the impulses that appear on a specified pin during the time interval defined with the <i>Period</i> variable. The number of the impulses is stored into the <i>No_Impulse</i> variable. Pin is automatically designated as input. <i>Period</i> is specified in milliseconds. If the oscillator is a 4Mhz one, check of a
Example:	<pre>WD var byte ` The supposition is to have not more ` then 255 impulses</pre>

4.11 DATA Effectuates writing into the EEPROM at the first programming

Example:	<p>data @5,1,2,3</p> <p>Writes in the values 1, 2, 3 on the locations 5, 6 and 7 in EEPROM memory.</p> <p>data word \$1234</p> <p>Writes in the values \$12 AND \$34 on the locations 0 and 1 in EEPROM memory.</p>
-----------------	---

4.12 DTMFOUT Generates the tone-dialing signal on the output pin

Syntax:	DTMFOUT <i>Pin, {Onms, Offms,} {Ton{, Ton...}}</i>
Description:	<p>Instruction DTMFOUT produces the tone encountered for example in the phones with tone dialing. Such characteristic tone is composed of two signals of different frequencies which serves for the detection of the pressed button. Pin is thereby designated output. The parameter "Onms" represents the duration time of each dial in milliseconds, while "Offms" is the duration of the brake between two consecutive tones. If no value of duration of either tone or brake is set, it goes without saying that "Onms" lasts 200ms and "Offms" 50ms. Tones are numerated 0-15. Those 0-9 are identical to those on a phone dial. Tone 10 represents button *, tone 11 button #, while to the tones 12-15 correspond the additional buttons A-D.</p>  <p>In order to obtain the desired sinusoidal signal at the output, the installation of a sort of filter is required.</p>
Example:	<pre> TRISB = \$FF ' All the pins of port A are exit ones Main: dtmfout PORTB.1,[2,1,2] ' Generate DTMF on RB1 loop: goto loop end </pre>

4.13 EEPROM Sets the initial contents for programming EEPROM

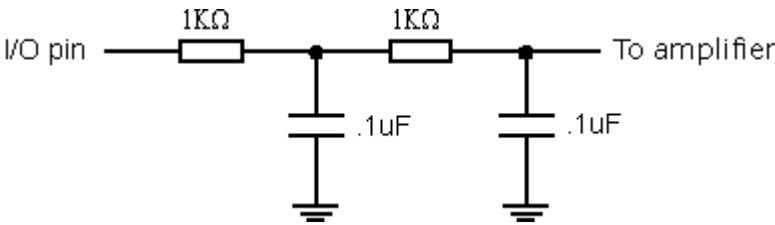
Syntax:	EEPROM <i>{@location,} constant {, constant}</i>
Description:	<p>In sets constants into the consecutive bytes of the EEPROM memory. If the optional value of the location is omitted, the first EEPROM instruction starts to store the constants beginning with an address 0, and the next instructions place the values on the following locations. If the value of location is stipulated, the values are written starting from that very location.</p> <p>Parameter "Constant" may be number or the sequence of constants. If "word" is not quoted before constant that is being written in, only the bytes of lowest weights are saved. The sequences of are stored as consecutive bytes of ASCII values.</p> <p>The instruction "EEPROM" is operative on only those PIC Microcontrollers, which possess</p>

	<p>EEPROM or FLASH programming memory built in the chip. The data are saved in the EEPROM space when the programming of microcontroller is definitely finished.</p> <p>For inwriting and reading of EEPROM memory in the course of the operation of the microcontroller, the instructions WRITE and READ are being used.</p>
Example:	<p>EEPROM @5,1,2,3</p> <p>Writes in the values 1, 2, 3 on the locations 5, 6 and 7 in EEPROM memory.</p> <p>EEPROM word \$1234</p> <p>Writes in the values \$12 AND \$34 on the locations 0 and 1 in EEPROM memory.</p>

4.14 END Marks the logical end of the program

Syntax:	END
Description:	Stops the further execution of the program and enters into the low energy consumption mode executing continuous SLEEP instructions in a loop. Instruction END should be put at the end of every program.
Example:	<pre>Main: goto Main end ^ The end of the program</pre>

4.15 FREQOUT Generates signal of a specified frequency on output pin

Syntax:	FREQOUT Pin, Onms, Freq1, Freq2
Description:	<p>FREQOUT generates the signals in the PWM form (<i>Pulse Width Modulation</i>) within the frequency range from 0 to 32767Hz on the pin defined in parameter "Pin" and with the duration specified in parameter "Onms".</p> <p>FREQOUT works best with a 20 MHz oscillator (while it is more difficult to filter the signal for the lower frequencies). "Onms" represents the duration of the signal in milliseconds.</p>  <p>In order to obtain the desired sinusoidal signal at output the installation of a sort of filter is required</p>

Example:	<p>freqout PORTB.1,2000,1000</p> <p>Generates the signal of the frequency 1000Hz in duration of 2s at the first pin of the p</p> <p>freqout PORTB.1,3000,1000,500</p> <p>Generates the signal of frequency 1000Hz and 500Hz in duration of 3s at the first pin</p>
-----------------	--

4.16 FOR-NEXT Repeating of the program segment

Syntax:	<pre> FOR Index = Start TO End {Step {-} Inc } { instructions NEXT {Index} </pre>
Description:	<p>The instructions of repeating one or more instructions. The conducting expression will determine how many times will repeating take place. "Index" is usually the variable employed for the control of how many times is for...next loop executed. If the parameter "Step" is not specified, it is understood that the variable "Index" is increased by one. (Index = Index + 1).</p>
Example:	<pre> auxiliary variable the program turns on and off the diodes at port B with 1s pause 200 times. auxiliary variable the program turns on and off the diodes at port B with 1s pause 100 times auxiliary variable the program turns on and off the diodes at port B with 1s pause 900 times </pre>

4.17 GOSUB Calls BASIC subroutines

Syntax:	GOSUB <i>label</i>
Description:	<p>Executes the PBP instructions of the program which are situated between label "label" and instruction RETURN. When program encounters the RETURN, the execution of the program goes on with the instruction line that follows GOSUB instruction. Part of the program code between the label and the RETURN instruction is commonly called subroutine.</p> <p>Subroutine can be "nested". In other words, it is possible that the subroutine calls some other program. Such programming shouldn't go beyond four levels depth because of the finite size of the PIC microcontroller stack.</p>
Example:	<pre> Main: gosub Blink ' Call subroutine Blink Loop: goto Loop Blink: ' Subroutine Blink PORTB=\$FF ' Turn on the diode on port B Pause 1000 ' Brake 1s PORTB=\$00 ' Turn off the diode on port B Pause 1000 ' Brake 1s Return End </pre>

4.18 GOTO Continues the execution of the program on a certain label

Syntax:	GOTO <i>label</i>
Description:	The execution of the program continues with the instruction line following the label "label". It is not recommended to use this command too often, because over-labeled programs are generally less intelligible.
Example:	<pre> Main: goto Blink ' Jump on label Blink Blink: ' Subroutine Blink PORTB=\$FF ' Turn on the diode on port B Pause 1000 ' Brake 1s PORTB=\$00 ' Turn off the diode on port B Pause 1000 ' Brake 1s goto Main End </pre> <p>The program above does exactly the same thing as the previous one, but without GOSUB instruction.</p>

4.19 HIGH Sets a logical "1" on the output pin

Syntax:	HIGH <i>Pin</i>
Description:	Sets the appropriate pin on the high level. Pin is thereby automatically designated output.
Example:	Main: high PORTA.0 ' Pin RA0 set on the high level Loop: goto Loop End

4.20 HSERIN Hardware asynchronous serial input

Syntax:	HSERIN { <i>Error</i> ,}{ <i>Timeout</i> , <i>Label</i> ,}[<i>Modifier</i> (,...)]
Description:	HSERIN receives one or more serial data. It can be used with PIC microcontrollers which have
Example:	B0 var byte

Example:	<pre> B0 var byte W1 var word Main: hserin [B0, dec W1] 'Take dec. digit from serial line goto Main end </pre>
-----------------	---

4.21 HPWM Generates PWM signal on the microcontroller pin

Syntax:	HPWM <i>Channel,Relation_on_off, Frequency</i>
Description:	<p>Command uses the hardware PWM on the microcontrollers who possess it for the generation of the PWM signal.</p> <p>The parameter "channel" defines the exact PWM channel that is to be used. In the two channel microcontrollers, the parameter "frequency" must be identical on both of them.</p> <p>The parameter "Relation_on_off" defines the relation between <i>on</i> and <i>off</i> signals on the pin. Value 0 sets the pin to always <i>off</i>, while 255 sets it to always <i>on</i>. All other values in the interval 0~255 define the appropriate ODNOS of <i>on</i> and <i>off</i> signals on the pin (for example, value 127 sets 50% <i>on</i> and 50% <i>off</i> signal).</p> <p>Parameter "Frequency" defines the frequency of the PWM signal (highest possible frequency for any oscillator is 32767 Hz) which depends on oscillator used. Lowest frequency depends on oscillator used.</p> <p>If not specified otherwise, PWM generates 0 timer by default.</p>
Example:	<pre> DEFINE HPWM2_TIMER 1 ' second channel uses timer 1 hpwm 2, 64, 1000 ' 25% PWM on 1kHz </pre>

4.22 HSEROUT Hardware asynchronous serial output

Syntax:	HSEROUT [<i>Item{,Item...}</i>]
Description:	<p>HSEROUT sends one or more serial data and is used in the PIC microcontrollers that have hardware supported serial communication (hardware USART). Parameters of serial transfer are determined by with the following DEFINE directives:</p> <pre> DEFINE HSER_RCSTA 90h ' Setting the receiving register DEFINE HSER_TXSTA 20h ' Setting the emitting register DEFINE HSER_BAUD 2400 ' Baud rate DEFINE HSER_SPBRG 25 ' Direct setting of SPBRG </pre> <p>When calculating transfer rate, HSERIN assumes that microcontroller works with the 4MHz oscillator. If different oscillator is used, new frequency must be specified with the following directive:</p> <pre> DEFINE OSC ' Specific oscillator frequency </pre>

<p>Format of serial data is 8N1 - 8 data bits, with no parity bit and with 1 stop bit. Some other formats, such as 7E1 (7 data bits, parity bit, 1 stop bit) or 7O1 (7 data bits, non-parity bit, 1 stop bit) may be used with the following DEFINE directives at the beginning of the program:</p> <pre>DEFINE HSER_EVEN 1 ' Only when we want to verify the parity</pre> <pre>DEFINE HSER_ODD 1 ' Only when we want to verify the non-parity</pre> <p>Serial transfer is hardware based, so you might need an additional driver for adjusting to RS-232 (MAX232).</p>													
<table border="1"> <thead> <tr> <th>Modifier</th> <th>Sends</th> </tr> </thead> <tbody> <tr> <td>{I}{S} BIN{1..16}</td> <td>binary number</td> </tr> <tr> <td>{I}{S} DEC{1..5}</td> <td>decimal number</td> </tr> <tr> <td>{I}{S} HEX{1..4}</td> <td>hexadecimal number</td> </tr> <tr> <td>REP c/n</td> <td>character c repeated n times</td> </tr> <tr> <td>STR ArrayVar {n}</td> <td>n character string</td> </tr> </tbody> </table>		Modifier	Sends	{I}{S} BIN{1..16}	binary number	{I}{S} DEC{1..5}	decimal number	{I}{S} HEX{1..4}	hexadecimal number	REP c/n	character c repeated n times	STR ArrayVar {n}	n character string
Modifier	Sends												
{I}{S} BIN{1..16}	binary number												
{I}{S} DEC{1..5}	decimal number												
{I}{S} HEX{1..4}	hexadecimal number												
REP c/n	character c repeated n times												
STR ArrayVar {n}	n character string												
Example:	<pre>B0 var byte B0 = 4 Main : hserout [dec B0, 10] ' send decimal number from variable B0 and constant 10 Loop: goto Loop end</pre>												

4.23 I2CREAD Reading data from I2C peripheral device

Syntax:	I2CREAD <i>Data, Frequency, Control_byte, {Address,} [Variable {, Variable...}]{Label}</i>
Description:	<p>Sends control and address data via I2C lines and received bytes are stored into "Variable".</p> <p>I2CREAD and I2CWRITE can be used for reading and writing data to peripheral units. These instructions work with I2C master byte in read and write modes and can be also used for communication with other devices with I2C interface, such as temperature sensors, A/D converters, etc.</p> <p>Higher 7 bits of control byte contain control code for chip selection or extra information on addresses, depending on device. The lowest bit is flag indicating the current mode - read or write.</p> <p>For example, for communicating with 24LC01B, requested address is 8-bit, control code is %1010 and <i>chip select</i> is unused, so that control byte would be %10100000 or \$A0.</p> <p>Formats of control bytes for several other serial EEPROMs are given in the table below:</p>

EEPROM	Capacity	Control word	Address size
24LC01B	128 bytes	%1010xxx0	1 byte
24LC02B	256 bytes	%1010xxx0	1 byte
24LC04B	512 bytes	%1010xxb0	1 byte
24LC08B	1K bytes	%1010xbb0	1 byte
24LC16B	2K bytes	%1010bbb0	1 byte
24LC32B	4K bytes	%1010ddd0	2 bytes
24LC65	8K bytes	%1010ddd0	2 bytes

bbb = block selection

ddd = device selection bits

xxx = has no effect

If 2-byte data (WORD) is received, higher byte is received first, and lower thereafter. For string transfer, STR goes before the name of the string, and number of clocks after \ .

a var byte[8]

I2CREAD PORTC.4, PORTC.3, \$a0, 0, [STR a\8]

If optional label is used, program will jump to the label if there is no response signal over the I2C interface. Standard transfer rate (100kHz) is achieved with 8MHz oscillator. For higher transfer rate (400kHz) 20MHz oscillator is used. If slower oscillator is used for the transfer, following directive should be used :

DEFINE I2C_SLOW 1

In order to have bipolar I2C clock interface and not an open collector, following DEFINE directive should be used:

DEFINE I2C_SCLOUT

Operating any peripheral units with I2C communication demands that you read supplier manuals and specifications.

Example:

B0 var byte

addr var byte

cont con %10100000 ' Control address of EEPROM

addr = 17 ' Data address is 17

Main:

I2CREAD PORTA.0, PORTA.1, cont, addr, [B0] ' Get data to variable B0

Loop: goto Loop

end

4.24 I2CWRITE Writing data to I2C peripheral device

Syntax:	I2CWRITE <i>Data, Frequency, Control_byte, {Address,} [Vari {, Vari...}]{,Label}</i>
Description:	I2WRITE sends control and address data via I2C interface. We define 8-bit or 16-bit address while defining variable put to address parameter (in order to correctly define address size, we must have accurate information on device we are communicating with).
Example:	B0 var byte addr var hvte

	end
--	-----

4.25 INPUT Designates I/O pin as input

Syntax:	INPUT <i>Pin</i>
Description:	INPUT designates the specific pin as input.
Example:	<p>Main:</p> <pre> input PORTA.0 ' Pin PORTA.0 is input. Instruction can be substituted with TRISB.0=1 TRISB.0=1 Loop: goto Loop end </pre>

4.26 IF-THEN-ELSE Conditional program branching

Syntax:	<p>IF Expression1 { AND / OR Expression2} THEN Label</p> <p>{instructions}</p> <p>ELSE</p> <p>{instructions}</p> <p>ENDIF</p>
Description:	<p>Instruction selects one of two possible program paths. Instruction IF is the fundamental instruction of program branching in PIC BASIC and it can be used in several ways to allow flexibility necessary for realization of decision making logic.</p> <div style="text-align: center;"> <pre> graph TD In[/in/] --> Expr{<expression>} Expr -- T --> Inst[instruction] Inst --> Expr Expr -- N --> Exit[/exit/] </pre> <p>if expression then instruction endif</p> </div> <p>The simplest form of instruction is shown on the picture above. Sample program below tests the button connected to RB0 - when the button is pressed program jumps onto the label "Add" where value of variable "w" is increased. If the button is not pressed, program jumps back onto the label "Main".</p>

Example:

w var byte

Main :

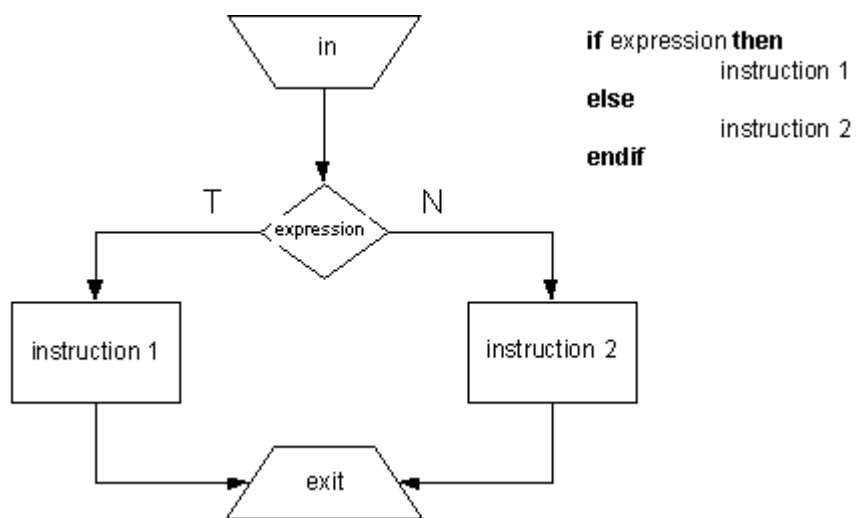
IF PORTB.0=0 **THEN** Add

goto Main

Add : W=W+1

End

More complex form of instruction is program branching with the ELSE part of instruction.



w var byte

Main :

IF PORTB.0=0 **THEN** Add

ELSE Subtract

ENDIF

goto Main

	<pre> Add : W=W+1 Subtract : W=W-1 End Same effect can be achieved directly : w var byte Main : IF PORTB.0=0 THEN W=W+1 ELSE W=W-1 ENDIF goto Main End </pre>
--	---

4.27 LCDOUT Prints data on LCD display

Syntax:	LCDOUT <i>Data {, Data...}</i>												
Description:	<p>LCDOUT sends the data to the LCD (<i>Liquid Crystal Display</i>). PIC BASIC supports various LCD models which have Hitachi 44780 controller or compatible one. LCD usually has either 14 or 16 pins for connection to a microcontroller. If there is character # before data, ASCII value of every data is sent to LCD. LCDOUT has the same modifiers as the instruction SEROUT2.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Modifier</th> <th style="text-align: left;">Sends</th> </tr> </thead> <tbody> <tr> <td>{I}{S} BIN{1..16}</td> <td>binary number</td> </tr> <tr> <td>{I}{S} DEC{1..5}</td> <td>decimal number</td> </tr> <tr> <td>{I}{S} HEX{1..4}</td> <td>hexadecimal number</td> </tr> <tr> <td>REP c/n</td> <td>character c repeated n times</td> </tr> <tr> <td>STR ArrayVar {n}</td> <td>n character string</td> </tr> </tbody> </table> <p>Before the first instruction is sent to LCD, program should wait for at least half a second for LCD to initialize.</p> <p>LCD display can be connected to PIC microcontrollers by either 4-bit or 8-bit bus. If 8-bit bus is used, all of 8 bits must be connected to the same port, while in the</p>	Modifier	Sends	{I}{S} BIN{1..16}	binary number	{I}{S} DEC{1..5}	decimal number	{I}{S} HEX{1..4}	hexadecimal number	REP c/n	character c repeated n times	STR ArrayVar {n}	n character string
Modifier	Sends												
{I}{S} BIN{1..16}	binary number												
{I}{S} DEC{1..5}	decimal number												
{I}{S} HEX{1..4}	hexadecimal number												
REP c/n	character c repeated n times												
STR ArrayVar {n}	n character string												

case of 4-bit bus all 4 bits must be either in the upper or the lower part of byte. R/W line should be connected to ground if LCD is used only for data display. PIC BASIC assumes that LCD is connected to specific pins if DEFINE directives do not say otherwise. Default is 4-bit bus with lines DB4-DB7 connected to RA0-RA3, RS pin connected to RA4 and E pin connected to RB 3. Also, it is assumed that LCD is 2x16. For changing any of the default settings, appropriate DEFINE directives can be used.

If LCD is connected to some other microcontroller lines it has to be defined with DEFINE directives, as shown in the following example.

```
DEFINE LCD_DREG  PORTB  ' port selection
DEFINE LCD_DBIT  4      ' initial bit (0 or 4) selection in case of 4-bit bus

DEFINE LCD_RSREG PORTB  ' port Register select
DEFINE LCD_RSBIT 1      ' Register Select bit
DEFINE LCD_EREG  PORTB  ' Enable port
DEFINE LCD_EBIT  0      ' Enable bit
DEFINE LCD_BITS  4      ' bus size – 4 or 8 bits
DEFINE LCD_LINES 2      ' number of LCD lines
DEFINE LCD_COMMANDS 2000 ' command delay in microseconds
DEFINE LCD_DATAUS 50    ' data delay in microseconds
```

Definitions above define 2-line LCD on 4-bit bus on the upper 4 bits of microcontroller port D. Register Select (RS pin) is on PORTD.2 and Enable is on PORTD.3.

Every LCD controller is in charge of certain commands. Commands are sent by instruction: LCDOUT \$FE, \$Kod. List of commands is shown in table below.

	Command	Operation
	\$FE, 1	clear display
	\$FE, 2	Return home (beginning of the first line)
	\$FE, \$0C	Turn off cursor
	\$FE, \$0E	Underline cursor on
	\$FE, \$0F	Blinking cursor on
	\$FE, \$10	Shifting cursor left
	\$FE, \$14	Shifting cursor right
	\$FE, \$C0	set cursor to the beginning of the second line
	\$FE, \$94	set cursor to the beginning of the third line
	\$FE, \$D4	set cursor to the beginning of the fourth line
Example:	<pre> B0 var byte B1 var byte Main: lcdout \$FE, 1, "Hello" ' Clear display and print "Hello" lcdout \$FE, \$C0 ' switch to second line lcdout B0 ' Display the value of B0 lcdout #B1 ' Display the value of B1 in ASCII code Loop: goto Loop end </pre>	

4.28 LCDIN Reads data from LCD RAM

Syntax:	LCDIN {Address,} [Var{, Var...}]
Description:	<p>LCDIN reads the given address of LCD RAM and stores data into a variable. When using this instruction, LCD Read/Write line must be connected to microcontroller. In case when LCD is used for data printing exclusively, this line can be connected to a logical zero. DEFINE directives inform the program about port and pin which Read/Write line is connected to:</p> <pre> DEFINE LCD_RWREG PORTE ' LCD read/write port DEFINE LCD_RWBIT 2 ' LCD read/write bit on port </pre>
Example:	<pre> B0 var byte Main: lcdin \$40, B0 ' Read data from LCD location \$40 and store it into B0 Loop: goto Loop End </pre>

4.29 {LET} Puts the value of the expression into a variable

Syntax:	{LET} { <i>Var=Expression</i> }
Description:	LET instruction stores value of the expression into a variable. Expression can be a constant, variable or value of some other expression. Commonly, the optional command word LET is excluded.
Example:	<pre> let B0 = B1 * B2 + B3 B0 = B1 * B2 + B3 </pre> <p>The two expressions are identical. The latter expression is missing command word "let".</p>

4.30 LOOKDOWN Searches the table of constants

Syntax:	LOOKDOWN <i>Value, [Const {, Const...}], Var</i>
Description:	Instruction LOOKDOWN searches the list of constants and determines the presence of given value. If a given value matches some of the constants, index of the appropriate constant is stored into variable. If the first constant matches our given value, variable is set to zero. If the second constant from the list matches our given value, variable is set to one, etc. If our value isn't present in the list, variable remains unchanged. Constants list can consist of both numerical and character (string) values. Each character of a string is treated as a separate ASCII value of a constant.
Example:	<pre> B0 var byte B1 var byte B0=\$f Main: lookdown B0, ("01234567890ABCDEF"), B1 ' convert hexadecimal character from B0 to a decimal value and store it into variable B1 PORTB=B1 ' PRIKAZI number on port B diodes loop: goto loop End </pre>

4.31 LOOKDOWN2 Searches the table of constants/variables

Syntax:	LOOKDOWN2 <i>Search, {Test} [Value {, Value...}], Var</i>
Description:	<p>LOOKDOWN2 searches the list of values and determines the presence of given value. If “Search” value matches some of the “Value” values, index of the appropriate constant is stored into “Var”.</p> <p>If “Search” matches the first value of the list, “Var” set to zero. If it matches the second value of the list, “Var” is set to one, etc. If “Search” value isn’t present in the list, “Var” remains unchanged.</p> <p>Optional parameter “Test” is used for testing if “Search” value is greater or lesser than a certain value. If “>” is used, index of the first matching constant is stored to “Var”. List of values can consist of 16-bit numbers, characters or variables. Every character of a string is treated as a separate ASCII value of that character (arrays of variables cannot be used with LOOKDOWN2 instruction). LOOKDOWN2 generates the code about 3 times greater than LOOKDOWN instruction does. Thus, when searching the list consisting of 8-bit constants and strings, use of LOOKDOWN is prefferable.</p>
Example:	<p style="text-align: center;">lookdown2 W0, [512, 7680 1024], B0</p>

	<pre> Digit var byte ' value of digit to be displayed Mask var byte ' mask of digit to be displayed Main: for i=0 to 9 Digit=i Lookup Digit, [\$3F, \$06, \$5B, \$4F, \$66, \$6D, \$7D, \$07, \$7F, \$6F], Mask PORTB=Mask ' Send the mask of a digit to port B pause 500 ' delay allowing to see digits changing next i ' Increase i by one goto Main ' Repeat the whole program end </pre>
--	---

4.33 LOOKUP2 Gets value from the table of constants/variables

Syntax:	LOOKUP2 <i>Index, [Value {, Value...}], Var</i>
Description:	<p>Instruction LOOKUP2 can be used for reading values from the table of values by index. If “Index” equals zero, “Var” attains the value of the first element in the list. If “Index” equals one, “Var” attains value of the second element in the list, etc. If “Index” is equal or greater than number of elements in the Look-up table “Var” remains unchanged.</p> <p>List of values can consist of 16-bit numbers, characters or variables. Every character of a string is treated as a separate ASCII value of that character. Arrays of variables cannot be used with LOOKUP2 instruction. LOOKUP2 generates the code about 3 times greater than LOOKUP instruction does. Thus, when searching the list consisting of 8-bit constants and strings, use of LOOKUP is prefferable.</p>
Example:	<pre> lookup2 B0, [256, 1024], W0 For B0=0, W0 will have value of 256 For B0=1, W0 will have value of 1024 For B0=2,3,... W0 will remain unchanged </pre>

4.34 LOW Puts logical zero to output pin

Syntax:	LOW <i>Pin</i>
Description:	Sets specific pin to zero. Pins is automatically designated output. Same effect can be achieved with PORTB=0.
Example:	<pre> Main: low PORTB.7 ' Set RB7 to a low level Loop: goto Loop End </pre>

4.35 NAP Turns off the processor for a short period of time

Syntax:	NAP <i>period</i>																		
Description:	<p>Instruction sets PIC microcontroller to <i>low power mode</i> (state of low energy consumption) for a short period of time. During this "nap", energy consumption is minimized. Stated periods are just approximations because these values were taken from <i>watch dog</i> timer and depend on chip and temperature:</p> <table border="1" data-bbox="646 1182 1251 1536"> <thead> <tr> <th>Period</th> <th>Delay [ms]</th> </tr> </thead> <tbody> <tr><td>0</td><td>18</td></tr> <tr><td>1</td><td>36</td></tr> <tr><td>2</td><td>72</td></tr> <tr><td>3</td><td>144</td></tr> <tr><td>4</td><td>288</td></tr> <tr><td>5</td><td>576</td></tr> <tr><td>6</td><td>1152</td></tr> <tr><td>7</td><td>2304</td></tr> </tbody> </table>	Period	Delay [ms]	0	18	1	36	2	72	3	144	4	288	5	576	6	1152	7	2304
Period	Delay [ms]																		
0	18																		
1	36																		
2	72																		
3	144																		
4	288																		
5	576																		
6	1152																		
7	2304																		
Example:	<pre> Main: nap 7' take a nap for 2.304 seconds Loop: goto Loop End </pre>																		

4.36 OUTPUT Designates I/O pin as output

Syntax:	OUTPUT <i>pin</i>
Description:	Designates specified pin as output.
Example:	<pre> Main: output PORTB.7 ‘ Pin RB7 is output TRISB.0 = 0 ‘ Same effect as above Loop : goto Loop End </pre>

4.37 OWIN Receives data via *one-wire* communication

Syntax:	OWIN <i>Pin, Mode, [Var1, Var2...]</i>									
Description:	<p>Parameter “Pin” is a variable containing the microcontroller pin connected to the element which has one-wire communication.</p> <p>Parameter “Mode” is value defined by parameters of communication.</p> <table border="1" data-bbox="347 1182 1551 1406"> <thead> <tr> <th>"Mode" bit</th> <th>How it works</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1 = sending the reset signal ahead of data</td> </tr> <tr> <td>1</td> <td>1 = sending the reset signal after data</td> </tr> <tr> <td rowspan="2">2</td> <td>0 = 8-bit data</td> </tr> <tr> <td>1= 1-bit data</td> </tr> </tbody> </table> <p>Parameters “Var1” and “Var2” are variables for containing the read data.</p>	"Mode" bit	How it works	0	1 = sending the reset signal ahead of data	1	1 = sending the reset signal after data	2	0 = 8-bit data	1= 1-bit data
"Mode" bit	How it works									
0	1 = sending the reset signal ahead of data									
1	1 = sending the reset signal after data									
2	0 = 8-bit data									
	1= 1-bit data									
Example:	<pre> Temperature var byte Main: OWIN PORTC.0, 0, [Temperature] ‘ read the temp. PORTB=Temperature ‘ display temperature on port B diodes goto Main End </pre>									

4.38 OWOUT Transmits data via *one-wire* communication

Syntax:	OWOUT <i>Pin, Mode, [Var1, Var2...]</i>									
Description:	Parameter “Pin” is variable containing the microcontroller pin connected to the element which has one-wire communication.									
	Parameter “Mode” is value defined by parameters of communication.									
	<table border="1"> <thead> <tr> <th>“Mode” bit</th> <th>How it works</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>1 = sending the reset signal ahead of data</td> </tr> <tr> <td>1</td> <td>1 = sending the reset signal after data</td> </tr> <tr> <td rowspan="2">2</td> <td>0 = 8-bit data</td> </tr> <tr> <td>1 = 1-bit data</td> </tr> </tbody> </table>	“Mode” bit	How it works	0	1 = sending the reset signal ahead of data	1	1 = sending the reset signal after data	2	0 = 8-bit data	1 = 1-bit data
“Mode” bit	How it works									
0	1 = sending the reset signal ahead of data									
1	1 = sending the reset signal after data									
2	0 = 8-bit data									
	1 = 1-bit data									
	Parameters “Var1” and “Var2” are variables for containing the read data.									
Example:	<pre> Main : OWOUT PORTC.0, 1, [\$CC, \$BE] ‘ sends reset signal and 2 values afterwards goto Main End </pre>									

4.39 PAUSE Pause (in milliseconds)

Syntax:	PAUSE <i>Period (in milliseconds)</i>
Description:	Instruction pauses the program for “Period” milliseconds. Period is 16-bit, allowing delay to be as long as 65 535ms (a bit over a minute). Unlike other delay instructions (NAP and SLEEP), PAUSE does not put the microcontroller to <i>low power mode</i> . Thus, PAUSE consumes more energy, but gets more accurate timing (it has precision of a system clock).
Example:	<pre> TRISB = 0 Main: PORTB = 255 pause 1000 ‘ Delay execution of next instruction line for 1 sec. PORTB = 0 pause 2000 ‘ Delay execution of next instruction line for 2 sec. goto Main End </pre>

4.40 PAUSEUS Pause (in microseconds)

Syntax:	PAUSEUS <i>Period (in milliseconds)</i>																
Description:	<p>PAUSEUS stops the program for “Period” milliseconds. Period is 16-bit (WORD), allowing delay to be as long as 65 535ms (a bit over a minute). Unlike other delay instructions (NAP and SLEEP), PAUSE does not put the microcontroller to <i>low power mode</i>. PAUSEUS consumes more energy than PAUSE, but gets much more accurate timing. Minimal delay of PAUSEUS depends on the crystal frequency.</p> <table border="1" data-bbox="646 1126 1251 1438"> <thead> <tr> <th>OSC</th> <th>Minimal delay</th> </tr> </thead> <tbody> <tr> <td>3 (3.58)</td> <td>20 us</td> </tr> <tr> <td>4</td> <td>24 us</td> </tr> <tr> <td>8</td> <td>12 us</td> </tr> <tr> <td>10</td> <td>8 us</td> </tr> <tr> <td>12</td> <td>7 us</td> </tr> <tr> <td>16</td> <td>5 us</td> </tr> <tr> <td>20</td> <td>3 us</td> </tr> </tbody> </table> <p>PAUSEUS works with default 4MHz crystal frequency. If frequency differs from default it is</p>	OSC	Minimal delay	3 (3.58)	20 us	4	24 us	8	12 us	10	8 us	12	7 us	16	5 us	20	3 us
OSC	Minimal delay																
3 (3.58)	20 us																
4	24 us																
8	12 us																
10	8 us																
12	7 us																
16	5 us																
20	3 us																
Example:	<pre> TRISB = 0 Main: PORTB = 255 pauseus 100 ‘ Delay execution of next instruction line for 100 microsec PORTB = 0 pauseus 3450 ‘ Delay execution of next instruction line for 3.450 ms goto Main End </pre>																

Example:	<pre> B0 var byte skala var byte Main : FOR skala=1 TO 255 pot PORTA.0, skala, B0 ' read value of potentiometer on RA0 IF B0>253 Then Over NEXT skala Over : PORTB=skala ' display value of the scale on port B diodes goto Main End </pre>
-----------------	---

4.42 PULSIN Measures impulse duration on input pin

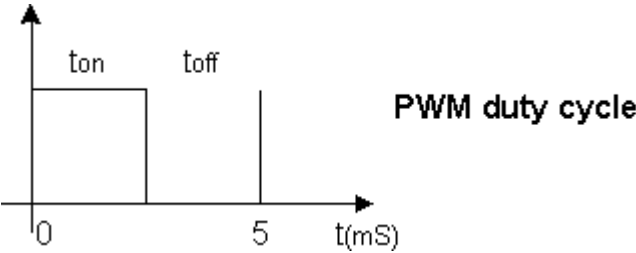
Syntax:	PULSIN <i>Pin, Level, Var</i>
Description:	<p>Instruction measures impulse duration with 10us resolution (when 4MHz oscillator is used) on a given pin. If level is zero it measures duration of low impulse and if level is one it measures duration of high impulse. Measured value of duration is put to variable "Var". Measuring can last from 10 to 65 535 microseconds for 16-bit variables. If impulse doesn't appear at all or it's duration is too long to be measured variable is set to zero.</p> <p>In case of 8-bit variable only lower 8 bits of a 16-bit word are used. Resolution depends on oscillator frequency. 4MHz oscillator has 10us resolution, while 20MHz oscillator has 2us resolution.</p>
Example:	<pre> W0 var word Main : pulsln PORTB.0, 1, W0 ' Measures high impulse on RB0 pin with 10us resolution and puts ' it to variable W0 goto Main End </pre>

4.43 PULSOUT Generates impulse on output pin

Syntax:	PULSOUT <i>Pin, Period</i>
Description:	<p>Instruction generates impulse of specific duration in tens of microseconds (when 4MHz oscillator is used) on a pin. Impulse is generated by double change of level on a pin, so that former state of pin defines polarity of an impulse. Chosen pin is automatically designated output.</p> <p>Resolution depends on oscillator frequency. 4MHz oscillator has 10us resolution, while 20MHz oscillator has 2us resolution</p>

Example:	Main : <pre> pulsout PORTB.7, 100 ' Generate 1ms impulse to RB7 pin goto Main End </pre>
-----------------	---

4.44 PWM Generates PWM signal on pin

Syntax:	PWM <i>Pin, Ratio, Cycle</i>
Description:	<p>Instruction sends PWM (Pulse-Width Modulation) impulses Ratio to pin defined with parameter "Pin" (for each PWM signal, cycle goes from 0 (0%) to 255 (100%)). This PWM cycle repeats itself for number of times defined with "Cycle" parameter. Pin direction is set to output just before generating PWM impulse and is set back to input afterwards.</p> <p>Cycle duration depends on the oscillator used. In case of 4MHz oscillator cycle duration is 5ms, while in case of 8MHz oscillator cycle duration is 1ms. Instruction PWM allows simple R/C circuit to be used for generating DC voltage like a simple D/A converter.</p> 
Example:	Main : <pre> pwm PORTB.7, 127, 100 ' Send pwm cycle with 50% of signal (ON) to RB7 goto Main End </pre>

4.45 RANDOM Generates pseudo-random number

Syntax:	RANDOM <i>Variable</i>
Description:	Instruction RANDOM stores pseudo-random number to variable. Variable should be 16-bit.
Example:	W0 var word Main : <pre> random W0 ' Put random number to variable W0 lcdout #W0 ' Display random number on LCD goto Main End </pre>

4.46 RCTIME Measures impulse duration on pin (similar to PULSIN)

Syntax:	RCTIME <i>Pin, State, Variable</i>
Description:	<p>RCTIME measures time period during which "pin" remains in a certain state. If the state remains unchanged variable is set to zero. RCTIME can be used for reading potentiometer or some other resistive element based on the time necessary for filling RC constant. Typical resistance measured is within 5K~50K range.</p> <p>Resolution depends on oscillator frequency. 4MHz oscillator has 10us resolution, while 20MHz oscillator has 2us resolution.</p>
Example:	<pre> W0 var word Main : low PORTA.0 ' Discharge the condenser pause 10 ' Discharging lasts for 10ms rctime PORTA.0, 0, W0 ' Measure duration of charging lcdout #W0 ' Display value of W0 on LCD goto Main End </pre>

4.47 READ Reads one byte of data from data EEPROM

Syntax:	READ <i>Address, Variable</i>
Description:	<p>Instruction READ reads data from internal EEPROM memory from the specified address and stores the result to "Variable". This instruction can only be used with PIC microcontrollers which have EEPROM built in the chip. If microcontroller is supplied with external EEPROM, instruction I2CREAD should be used instead.</p>
Example:	<pre> B0 var byte W var word Main : READ 5, B0 ' read data from EEPROM, address 5 and put it to variable B0 READ 6, W.BYTE0 ' load 16-bit data READ 7, W.BYTE1 ' from addresses 6 and 7 to variable W Loop: goto Loop End </pre>

4.48 READCODE Reads 2 bytes (word) of program code from the address

Syntax:	READCODE <i>Address, Variable</i>
Description:	READCODE reads program code from a given address and puts the result to 16-bit variable. PIC16F87X microcontroller family allows reading and writing program code while microcontroller is in operation.
Example:	<pre> Wo var word Main : readcode 100, W0 ' load data from program FLASH memory, address 100 to var. W0 Loop : goto Loop End </pre>

4.49 RETURN Return from the subroutine

Syntax:	RETURN
Description:	Instruction RETURN executes return from the program routine which program jumped onto via GOSUB instruction.
Example:	<pre> Main : gosub portb_on ' call a subroutine init_ram Loop : goto Loop portb_on: PORTB=\$FF ' Light all port B diodes return ' return from subroutine End </pre>

4.50 REVERSE Changes pin orientation

Syntax:	REVERSE <i>Pin</i>
Description:	Instruction REVERSE inverts orientation of a specified pin. If pin is input, REVERSE changes it to output and vice versa.
Example:	<pre> Main : reverse PORTA.0 ' Change orientation of RA0 pin Loop : goto Loop End </pre>

4.51 SELECT-CASE Conditional multiple program branching

Syntax:	<pre>SELECT CASE Var CASE Expression1 {, Expression} Instructions... CASE Expression2 {, Expression} Instructions... CASE Expression3 {, Expression} Instructions... CASE ELSE Instructions... END SELECT</pre>
Description:	Although conditional SELECT-CASE branching can be made with multiple IF-THEN instructions, it is easier and more sensible to use this instruction in certain situations. Instruction allows "Expression" to be a constant, one of the constants or a comparison to a certain constant.
Example:	W var byte 60 B var byte

```

PORTB=B

Pause 3000

CASE ELSE

B=FF

PORTB=B

Pause 3000

END SELECT

NEXT W

END

```

The example above cycles numbers from 0 to 9 in the SELECT CASE branching. If W equals zero port B diodes will take value of 1. If W equals 1, 2 or 3 port B diodes will take value of 2.

If W equals 4 or 5 port B diodes will take value of 255 because 4 and 5 haven't been defined - therefore, value from CASE ELSE part of the instruction is taken.

If W is greater than 5, port B diodes will take value of 3.

4.52 SERIN Asynchronous serial input (like with BS1)

Syntax:	SERIN <i>Pin, Mode, {Timeout, Label}, {[Qual...], }{Item...}</i>		
Description:	SERIN receives one or more values on a specified pin "Pin" using the standard asynchronous format 8N1 (8 data bits, no parity check and one 'stop' bit).		
	Instead of numerical value ranging from 0 to 15, Mode can be a name if "modedefs.inc" library is included ahead.		
	Mode	Mode number	Baud rate
	T2400	0	2400
	T1200	1	1200
	T9600	2	9600
	T300	3	300
	N2400	4	2400
	N1200	5	1200
	N9600	6	9600
	N300	7	300
			True
			Inverted
	SERIN instruction can include label (parameter "Label") which the program will jump onto if there is no data received during the specified time period (parameter "Timeout" - default value is 1ms).		

	<p>There can be qualifier within brackets [] ahead of every data. SERIN must receive these bytes in correct order before receiving data words. If any received byte doesn't match next byte's qualifier, marking process begins anew - next received byte is compared to the first on the qualifying list. Qualifying content can be a constant, variable or character string. Every character in a string is treated as a separate qualifier.</p> <p>When qualifiers are set, SERIN tries to save data to variables. If there is character # ahead of variable SERIN converts decimal value to ASCII and stores the result in that variable.</p> <p>SERIN works with 4MHz oscillator by default. In order to achieve certain transfer rate with other oscillators, it is necessary to use appropriate "DEFINE Osc" directive.</p>
<p>Example:</p>	<p>B0 var byte</p> <p>Main :</p> <p> ' Wait for character "A" to be received on serial line on pin RB0 and store next</p> <p> ' received character to variable</p> <p> serin PORTB.0, N2400, ["A"], B0</p> <p>variable B0</p> <p> lcdout B0 ' Display content of B0 on LCD</p> <p>Loop : goto Loop</p> <p> End</p>

4.53 SERIN2 Asynchronous serial input (like with BS2)

<p>Syntax:</p>	<p>SERIN2 Pin{FlowPin}, Mode, {ParityLabel}, {Timeout, Label}, [Item...]</p>
<p>Description:</p>	

Mode	Mode number	Baud rate	State
T2400	0	2400	True
T1200	1	1200	
T9600	2	9600	
T300	3	300	
N2400	4	2400	Inverted
N1200	5	1200	
N9600	6	9600	
N300	7	300	

Optional "FlowPin" can be used to prevent eventual data loss in high speed transfers. If used, "FlowPin" is automatically set to regular state (depends on polarity from Mode parameter - table above) in order to allow transfer of every character.

Mode can be used for defining baud rate and serial transfer parameters. Lower 13 bits determine baud rate. Bit 13 selects (non)parity check. Bit 14 selects inverted or true level, while bit 15 is unused. Transfer rate determines bit duration in microseconds. To determine bit duration for a given transfer rate, following equation is used :

$$(1000000 / \text{baud rate}) - 20$$

Table below shows several standard transfer rates:

Baud Rate	bit 0-12
300	3313
600	1646
1200	813
2400	396
4800	188
9600	84
19200	32

Bit 13 enables parity check if bit 13 equals 1 and disables it for 0. For bit13 = 0 transfer format is 8N1. In case that parity check is needed, following directive should be used :

```
DEFINE SER2_ODD      1
```

Bit 14 selects data level of flow control pins. If bit 14 equals 0 data is received true, while bit14 = 1 receives inverted data.

Some of standard settings include :

Mode = 84 (9600 baud, no parity check, true)

Mode = 16780 (2400 baud, no parity check, inverted)

Mode = 27889 (300 baud, parity check, inverted)

Optional label "ParityLabel" specifies label which program jumps onto if transfer error occurs (this label makes sense only if parity bit is on).

"Timeout" and "Label" allow program to proceed from designated label if there is no data in specified time period. Waiting time is expressed in milliseconds.

DEFINE directive allows transfer of data with size greater than 8, that is 7 with parity check. SER2_BITS allows transfer of data ranging from 4 to 8 bits.

SERIN2 supports many different data modifiers that can be combined to allow various input data formats.

Modifier	How it works
BIN{1..16}	Takes binary digits
DEC{1..5}	Takes decimal digits
HEX{1..4}	Takes hexadecimal digits
SKIP n	Skips next n characters
STR ArrayVar{n}{c}	Takes the sequence of n characters that ends with the character c (optional)
WAIT ()	waits for character sequence
WAITSTR ArrayVar{n}	waits for a string

If prefix BIN is used ahead of variable, ASCII character in binary value of variable will be received. For example, if we write BIN B0 and received value is "1000" B0 will take value of 8.

If prefix DEC is used ahead of variable, ASCII character in decimal value of variable will be received. For example, if we write DEC B0 and received value is "123" B0 will take value of 123.

If prefix HEX is used ahead of variable, ASCII character in hexadecimal value of variable will be received. For example, if we write HEX B0 and received value is "FE" B0 will take value of 254.

Key word SKIP followed by a number enables that many characters from input row

	<p>to be skipped. For example, SKIP 4 would skip 4 characters.</p> <p>If key word STR is followed by variable of string type, number "n" and optional ending char, character string will be received. String length is defined with "n" or with appearing of final element of a string.</p> <p>Data bytes received usually go after one or more identification bytes. Identification bytes come within small brackets after WAIT. It means that the sequence of received bytes must match the sequence of identification bytes. Otherwise, if one of received bytes doesn't match following byte in identifier sequence, identification process starts anew - next received byte is compared to the first identification byte.</p> <p>Identification byte can be a constant, variable or array of constants. In the last case, every constant is treated as a separate identifier.</p> <p>WAITSTR is used in a similar way as WAIT, except for the fact that the key is character string instead of byte sequence.</p> <p>Instruction SERIN2 assumes that microcontroller clock works at 4MHz. In case of different oscillator it is necessary to make adjustment with following directive :</p> <p style="text-align: center;">DEFINE OSC.</p>
<p>Example:</p>	<p style="text-align: center;">serin2 PORTB.0, 16780, [wait("A"), B0]</p> <p>wait for character "A" to be received to RB0 pin and store next received character to variable B0.</p> <p style="text-align: center;">serin2 PORTB.0, 84, [skip 2, dec 4, B0]</p> <p>Skip 2 characters and receive next 4 decimal numbers.</p>

4.54 SEROUT Asynchronous serial output (like with BS1)

<p>Syntax:</p>	<p>SEROUT <i>Pin, Mode, [Item{, Item...}]</i></p>
<p>Description:</p>	<p>SERIN sends one or more values to a specified pin "Pin" using the standard asynchronous format 8N1 (8 data bits, no parity check and one 'stop' bit). Transfer modes ("Mode") include :</p>

Mode	Mode number	Baud Rate	State
T2400	0	2400	Driven True
T1200	1	1200	
T9600	2	9600	
T300	3	300	
N2400	4	2400	Driven Inverted
N1200	5	1200	
N9600	6	9600	
N300	7	300	
OT2400	8	2400	Open True
OT1200	9	1200	
OT9600	10	9600	
OT300	11	300	
ON2400	12	2400	Open Inverted
ON1200	13	1200	
ON9600	14	9600	
ON300	15	300	

Instead of numerical value from 0 to 15, Mode can be a name if "modedefs.inc" library is included ahead.

If there is character # ahead of variable SEROUT converts decimal value to ASCII and sends it. For example, if B equals 34 then #B sends '3' and '4'.

SEROUT works with 4MHz oscillator by default. In case of different oscillator it is necessary to make adjustment with following directive : DEFINE OSC.

In cases of slower receiving device, it is necessary to wait for a certain amount of time when sending next data. DEFINE directive enables delay ranging from 1 to 65 535 microseconds (0.001 to 65.535 milliseconds) between sending 2 characters.

```
DEFINE CHAR_PACING 1000 ' 1ms delay between 2 chars
```

Example:

```
B0 var byte

Main:

    B0 = 25

    serout PORTA.3, N2400, [#B0, 13] ' Send ASCII value of B0 and constant
13 to RA3 via serial line
```

	<pre> Loop : goto Loop End </pre>
--	--

4.55 SEROUT2 Asynchronous serial output (like with BS2)

Syntax:	SEROUT2 <i>Pin</i> { <i>FlowPin</i> }, <i>Mode</i> , { <i>Pace</i> , }, { <i>Timeout</i> , <i>Label</i> }, [<i>Item</i> ...]																
Description:	<p>SEROUT2 sends one or more values to pin determined with parameter "Pin". "Pin" is automatically designated output, while optional "FlowPin" is designated input. Optional "FlowPin" is used for indicating data loss at receiver. Level of permission depends on data transfer mode determined by "Mode".</p> <p>Optional parameters "Timeout" and "Label" allow program to proceed and in case that "FlowPin" doesn't change to state of transfer allowed in a given time period. Wait time "Timeout" is entered in milliseconds.</p> <p>In some cases transfer rate of SEROUT2 can be too high for receiving device. Then, it is more efficient to set delay between 2 characters using the "pace" parameter instead of using extra pin as "FlowPin". In this way, it is possible to provide sufficient delay when sending data.</p> <p>Mode is used to determine baud rate and important parameters of serial transfer. Lower 13 bits determine baud rate. Bit 13 selects (non)parity check. Bit 14 selects inverted or true level, while bit 15 is used to determine if connection is currently in transfer or not. Transfer rate determines bit duration in microseconds. To determine bit duration for a given transfer rate, following equation is used :</p> $(1000000 / \text{baud rate}) - 20$ <p>Table below shows several standard transfer rates:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Baud Rate</th> <th>bits 0-12</th> </tr> </thead> <tbody> <tr> <td>300</td> <td>3313</td> </tr> <tr> <td>600</td> <td>1646</td> </tr> <tr> <td>1200</td> <td>813</td> </tr> <tr> <td>2400</td> <td>396</td> </tr> <tr> <td>4800</td> <td>188</td> </tr> <tr> <td>9600</td> <td>84</td> </tr> <tr> <td>19200</td> <td>32</td> </tr> </tbody> </table> <p>If set, bit 13 enables parity check. Transfer format is standard 8N1 (8 data bits, no parity check, one 'stop' bit) and for bit13 = 1 format is 7E1 (7 data bits, parity bit and one 'stop' bit).</p> <p>Bit 14 selects data level of "flow control" pins. If bit 14 equals 0 data is received true, while bit14 = 1 receives inverted data (this can used to avoid installation of</p>	Baud Rate	bits 0-12	300	3313	600	1646	1200	813	2400	396	4800	188	9600	84	19200	32
Baud Rate	bits 0-12																
300	3313																
600	1646																
1200	813																
2400	396																
4800	188																
9600	84																
19200	32																

RS232 communication driver - MAX232).

Bit 15 determines if data pin is still connected (bit15 = 0) or disconnected from data transfer line. This option is useful in case of connecting multiple devices to common serial line.

Some of standard settings include :

Mode = 84 (9600 baud, no parity check, true)

Mode = 16780 (2400 baud, no parity check, inverted)

Mode = 27889 (300 baud, parity check, inverted)

DEFINE directive SER2_BITS allows transfer of data with size different than 8 (7 with parity check). SER2_BITS allows transfer of data ranging from 4 to 8 bits. Default value is 8 bits.

SEROUT2 supports many different data modifiers that can be combined in order to allow various input data formats.

Modifier	How it works
{I}{S} BIN{1..16}	Sends binary digits
{I}{S} DEC{1..16}	Sends decimal digits
{I}{S} HEX{1..16}	Sends hexadecimal digits
REP c\n	Sends character "c", "n" times
STR ArrayVar\n{c}	Sends of "n" characters sequence that ends with the character "c" (optional)

If prefix BIN is used ahead of variable, ASCII character in binary value of variable will be sent. For example, if we write BIN B0 and B0 = 8, bits 1000 will be sent serial.

If prefix DEC is used ahead of variable, ASCII character in decimal value of variable will be sent. For example, if we write DEC B0 and B0 = 123, data "123" will be sent.

If prefix HEX is used ahead of variable, ASCII character in hexadecimal value of variable will be sent. For example, if we write HEX B0 and B0 = 254, SEROUT2 will send "FE".

REP followed by a character and a number of repeating provides more compact form of writing long strings of same characters. For example, REP "0"\4 stands for "0000"

STR followed by variable of string type and an optional numerical parameter "count" executes sending of character string. String length is determined by "count" or by appearance of character "0" in a string.

Optional parameters can be used ahead or behind BIN, DEC and HEX. In case

that "I" is used ahead of any of these, output data will begin with %@, #@ or \$@ in order to mark current value as binary, decimal or hexadecimal.

In case that "S" (signed) is used ahead of BIN, DEC or HEX , output data will begin with "-" if highest data bit is set to 1. This allows transfer of negative values. You should bear in mind, though, that all mathematical and comparison operations work with unsigned numbers. Still, unsigned numbers arithmetic allows signed values as results. For example, in case of B0 = 9 - 10, DEC B0 gets value of "255", whereas SDEC B0 sends "1" after the transfer of the highest bit.

BIN, DEC and HEX can be followed by a number. It is common practice to write numerical data in exact number of digits needed, so that leading zeros are erased and not sent. In case that BIN, DEC and HEX are followed by a number, SEROUT2 will always send that exact number of data, adding leading zeros if needed. For example, BIN6 8 sends BIN "001000", while BIN2 8 sends "00". All these modifies can be used simultaneously (i.e. ISDEC4 B0).

Instruction SEROUT2 assumes that microcontroller clock works at 4MHz. In case of different oscillator it is necessary to make adjustment with following directive :

DEFINE OSC

Example:	<p>B0 = 25</p> <p>SEROUT2 PORTA.3, 16780, [DEC B0, 10]</p> <p>Send decimal value of variable B0 and "LineFeed" via serial line (2400 bauds) to pin RA3.</p> <p>B0 = 25</p> <p>SEROUT2 PORTA.1, 84, ["B0=", IHEX4 B0]</p> <p>Send string "B=" and 4-character hexadecimal value of variable B0 to RA1 at 9600 bauds.</p>
-----------------	---

4.56 SHIFIN Synchronous serial input

Syntax:	SHIFIN <i>DataPin, ClockPin, Mode, [Var{Bits}...]</i>																		
Description:	<p>Instruction SHIFIN shifts receiving bits on a given pin in synchrony with "ClockPin" frequency signal and stores them to variable. "Var\Bits" optionally specifies the number of bits to be shifted. If nothing is specified, default number of bits is 8.</p> <p>Depending on shifting direction (from MSB to LSB or vice versa) various transfer modes can be defined.</p> <p>Transfer modes Mode are defined within MODEDEFS.BAS library. To use them, it is necessary to include mentioned library at the beginning of the program with : Include "modedefs.bas"</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">"Mode"</th> <th style="text-align: center;">Mode</th> <th style="text-align: left;">Operation</th> </tr> </thead> <tbody> <tr> <td>MSBPRE</td> <td style="text-align: center;">4</td> <td>First, the highest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.</td> </tr> <tr> <td>LSBPRE</td> <td style="text-align: center;">1</td> <td>First, the lowest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.</td> </tr> <tr> <td>MSBPOST</td> <td style="text-align: center;">5</td> <td>First, the highest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.</td> </tr> <tr> <td>LSBPOST</td> <td style="text-align: center;">3 6</td> <td>First, the lowest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.</td> </tr> <tr> <td></td> <td style="text-align: center;">7</td> <td>First, the highest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.</td> </tr> </tbody> </table>	"Mode"	Mode	Operation	MSBPRE	4	First, the highest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.	LSBPRE	1	First, the lowest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.	MSBPOST	5	First, the highest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.	LSBPOST	3 6	First, the lowest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.		7	First, the highest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.
"Mode"	Mode	Operation																	
MSBPRE	4	First, the highest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.																	
LSBPRE	1	First, the lowest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.																	
MSBPOST	5	First, the highest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.																	
LSBPOST	3 6	First, the lowest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.																	
	7	First, the highest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.																	

		inactive on a logical zero.
	4	First, the highest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.
	5	First, the lowest bit is shifted. Data is read ahead of sending clock. Clock is inactive on a logical one.
	6	First, the highest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.
	7	First, the lowest bit is shifted. Data is read after sending clock. Clock is inactive on a logical one.
<p>Shifting frequency is about 50KHz, depending on oscillator used. Active state lasts for at least 2 microseconds. Using the directive DEFINE enables additional delay (up to 65.535 miliseconds) for slowing down the clock.</p> <pre>DEFINE SHIFT_PAUSEUS 100 ' Slowing down the clock for additional 100ms</pre>		
Example:	<p>shiftin Data, Clock, MSBPRES, [RxData]</p> <p>Sends the contents of input SHIFT register to variable RxData so that the first bit is MSB.</p>	

4.57 SHIFTOUT Synchronous serial output

Syntax:	SHIFTOUT <i>DataPin, ClockPin, Mode, [Var\Bits]...</i>																	
Description:	<p>Instruction SHIFTOUT shifts bits of variable "Var" on a given pin in synchrony with "ClockPin" frequency signal. "Var\Bits" optionally specifies the number of bits to be shifted. If nothing is specified, default number of bits is 8.</p> <p>Transfer modes Mode are defined within MODEDEFS.BAS library. To use them, it is necessary to include mentioned library at the beginning of the program with : include modedefs.bas</p> <p>Shifting frequency is about 50KHz, depending on oscillator used. Active state lasts for at least 2 microseconds. Using the directive DEFINE enables additional delay (up to 65.535 miliseconds) for slowing down the clock.</p> <pre>DEFINE SHIFT_PAUSEUS 100 ' Slowing the clock for additional 100ms</pre> <table border="1"> <thead> <tr> <th>"Mode"</th> <th>Mode number</th> <th>Operation</th> </tr> </thead> <tbody> <tr> <td>LSBFIRST</td> <td>0</td> <td>First, the lowest bit is shifted.. Clock is inactive on a logical zero.</td> </tr> <tr> <td>MSBFIRST</td> <td>1</td> <td>First, the highest bit is shifted.. Clock is inactive on a logical zero.</td> </tr> <tr> <td></td> <td>4</td> <td>First, the lowest bit is shifted.. Clock is inactive on a logical one.</td> </tr> <tr> <td></td> <td>5</td> <td>First, the highest bit is shifted.. Clock is inactive on a logical one.</td> </tr> </tbody> </table>			"Mode"	Mode number	Operation	LSBFIRST	0	First, the lowest bit is shifted.. Clock is inactive on a logical zero.	MSBFIRST	1	First, the highest bit is shifted.. Clock is inactive on a logical zero.		4	First, the lowest bit is shifted.. Clock is inactive on a logical one.		5	First, the highest bit is shifted.. Clock is inactive on a logical one.
"Mode"	Mode number	Operation																
LSBFIRST	0	First, the lowest bit is shifted.. Clock is inactive on a logical zero.																
MSBFIRST	1	First, the highest bit is shifted.. Clock is inactive on a logical zero.																
	4	First, the lowest bit is shifted.. Clock is inactive on a logical one.																
	5	First, the highest bit is shifted.. Clock is inactive on a logical one.																
Example:	<p>B0 var byte</p> <p>B1 var byte</p> <p>W0 var byte</p> <p>Main :</p> <pre>shiftout PORTA.0, PORTA.1, MSBFIRST, [B0, B1]</pre>																	

	<ul style="list-style-type: none"> ' Sends the contents of variables B0 and B1 to output SHIFT register so that the first ' transferred bit is MSB <p>shiftout PORTA.0, PORTA.1, MSBFIRST, [W0\4]</p> <ul style="list-style-type: none"> ' Sends 4 bits of variable W0 so that the first transferred bit is MSB <p>Loop : goto Loop</p> <p>End</p>
--	--

4.58 SLEEP Turns off the processor for a given time period

Syntax:	SLEEP <i>Period</i>
Description:	Instruction puts the microcontroller to a state of low energy consumption for "Period" of seconds. "Period" is a 16-bit value allowing maximal delay of 65 535 seconds (about 18h). SLEEP uses the watchdog timer (WDT) with granularity about 2.3 seconds. RC oscillator is less temperature stable than system clock, making WDT somewhat less accurate.
Example:	<p>Main :</p> <p style="padding-left: 40px;">sleep 60 ' Go to low power mode for next 60 sec</p> <p>Loop : goto Loop</p> <p>End</p>

4.59 SOUND Generates sound or white noise on a given pin

Syntax:	SOUND <i>Pin, (Note, Duration{, Note, Duration})</i>
Description:	<p>Instruction generates tone and/or noise on a given pin. For Note=0 there is no sound generated. If Note falls within range of 1-127 tones are generated, while range of 128-255 generates noise.</p> <p>Tones and noises are sorted in an ascending fashion (1 and 128 are the lowest frequencies, 127 and 255 are the highest). Duration ranges from 0 to 255 and defines sound duration in 12ms increments ("Note" and "Duration" don't have to be constants).</p> <p>Sound is being sent to output in form of sequence of TTL rectangle impulses. Thanks to the outstanding I/O features of PIC microcontrollers, a speaker can be driven directly through electrolytical capacitor. Piezo speakers can be driven directly.</p>
Example:	<p>Main :</p> <p style="padding-left: 40px;">sound PORTB.7, (100, 10, 50, 10) ' Sends 2 sounds in sequence to pin RB7</p> <p>Loop : goto Loop</p> <p>End</p>

4.60 STOP Stops the program execution

Syntax:	STOP
----------------	-------------

Description:	Instruction stops the program execution by commencing the infinite loop. This instruction does not put the microcontroller to low power mode.
Example:	<pre> Main : STOP ' Stop the program execution in this line Loop : goto Loop End </pre>

4.61 SWAP Exchanges values of two variables

Syntax:	SWAP <i>Variable1, Variable1</i>
Description:	Instruction SWAP exchanges values of two variables. It can be used with variables of bit, byte and word types. SWAP can be used with strings, but only with those that have constant indexes.
Example:	<pre> B0 var byte B1 var byte temp var byte Main : temp = B0 B0 = B1 B1 = temp ' classical way to do it swap B0, B1 ' ...and easier way to do it Loop : goto Loop End </pre>

4.62 TOGGLE Inverts pin states

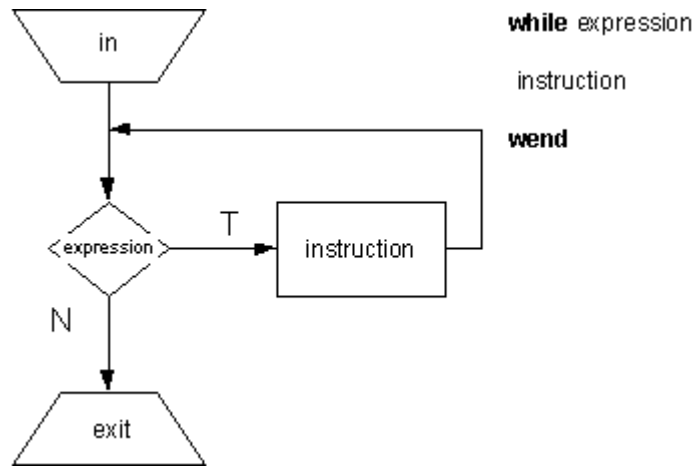
Syntax:	TOGGLE <i>Pin</i>
Description:	Instruction inverts state of a specified pin. "Pin" is automatically designated output.
Example:	<pre> Main : low PORTB.0 ' Set the state of pin RB0 to low level as starting condition toggle PORTB.0 ' Change state of pin RB0 to high level Loop : goto Loop End </pre>

4.63 WRITE Writes byte of data to data EEPROM

Syntax:	WRITE <i>Address, Value</i>
Description:	<p>Instruction writes "Value" to a specified address of EEPROM. WRITE can only be used with PIC microcontrollers that have EEPROM built in chip.</p> <p>If 2-byte variable is being stored, two bytes are written in sequence :</p> <pre>WRITE Address, Variable.BYTE0 WRITE Address, Variable.BYTE1</pre>
Example:	<pre>B0 var byte Main : B0 = \$EA write 5, B0 ' Writes value \$EA to location 5 of EEPROM Loop : goto Loop End</pre>

4.64 WRITECODE Writes two bytes (word) of data to program memory

Syntax:	WRITECODE <i>Address, Value</i>
----------------	--



Example:

i Var byte

Main :

i = 1

WHILE i < 10 ' when i reaches 10 program stops and port B has value of 9

i = i + 1

PORTB = i

Pause 1000

WEND

goto Main

End

Chapter 5

SAMPLE PROGRAMS FOR SUBSYSTEMS WITHIN THE MICROCONTROLLER

[Introduction](#)

[5.1 Using the interrupt mechanism](#)

[5.2 Using the internal AD converter](#)

[5.3 Using the TMR0 timer](#)

[5.4 Using the TMR1 timer](#)

[5.5 Using the PWM subsystem](#)

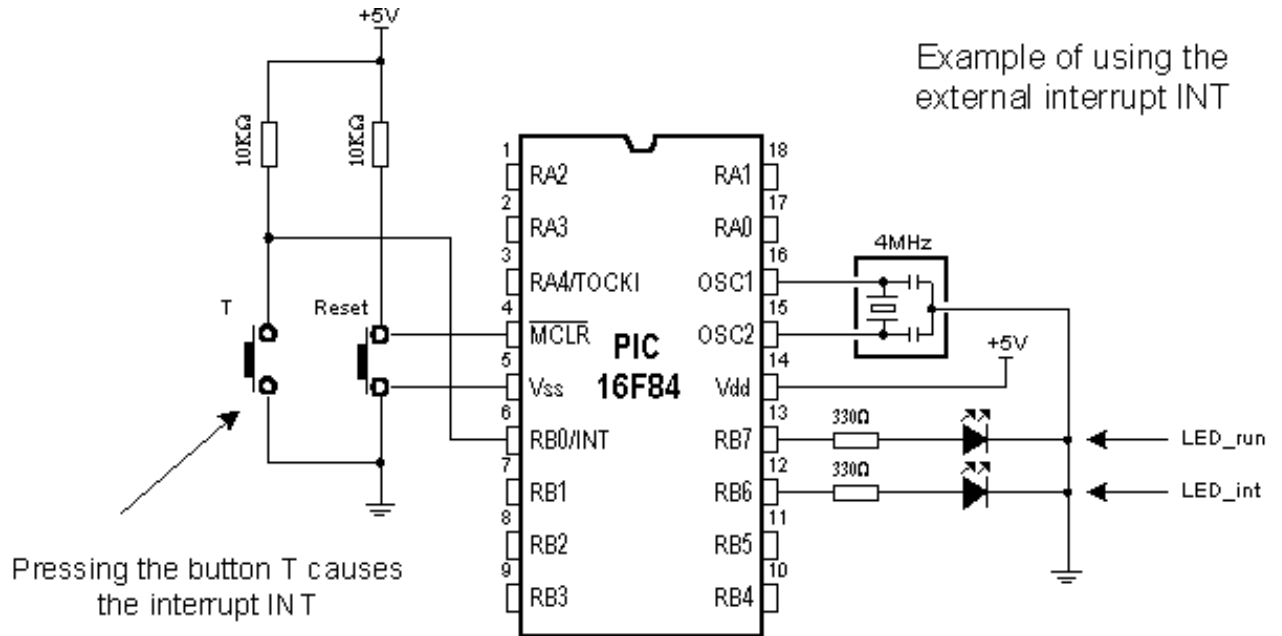
[5.6 Using the hardware UART subsystem \(RS-232 communication\)](#)

Introduction

Every microcontroller is supplied with at least a few integrated subsystems - commonly, these include timers, interrupt mechanisms and AD converters. More powerful microcontrollers can command greater number of built-in subsystems. Some of frequently encountered systems are detailed in this chapter.

5.1 Using the interrupt mechanism

Interrupts are mechanisms which enable instant microcontroller response to events such as : TMR0 counter overflow, state changes on RB0/INT pin, data is received over serial communication, etc. With bigger microcontrollers, number of interrupt sources is even greater. In normal mode, microcontroller executes the main program as long as there are no occurrences that would cause interrupt. When interrupt does take place microcontroller stops the execution of the main program and starts executing part of the program (interrupt routine) that will analyze and handle the interrupt. Analysis is necessary because PIC microcontrollers call the same interrupt routine in response to any of the mentioned events. Therefore, the first task is to determine which event caused the interrupt. After the analysis comes the interrupt handling, which is executing the appropriate part of program code tied to a certain event.



Button T is connected to the external interrupt input INT (pin RB0/INT) so that pressing the button is considered an interrupt occurrence. In order to see the change caused by interrupt LED diodes are connected to the pins RB6 and RB7. *LED_run* diode signalizes that the main program is being executed, while *LED_int* diode signalizes the interrupt caused by pressing the button T. Following instructions are used in PIC BASIC programs which contain interrupt routine :

- On Interrupt goto** *Address* Defines the interrupt vector (address of interrupt routine)
- Disable** Disables the interrupts
- Enable** Enables the interrupts
- Resume** Return to the main program after handling the event

Following example demonstrates usage of external interrupt INT located on pin RB0. At the same time, program gives an example how to handle multiple interrupt sources.

```

symbol LED_run = PORTB.7 ' LED_run is connected to pin RB7
symbol LED_int = PORTB.6 ' LED_int is connected to pin RB6
TRISB = %00111111 ' Pins RB7 and RB6 are output
OPTION_REG = %10000000 ' Turn off Pull-up resistors and
                        ' set the interrupt on the
                        ' descending edge of the signal

On Interrupt Goto ISR ' Interrupt vector
INTCON = %10010000 ' Enable external interrupts
PORTB = 0 ' Initial value on port B

Main: ' Beginning of the main program
LED_run=1 ' While there is no interrupt
          ' diode on RB7 is on while diode
          ' on RB6 is off
LED_int=0
goto Main ' Jump back to the beginning
Disable

ISR: ' Interrupt routine
if INTCON.0 = 1 then RBIF ' Change has occurred on RB4-RB7
if INTCON.1 = 1 then INTF ' Change has occurred on RB0/INT
if INTCON.2 = 1 then TOIF ' Overflow has occurred on TMRO
if EECON1.4 = 1 then EEIF ' Writing to EEPROM is finished

RBIF:
INTCON.0 = 0 ' Clear RBTF flag
'| Deo programa koji vrši
'| obradu prekida
goto Exit_ISR ' Exit from interrupt routine

INTF:
LED_int=1 ' When interrupt occurs diode RB7
LED_run=0 ' is off while diode on RB6 is on
Pause 500 ' pause for making change visible
INTCON.1 = 0 ' Clear INTF flag
goto Exit_ISR

TOIF:
INTCON.2 = 0 ' Clear TOIF flag
'|
'|
goto Exit_ISR

EEIF:
EECON1.4 = 0 ' Clear EEIF flag
'|
'|
goto Exit_ISR

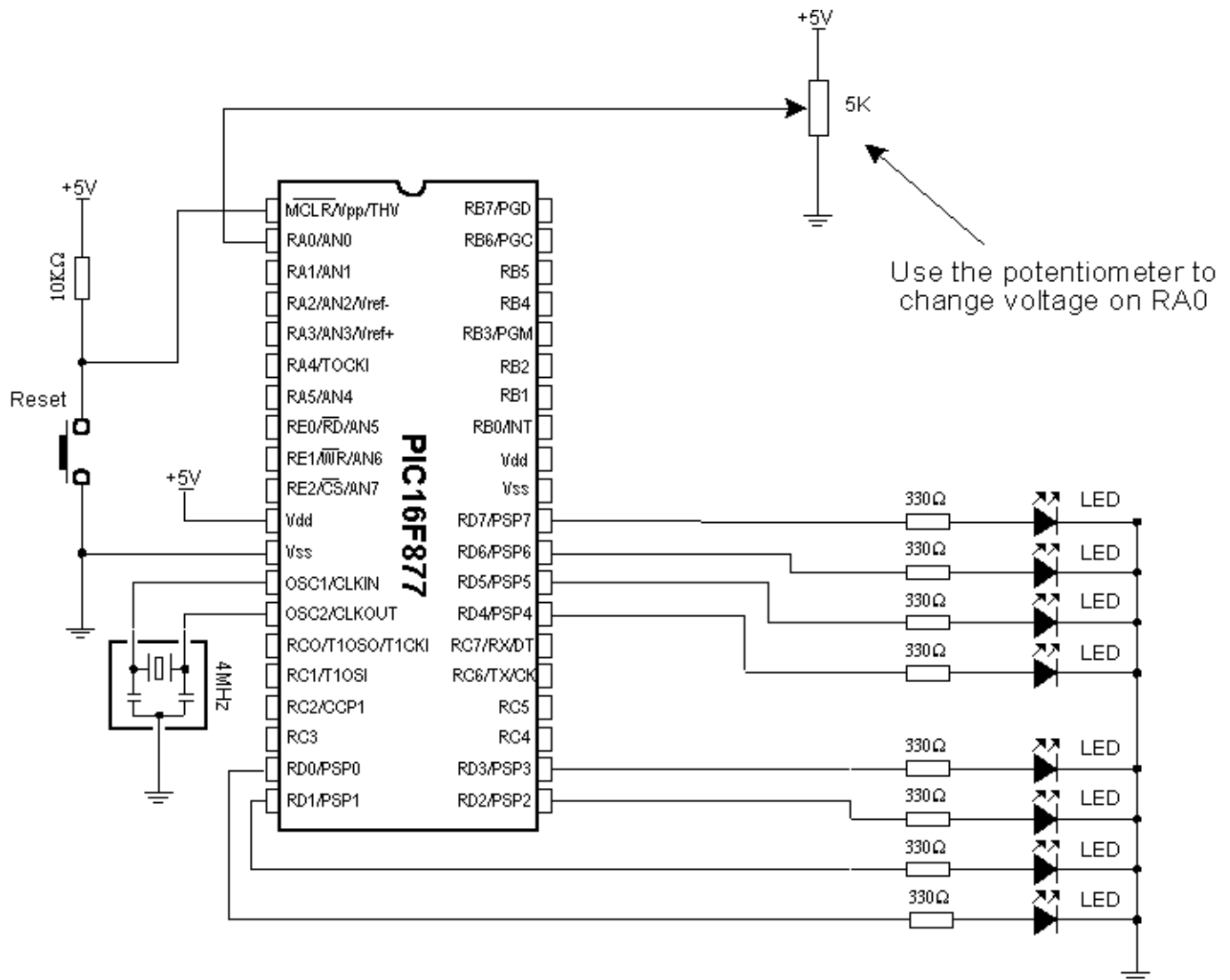
Exit_ISR:
Resume ' Exit from interrupt routine
Enable ' Enable interrupts
End ' End of the program

```

5.2 Using the internal AD converter

Certain microcontrollers have built in analog-digital converter (abbrev. ADC). Usually, these AD converters do not exceed 8 to 10 bits resolution allowing them voltage sensitivity of 19.5mV with 8-bit resolution and 4.8mV with 10-bit resolution (assuming that default 5V voltage is used).

The simplest AD conversion program would use 8-bit resolution and 5V of microcontroller power as referent voltage (value which the value "read" from the microcontroller pin is compared to). In the following example we measure voltage on RA0 pin which is connected to the potentiometer (picture below).



Potentiometer gives 0V in one terminal position and 5V in the other, so that digitalized voltage can take values ranging from 0 to 256 due to the fact that 8-bit conversion is used. The following program reads voltage on RA0 pin and displays it on port B diodes. If not one diode is on, result is zero and if all of diodes are on, result is 255.

Program: INT_ADC1.BAS

```

TRISA = %11111111 ' Port A is input
ADCON1 = %10000010 ' Port A is in analog mode, 0 and 5V are
                    ' referent voltage values and the result
                    ' is right formatted (higher 6 bits of
                    ' ADRESH are zeros).

ADCON0 = %11000001 ' ADC clock is generated by internal RC
                    ' circuit, voltage is measured on RA0
                    ' and allows the use of AD converter

Pause 500          ' Half a second pause

Main: ADCON0.2 = 1    ' Beginning of conversion

Cekaj: Pause 5
      If ADCON0.2 = 1 Then Cekaj      ' Wait for AD conversion
                                          ' to be finished

      POERTD = ADRESH      ' Set the lower 8 bits on PORTD
      Pause 500          ' Half a second pause
      Goto Main          ' Repeat all
      End                ' End of the program

```

At the very beginning, it is necessary to properly initialize 2 bit registers ADCON1 and ADCON0. Afterwards, only thing required is to set ADCON0.2 bit which initializes the conversion and checks ADCON0.2 to determine if conversion is over. After the conversion is over, result is stored into ADRESH and ADRESL where from it can be copied. Former example can also be carried out via ADCIN instruction. Following example uses 10-bit resolution and ADCIN instruction.

Program: INT_ADC2.BAS

```

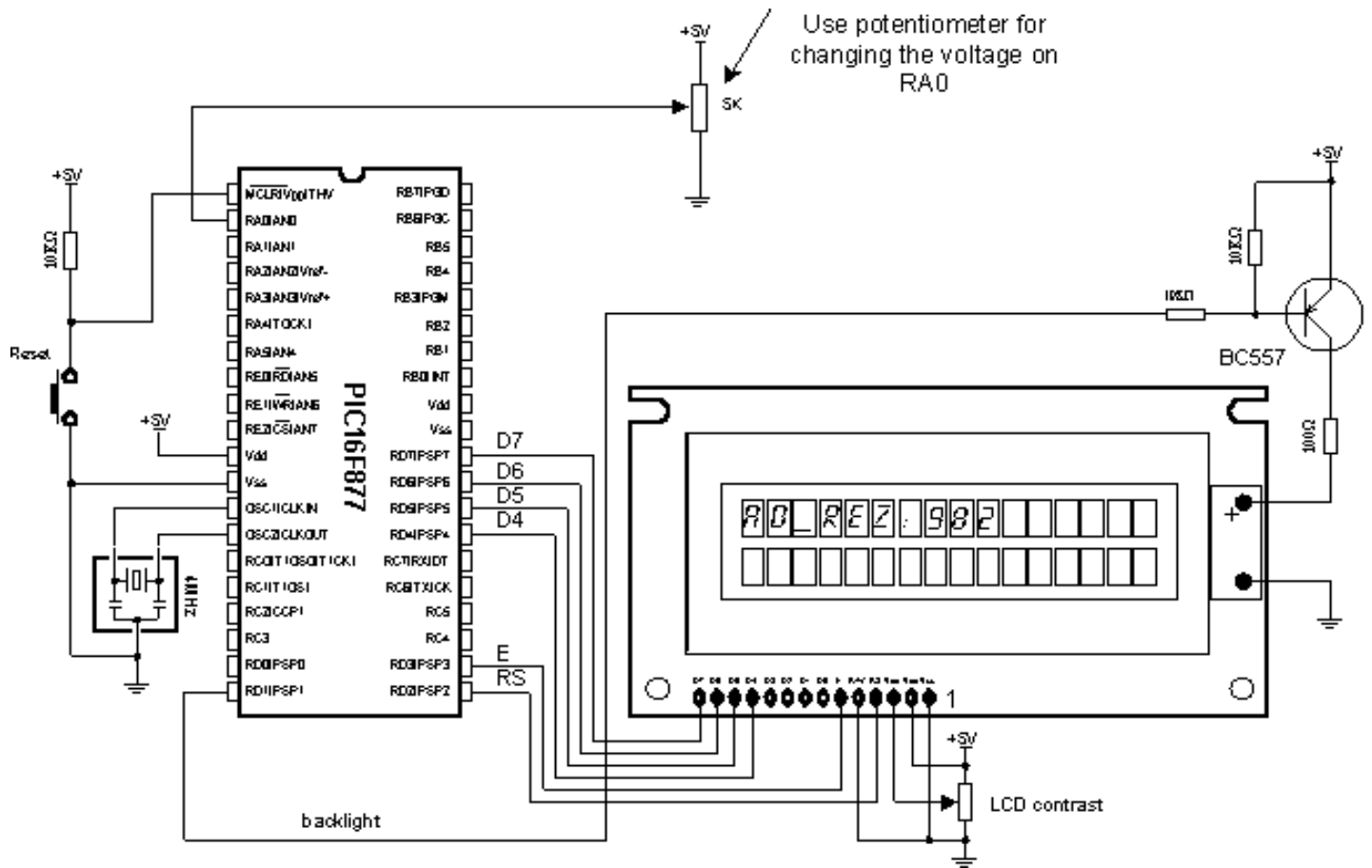
Define ADC_BITS      10    ' Number of bits
Define ADC_CLOCK    0     ' Clock (0=Oscillator /2)

ADC_Rez var word     ' Result of AD conversion is 16 bits.
TRISA = %11111111    ' Port A is input
TRISD = %00000000    ' Port D is output
ADCON1 = %10000010  ' Port A is in analog mode, 0 and 5V are
                    ' referent voltage values and the result
                    ' is right aligned.

Main: ADCIN 0, ADC_Rez ' Execute conversion and store resulting
                    ' 16 bits into variable ADC_Rez.
      PORTB=ADC_Rez.BYTE.0 ' display the resulting lower 8 bits
                    ' on port B
      Pause 500          ' Half a second pause
      Goto Main          ' Repeat all
      End                ' End of the program

```

As one port is insufficient, LCD can be used for displaying all of the 10 bits of result. Connection scheme is on the picture below and appropriate program follows.





```
DEFINE LCD_DREG PORTD
DEFINE LCD_DBIT 4
DEFINE LCD_BITS 4
DEFINE LCD_RSREG PORTD
DEFINE LCD_RSBIT 2
DEFINE LCD_EREG PORTD
DEFINE LCD_EBIT 3
DEFINE LCD_LINES 2
DEFINE LCD_COMMANDUS 2000
DEFINE LCD_DATAUS 40

Define ADC_BITS      10      ' Number of bits
Define ADC_CLOCK     1       ' Clock (0=Oscillator /8)

AD_Rez var word      ' Result of AD conversion is 16 bits.
TRISA = %11111111    ' Port A is input
ADCON1 = %10000010   ' Port A is in analog mode, 0 and 5V are
                      ' referent voltage values and the result
                      ' is formatted.

Main: ADCIN 0, AD_Rez ' Execute conversion and store resulting
                      ' 16 bits into variable ADC_Rez.

Lcdout $fe, 1        ' Clear the LCD
Lcdout $fe, 2        ' Set cursor to first line first
                      ' character
Lcdout "AD_rez: ", DEC AD_rez ' Print "AD_rez:" and
                      ' result of AD conversion

Pause 500           ' Half a second pause
Goto Main           ' Repeat all
End                 ' End of the program
```

5.3 Using the TMR0 timer


TMR0 timer is 8-bit and has working range of 255. Assuming that 4MHz oscillator is used, time period TMR0 can measure falls into 0-256 microseconds range (with 4MHz frequency TMR0 increments by one microsecond). If prescaler is used that period can be prolonged, because prescaler divides the clock in a certain ratio (prescaler settings are made in OPTION_REG register).

Following program illustrates use of TMR0 timer for generating 1 second time period. Prescaler is set to 32, so that internal clock is divided by 32 and TMR0 increments every 31 microseconds. If TMR0 is initialized on 96, overflow occurs in $(256-96)*31 \text{ us} = 5 \text{ ms}$. If variable "Brojac" is increased every time interrupt takes place, we can measure time according to the value of variable "Brojac". If "Brojac" is set to 200, time will total $200*5 \text{ ms} = 1 \text{ second}$.

Before the main program, TMR0 should have interrupt enabled (bit 2) and GIE bit (bit 7) in INTCON register should be set.

```

Program: TMR0.BAS



symbol LED = PORTB.1      ' LED diode is connected to RBO
brojac var byte        ' Temporary counter
TRISB = 0                   ' Pins of port B are output
PORTB = 0
INTCON = %00100000         ' Enable interrupt TMR0
OPTION_REG = %10000100    ' Set prescaler to 32
brojac = 0                  ' Initializing temporary counter
TMR0 = 96                   ' Initialization of TMR0

On Interrupt Goto ISR     ' Interrupt vector
INTCON = %10100000        ' Enable interrupts
Main:                       ' Beginning of the program
  if brojac = 200 then   ' Change the state of LED diode
    toggle LED           ' on every 200 * 5ms = 1s
    brojac = 0             ' Discard the counter
  endif
  goto Main              ' Jump to the beginning
Disable

ISR:
  brojac = brojac + 1      ' Increase counter by 1
  TMR0 = 96                ' Initialize the counter
  INTCON.2 = 0             ' Clear TOIF flag
  Resume                ' Return to the main program
Enable
End
```

5.4 Using the TMR1 timer

Unlike TMR0, TMR1 is 16-bit and has working range of 65536. Assuming that 4MHz oscillator is used, time period TMR1 can measure falls into 0-65536 microseconds range (with 4MHz frequency TMR01 increments by one microsecond). If prescaler is used that period can be prolonged, because prescaler divides the clock in a certain ratio (prescaler settings are made in T1CON register).

Before the main program, TMR1 should be enabled by setting the zero bit in T1CON register. Besides that, first bit of the register should be set to zero, thus defining the internal clock for TMR1.

Besides T1CON, other important registers for working with TMR1 include PIR1 and PIE1. The first contains overflow flag (zero bit) and the other is used to enable TMR1 interrupt (zero bit).

When TMR1 interrupt is enabled and its flag reset only thing left to do is to enable global interrupts (bit 7) and peripheral interrupts (bit 6) in the INTCON register.

The following program illustrates use of TMR1 register for generating 10 seconds time period. Prescaler is set to 00 so there is no dividing the internal clock and overflow occurs every 65.536 ms. If variable "Brojac" is increased every time interrupt takes place, we can measure one minute period according to the variable "Brojac". If "Brojac" is set to 152, time will total $152 \times 65.536 \text{ ms} = 9.960 \text{ second}$.

Program: TMR1.B&S

```

symbol LED = PORTB.1      ' LED diode is connected to RB1
Brojac var byte           ' Temporary counter
TRISB=%00000000           ' Pins of port B are output
T1CON=%00000001           ' Prescaler is 1:1 and enables
                            ' interrupt TMR1

PIR1.0=0                   ' clear the overflow flag of
                            ' TMR1 timer to prevent
                            ' generating interrupt at instant

PIE1= %00000001           ' Enable interrupt TMR1
Brojac = 0                 ' Initializing temporary counter
TMR1 = 0                   ' Initialization of TMR1
PORTB = 0                  ' All diodes on port D are off
On Interrupt Goto ISR     ' Interrupt vector
INTCON = %11000000        ' Enable interrupts
Main:                      ' Beginning of the program
    if Brojac = 152 then  ' Change the state of LED diode
        toggle LED        ' on every 152 * 65,5mS = 10s
        Brojac = 0         ' Discard the counter
    Endif
    goto Main              ' Jump to the beginning
    Disable
ISR:
    Brojac = Brojac + 1    ' Increase the counter by 1
    PIR1.0 = 0            ' Clear TOIF flag
    Resume                ' Return to the main program
    Enable
    End

```

5.5 Using the PWM subsystem

Microcontrollers of PIC16F87X series have one or two PWM outputs built-in (those in 40-pin casing have 2, while those in 28-pin casing have 1). PWM outputs are located on RC1 and RC2 pins in case of 40-pin microcontrollers and on RC2 pin in case of 28-pin microcontrollers. HPWM instruction greatly simplifies using the PWM. There are only 3 parameters to be set :

PWM Channel : defines which PWM channel is used; "1" defines channel on RC1 pin, while "2" defines channel on RC2 pin.

Ratio_S_P : defines the ratio of *on* and *off* signals on pin. "0" defines continual *off* state, whereas "255" defines continual *on* state. All values within these limits define appropriate ratio of *on* and *off* signals on pin. (i.e. "127" gives 50% of 0V on output and 50% of 5V on output).

Frequency : defines PWM signal frequency. Top frequency for any oscillator is 32767Hz.

The following example demonstrates use of PWM for getting various light intensities on LED diode connected to RC1 pin (PWM channel 0). Parameter defining ratio of *on* and *off* signals is continually increased in the *for-next* loop and takes value from 0 to 255, resulting in continual intensifying of light on LED diode. After value of 255 has been reached, process begins anew.

```
Program: HPWM.BAS

i var byte          ' Temporary variable
Odnos_S_P var byte ' Variable containing value of signal
                    ' and pause ratio. If 0 then the
                    ' signal is 0V all the time and if 255
                    ' signal is 5V all the time
Odnos_S_P=0        ' Initialization

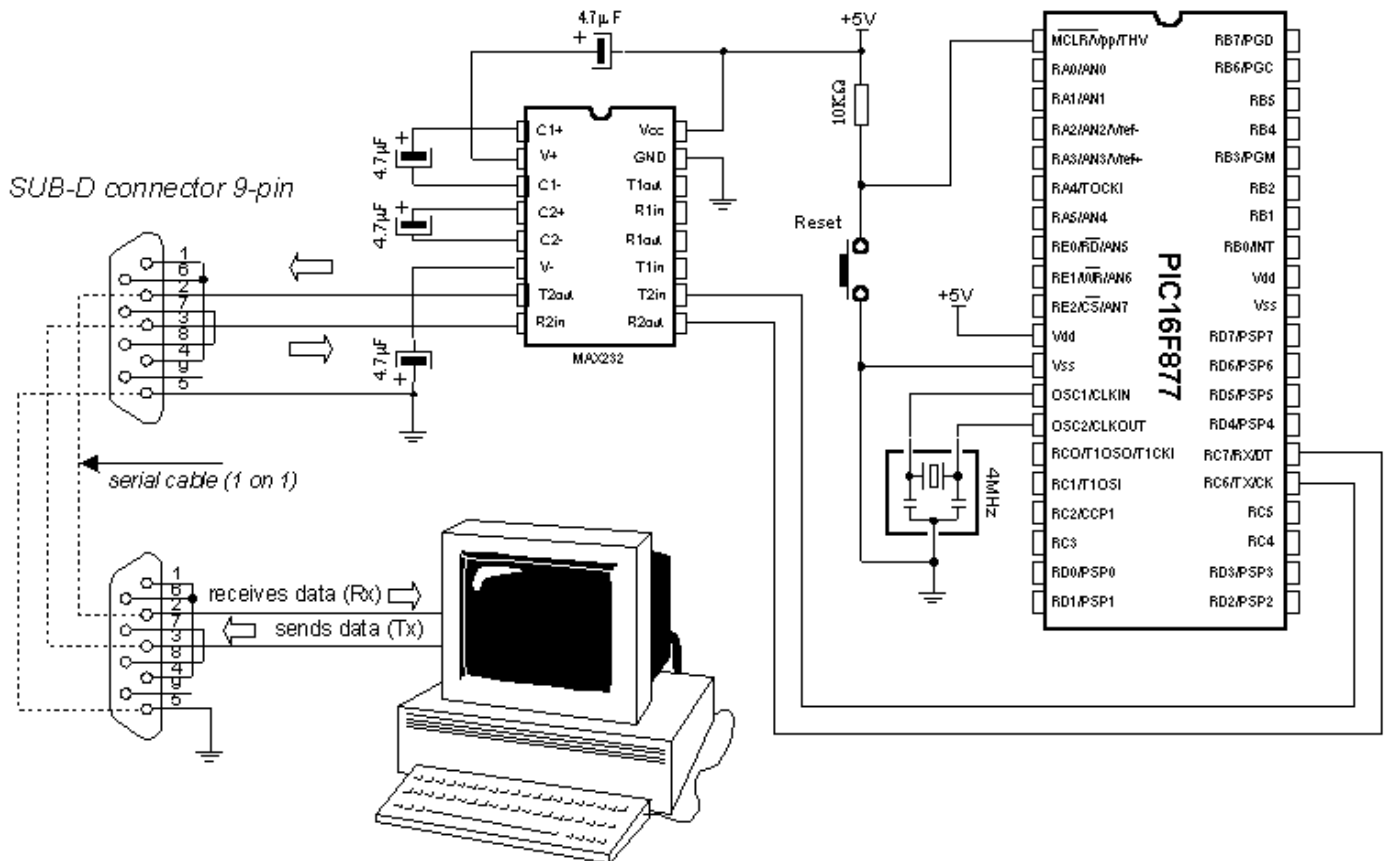
Main:
  For i=0 to 255
    HPWM 1,Odnos_S_P,2000 ' Operating with PWM channel
                          ' that is RC1 pin

    Odnos_S_P=Odnos_S_P+1
    pause 100
  Next i
  i=0
  Odnos_S_P=0
  Pause 4000
  goto Main
End
```

5.6 Using the hardware UART subsystem (RS-232 communication)

Easiest way to transfer data between microcontroller and some other device (i.e. PC or other microcontroller) is the RS-232 communication. It is serial asynchronous 2-line (Tx for transmitting and Rx for receiving) data transfer for within 10m range.

This example shows data transfer between the microcontroller and PC connected by RS-232 line interface (MAX232) which has role of adjusting signal levels on the microcontroller side (it converts RS-232 voltage levels +/- 10V to TTL levels 0-5V and vice versa). Microcontroller can achieve communication with serial RS-232 line via hardware UART (*Universal Asynchronous Receiver Transmitter*) which is the integral part of PIC16F87X microcontrollers.



UART contains special registers for receiving and transmitting data as well as BAUD RATE generator for determining data transfer rate.

The program below illustrates use of hardware serial communication subsystem (serial communication can also be software based on any of 2 microcontroller pins). Data received from PC is stored into variable B0 and sent back to PC as confirmation of successful transfer. Thus, it is easy to check if communications works properly. Transfer format is 8N1 and transfer rate is 2400 baud.

In order to achieve communication, PC must have the communication software. One such program is part of the *MicroCode* studio. It can be accessed by clicking *View* and then *Serial Communication Window*. New window will appear on screen and can be used for adjusting transfer settings. First it is necessary to set transfer rate by clicking *Baudrate* on the left of the window (set it to 2400, because microcontroller is set to that rate). Communication port is selected by clicking one of the 4 available depending on port connected to a serial cable.

After making adjustments, clicking *Connect* starts the communication. Type your message and click *Send Message* - message is sent to the microcontroller and back, where it is displayed on the screen.



```
B0 var byte          ' Var. for storing received data
TRISC = %10111111    ' PC6 output TX pin, rest is input
SPBRG = 25           ' Set Baud rate to 2400
RCSTA = %10010000    ' Enable serial port and reception
TXSTA = %00100000    ' Enable asynchronous data sending

Main:
  Gosub charin        ' Receiving data via serial line
  If B0 = 0 Then Main ' Data is not received
  Gosub charout       ' If received send it back
  Goto Main           ' Repeat the loop

charin:
  B0 = 0              ' B=0 if data is not received
  If PIR1.5 = 1 Then ' If PIR1.5 = 1 data
                    ' is in RCREG
  B0 = RCREG          ' Load the data from the receiving
                    ' register RCREG to B0

  Endif
  Return

charout:
  If PIR1.4 = 0 Then charout      ' Wait for sending
                                  ' register to be ready
  TXREG = B0                      ' Send the data to
                                  ' sending register

  Return

End                                ' End of the program
```

Chapter 6

SAMPLES WITH PIC16F84 MICROCONTROLLER

Introduction

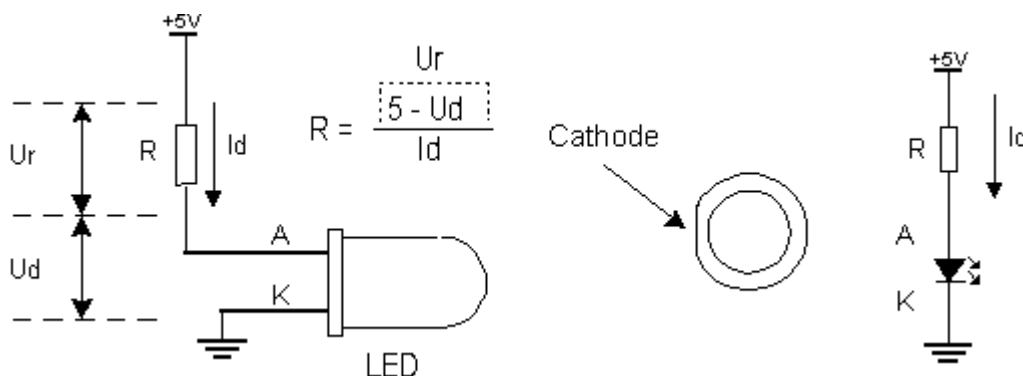
- [6.1 LED diode](#)
- [6.2 Button](#)
- [6.3 Generating sound](#)
- [6.4 Potentiometer](#)
- [6.5 Seven-segment displays](#)
- [6.6 Step motor](#)
- [6.7 Input shift register](#)
- [6.8 Output shift register](#)
- [6.9 Software serial communication](#)
- [6.10 Building light control](#)

Introduction

This chapter gives detailed examples of connecting PIC16F84 microcontroller to peripheral components and appropriate programs written in BASIC. All of the examples contain electrical connection scheme and program with comments and clarifications. You have the permission to directly copy these examples from the book or download them from the web site <http://www.mikroelektronika.co.yu/>.

6.1 LED diode

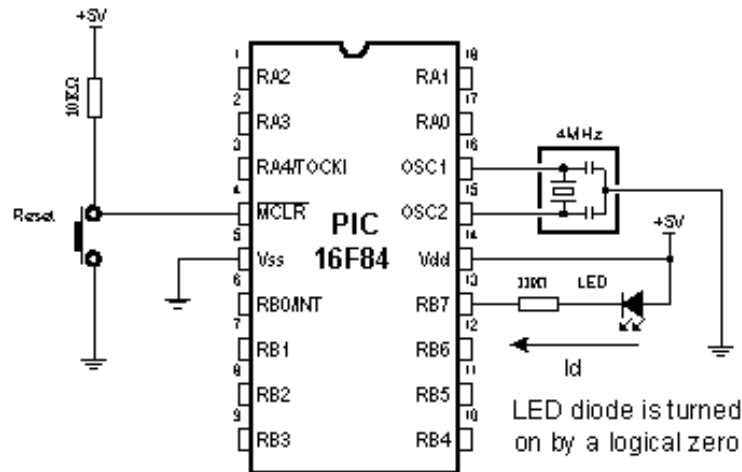
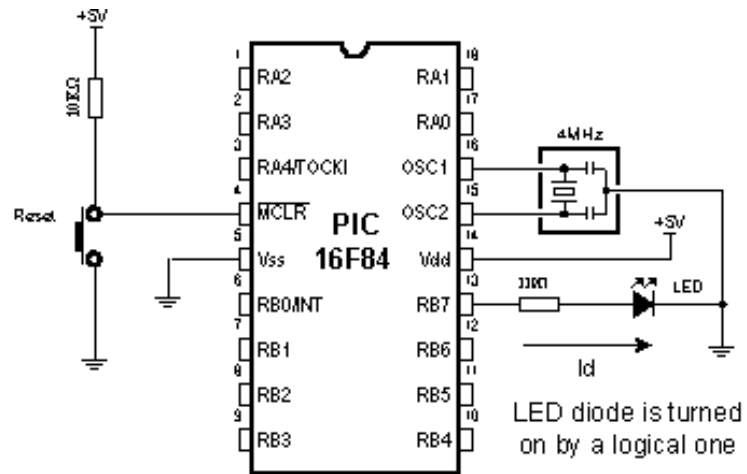
One of the most frequently used components in electronics is surely the LED diode (LED stands for *Light Emitting Diode*). Some of common LED diode features include : size, shape, color, working voltage (Diode voltage) U_d and electric current I_d . LED diode can have round, rectangular or triangular shape, although manufacturers of these components can produce any needed shape by order. Size i.e. diameter of round LED diodes ranges from 3 to 12 mm, with 3 or 5 mm sizes most commonly used. Color of emitting light can be red, yellow, green, orange, blue, etc. Working voltage i.e. necessary for LED diode to emit light is 1.7V for red, 2.1V for green and 2.3 for orange color. This voltage can be higher depending on the manufacturer. Normal current I_d through diode is 10 mA, while maximal current reaches 25 mA. High current consumption can present problem to devices with battery power supply, so in that case low current LED diode ($I_d \sim 1-2$ mA) should be used. For LED diode to emit light with maximum capacity, it is necessary to connect it properly or it might get damaged.



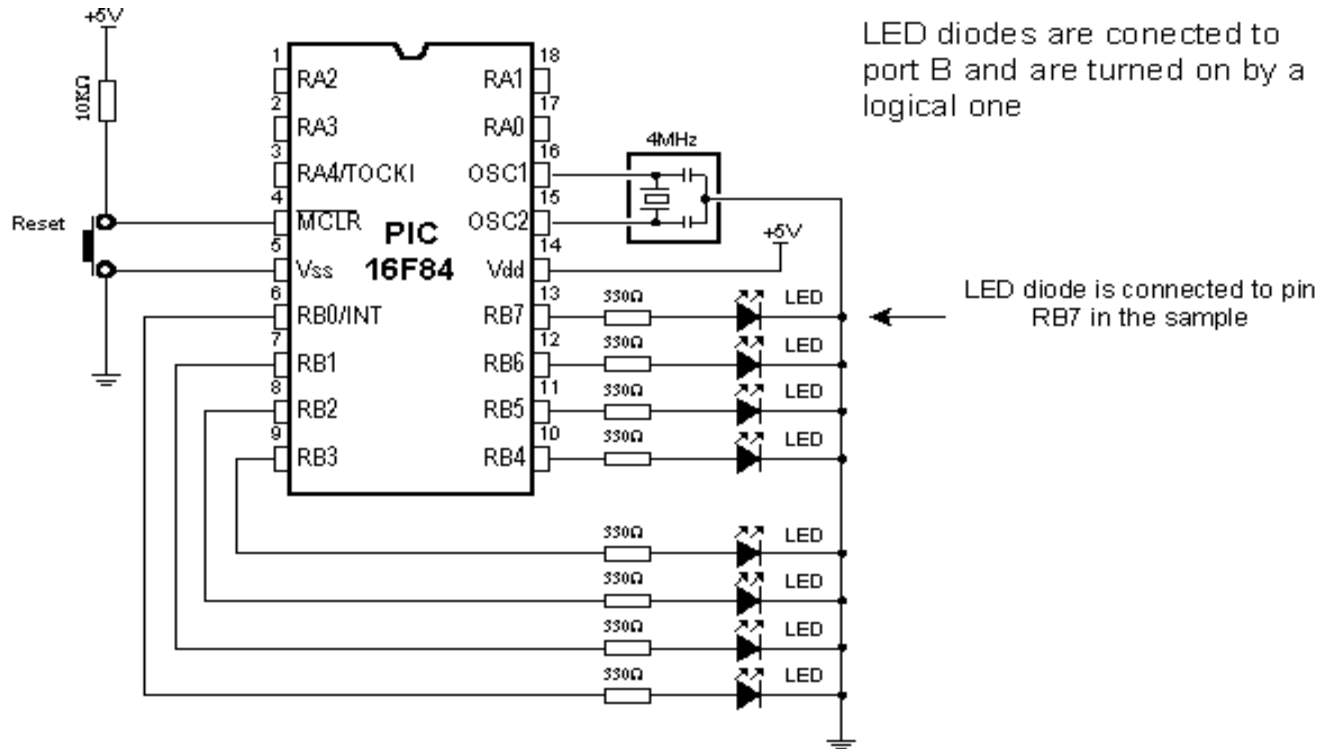
The positive pole is connected to anode, while ground is connected to cathode. For matter of differentiating the two, cathode is marked by mark on casing and shorter pin. Diode will emit light only if current flows from anode to cathode; in the other case there will be no current. Resistor is added serial to LED diode, limiting the maximal

current through diode and protecting it from damage. Resistor value can be calculated from the equation on the picture above, where U_r represents voltage on resistor. For +5V power supply and 10 mA current resistor used should have value of 330Ω .

LED diode can be connected to microcontroller in two ways. One way is to have microcontroller "turning on" LED diode with logical one and the other way is with logical zero. The first way is not so frequent (which doesn't mean it doesn't have applications) because it requires the microcontroller to be diode current source. The second way works with higher current LED diodes.



The following example uses instructions High, Low and Pause to turn on and off LED diode connected to seventh bit of port B every half second.



```

Program: LED.BAS

Loop:
    High PORTB.7      ' Turn on LED
    Pause 500         ' Half a second pause

    Low PORTB.7       ' Turn off LED
    Pause 500         ' Half a second pause

    Goto loop         ' Go back to Loop

End                  ' End of program
    
```

6.2 Button

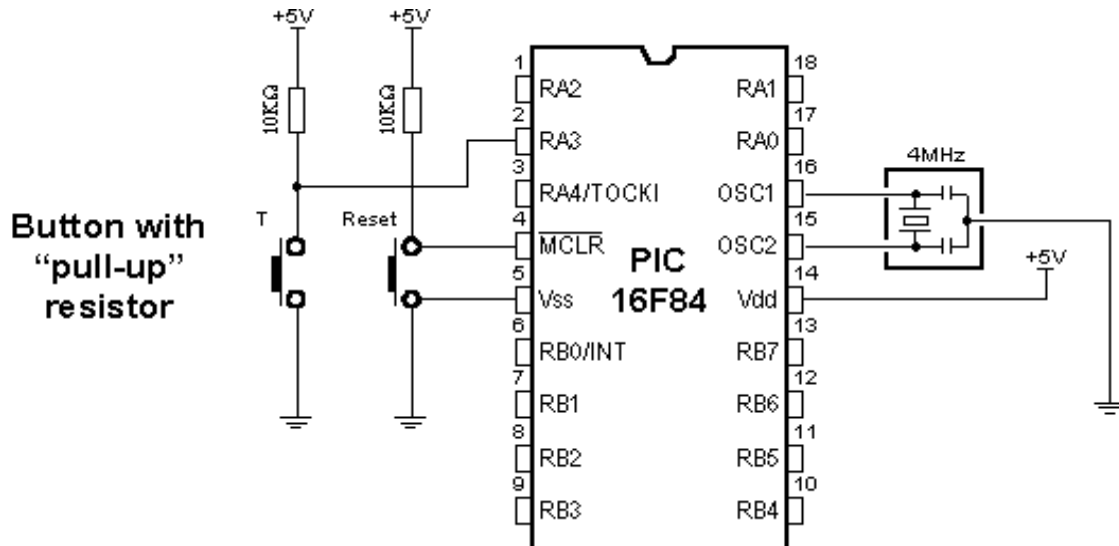
Button is a mechanical component which connects or disconnects two points A and B over its contacts. By function, button contacts can be normally open or normally closed.



Pressing the button with normally open contact connects the points A and B, while pressing the button with normally closed contact disconnects A and B.

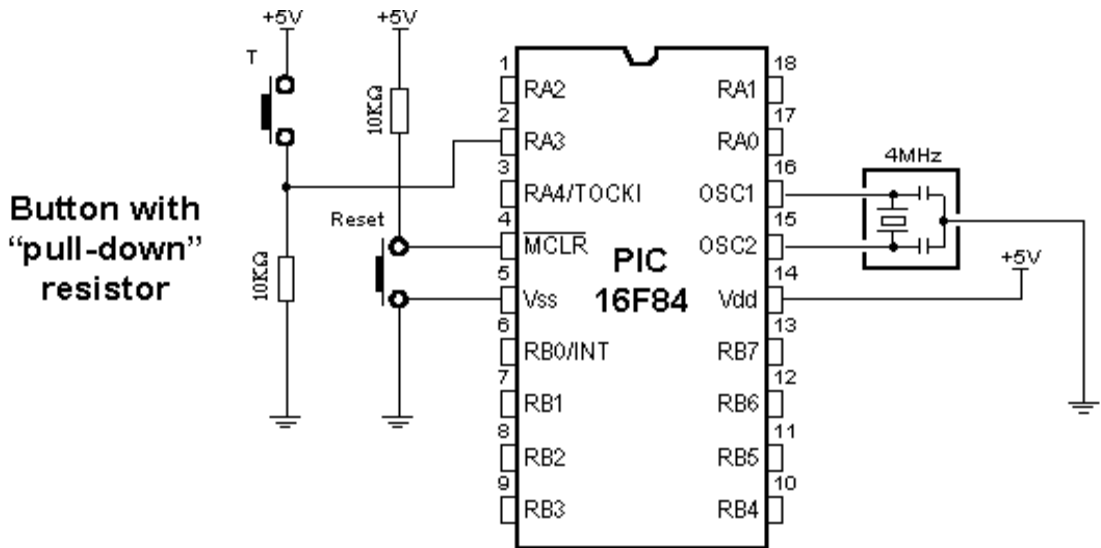
Buttons can be connected to the microcontroller in one of two ways:

In the first case, button is connected in a way that logical one (+5V) remains on microcontroller input pin while button is not pressed. Resistor between a button and power voltage has role of holding the input pin in defined state when the button is not pressed (in this case a logical one). This is necessary as a protection from glitch on input pin that might cause misinterpretation of program, i.e. as if button is pressed when it is not.



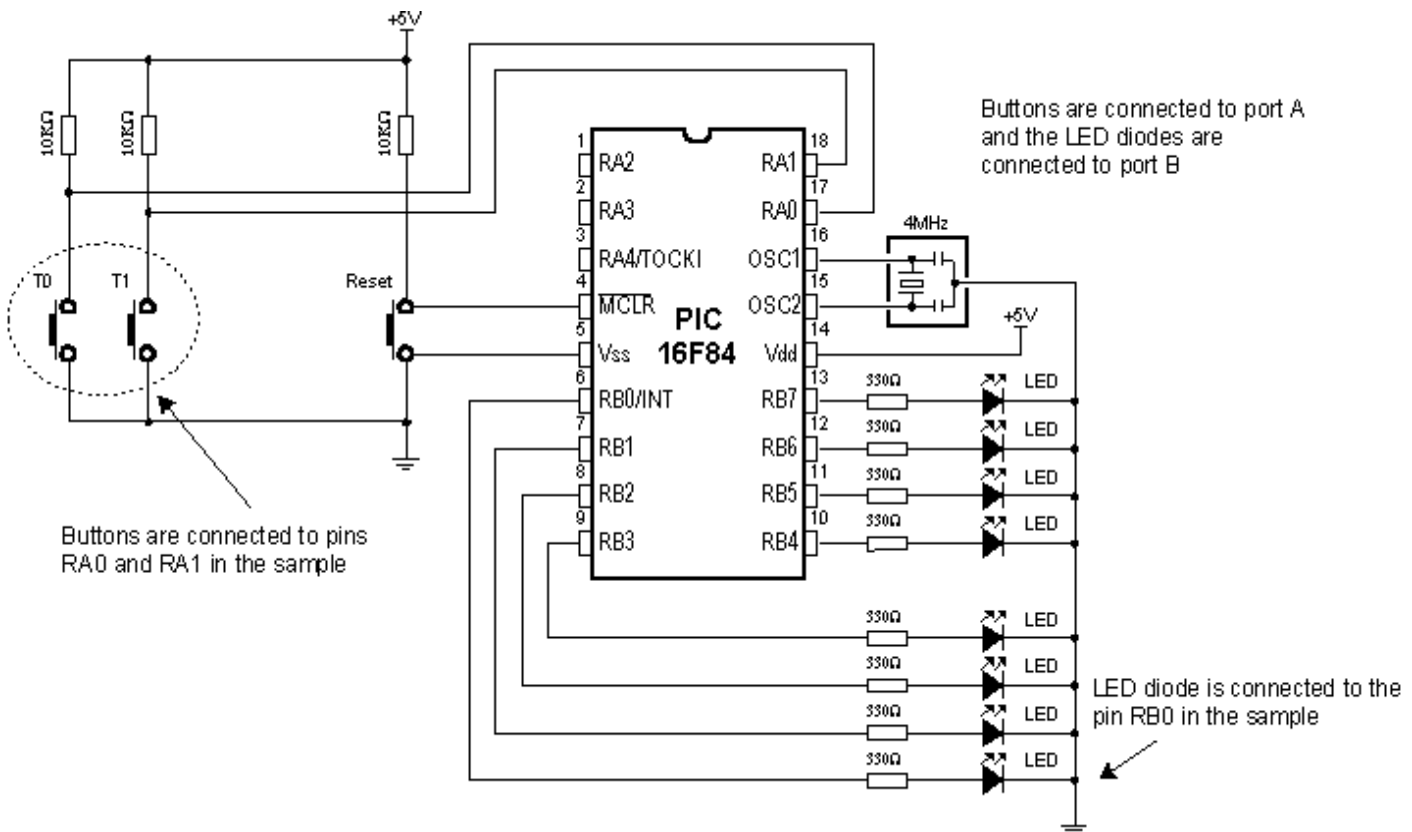
When the button is pressed, input pin is short circuited to the ground (0V) which indicates change on input pin. Voltage has dropped from 5V to 0V. This change is interpreted by program as if button was pressed and part of program code tied to a button (for example turn on LED diode) is then executed. This way of defining pin states is called defining with "pull-up" resistors, associating that the line is held up on the logical one level.

In the other case, button is connected in a way that logical zero remains on input pin. Now, resistor is between input pin and a logical zero, meaning that pressing the button brings logical one to input pin. Voltage goes up from 0V to +5V. Microcontroller program should recognize change on input pin and execute the specific part of program code. This way of defining pin states is called defining with "pull-down" resistors, associating that the line is held down on the logical zero level.



Button with "pull-down" resistor

Common way to connect the button is with *pull-up* resistors, meaning that pressing the button changes pin state from logical one to logical zero. Following picture displays four button connected to the microcontroller using the *pull-up* resistors.



Buttons are connected to port A and the LED diodes are connected to port B

Buttons are connected to pins RA0 and RA1 in the sample

LED diode is connected to the pin RB0 in the sample

Problem that occurs when working with buttons is contact debounce in the moment when button is pressed. Debounce is consequence of the contact and heavily depends on the very button.

One of the ways to solve the contact debounce problem is given in the following part of program code :

```
        if Button0=0 then Wait0    ` If Button0=0, jump to Wait0
        if Button1=0 then Wait1    ` If Button1=0, jump to Wait1
Wait0: if Button0=0 then Wait0    ` If Button0=0, wait until it is
        w=w+1                        ` released and increase w
Wait1: if Button0=0 then Wait0    ` If Button1=0, wait until it is
        w=w-1                        ` released and decrease w
```

Pressing the *Button0* causes the program to jump to address *Wait0* where it remains in the loop until the button is released (this achieves that single button push is just once handled in program). When *Button0* is released program continues executing instructions (in this case variable *W* is increased by one). Pressing *Button1* causes the same effect, except that variable *W* is decreased by one.

Problem might arise if an interrupt or some other source slows down the program execution, so that program finds itself on *Wait0* or *Wait1* lines after the button is released. This might cause program blocking until button is pressed again.

In the following program for reading the button states, BASIC instruction *Button* is used which eliminates the contact debounce.

The program reads buttons *T0* and *T1* which are connected to the pins *RA0* and *RA1*, respectively. Pressing the button *0* executes part of program code which turns on LED diode on pin *RB0*. Pressing the button *1* executes part of program code which turns off LED diode on the same pin. The mentioned instruction is among the most complex instructions of BASIC program language. Besides few arguments that should be defined, instruction has an argument for setting the delay time between recognition of two different button pressures (the third argument). Its setting depends on the purpose of the button as well as mechanical properties of the button. Still, it came clear over time that maximal value of last argument represents the best solution for most applications, because of great disproportion in human reaction and microcontroller speed.



```
B0 var byte ' Variable used by instruction BUTTON

symbol Button0 = PORTA.0 ' Button 0 is connected to pin RA0
symbol Button1 = PORTA.1 ' Button 1 is connected to pin RA1
symbol LED = PORTB.0      ' LED diode is connected to RB0

TRISA = $FF ' All pins of port A are input
TRISB = $00 ' All pins of port B are output
PORTB = $00 ' Turn off all LED diodes at start

Main:
B0 = 0      ' Initialize the variable B0
' If Button 0 is pressed jump onto LedOn
button Button0,0,255,0,B0,1,LedOn

B0 = 0
' If Button 1 is pressed jump onto LedOff
button Button1,0,255,0,B0,1,LedOff
goto Main  ' Jump back to the beginning of the program

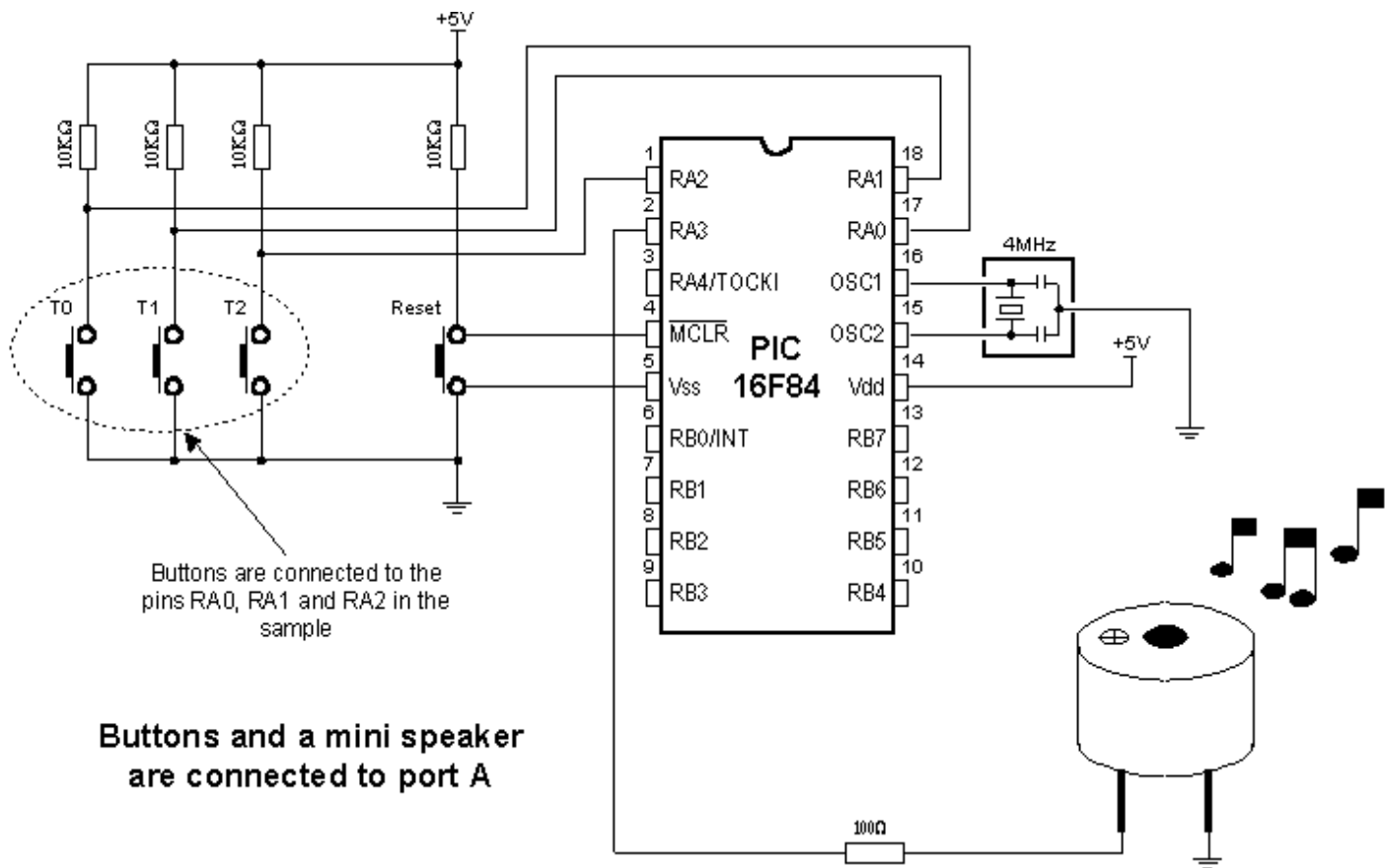
LedOn:
LED = 1     ' Turn on LED diode
goto Main

LedOff:
LED = 0     ' Turn off LED diode
goto Main

end      ' End of program
```

6.3 Generating sound

Sometimes it is necessary to provide sound signalization on device, besides the visual one (LED diodes). The following example shows one way to generate sound signal using the mini speaker and BASIC instruction *Sound*.



Buttons are connected to pins RA0, RA1 and RA2. Pressing any of these executes part of the code for generating impulse sequence on RA3 pin, which can be heard as a monotonous sound or a melody on mini speaker. Consecutive execution of instruction *Sound* with different parameters allows composing various melodies.

In the following program, pressing the button T0 generates one monotonous sound on a mini speaker, while pressing the buttons T1 and T2 executes sequences of *Sound* instructions which can be heard as two different melodies on a mini speaker.



```
B0 var byte ' Variable used by instruction BUTTON

symbol Button0 = PORTA.0 ' Button T0 is connected to RA0
symbol Button1 = PORTA.1 ' Button T1 is connected to RA1
symbol Button2 = PORTA.2 ' Button T2 is connected to RA2
symbol BeepPort = PORTA.3 ' Mini speaker connected to RA3
symbol BeepTris = TRISA.3

TRISA = $1f ' Initialization of port A
BeepTris = 0 ' Initializing the mini speaker
BeepPort = 0

Main:
' If button T0 is pressed jump onto Play0
B0 = 0
button Taster0,0,255,0,B0,1,Play0

' If button T1 is pressed jump onto Play1
B0 = 0
button Taster1,0,255,0,B0,1,Play1

' If button T2 is pressed jump onto Play2
B0 = 0
button Taster2,0,255,0,B0,1,Play2

goto Main ' Repeat the loop

Play0:
sound BeepPort, [110,255] ' Monotonous sound
goto Main ' Jump to beginning

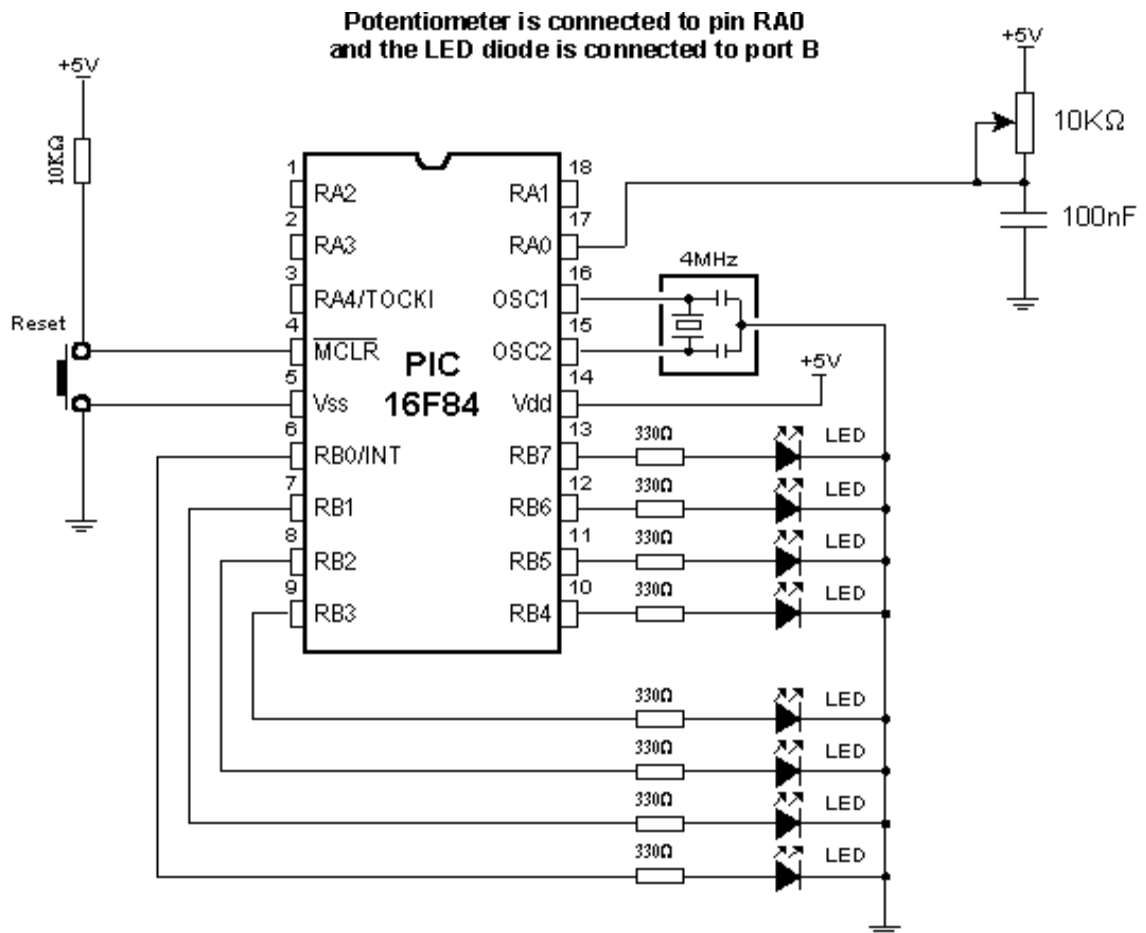
Play1:
sound BeepPort, [105,50,110,50,120,50] ' First melody
goto Main

Play2:
sound BeepPort, [120,50,110,50,105,50] ' Second melody
goto Main

End ' End of program
```

6.4 Potentiometer

In order to measure and display analog values, besides the microcontroller, it is necessary to have an AD converter. This can be an expensive solution if some less precise measuring is required, for example potentiometer voltage. For this reason PIC BASIC features the POT instruction for using the microcontroller without AD converter.



RC pair which consists of potentiometer (typical resistance in 5-50k range) and a 100nF capacitor is connected to RA0 pin. Reading the potentiometer is based upon measuring the time period between capacitor discharging and charging. Measuring scale ranges from 0 to 255 as if 8-bit AD converter was used.

The following program reads potentiometer value in 0-255 range and displays it on LED diodes connected to the port B.

Program: POT.B&3

```

B0 var byte      ' Variable used by instruction POT
                  ' Potentiometer is connected to RA0

symbol Potentiometer = PORTA.0

TRISA = %ff      ' Port A is designated input
TRISB = 0        ' Port B is designated output

Main:
                  ' Read the value of potentiometer

pot Potentiometer, 255, B0

PORTB = B0       ' Display value on LED diodes

pause 10         ' 10 ms pause

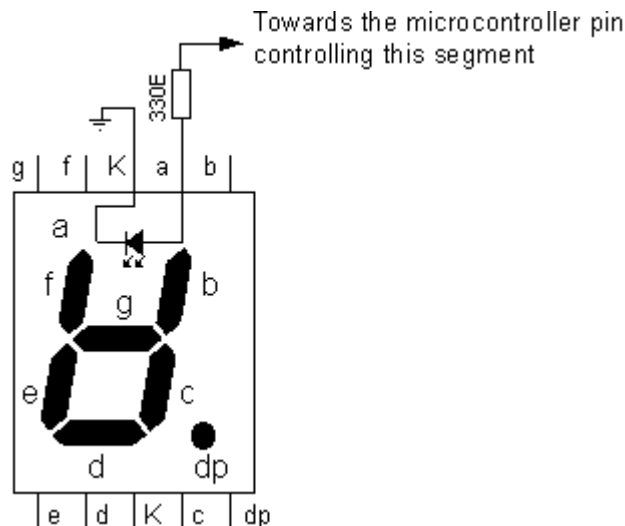
goto Main        ' Repeat the loop

end              ' End of program

```

6.5 Seven-segment displays

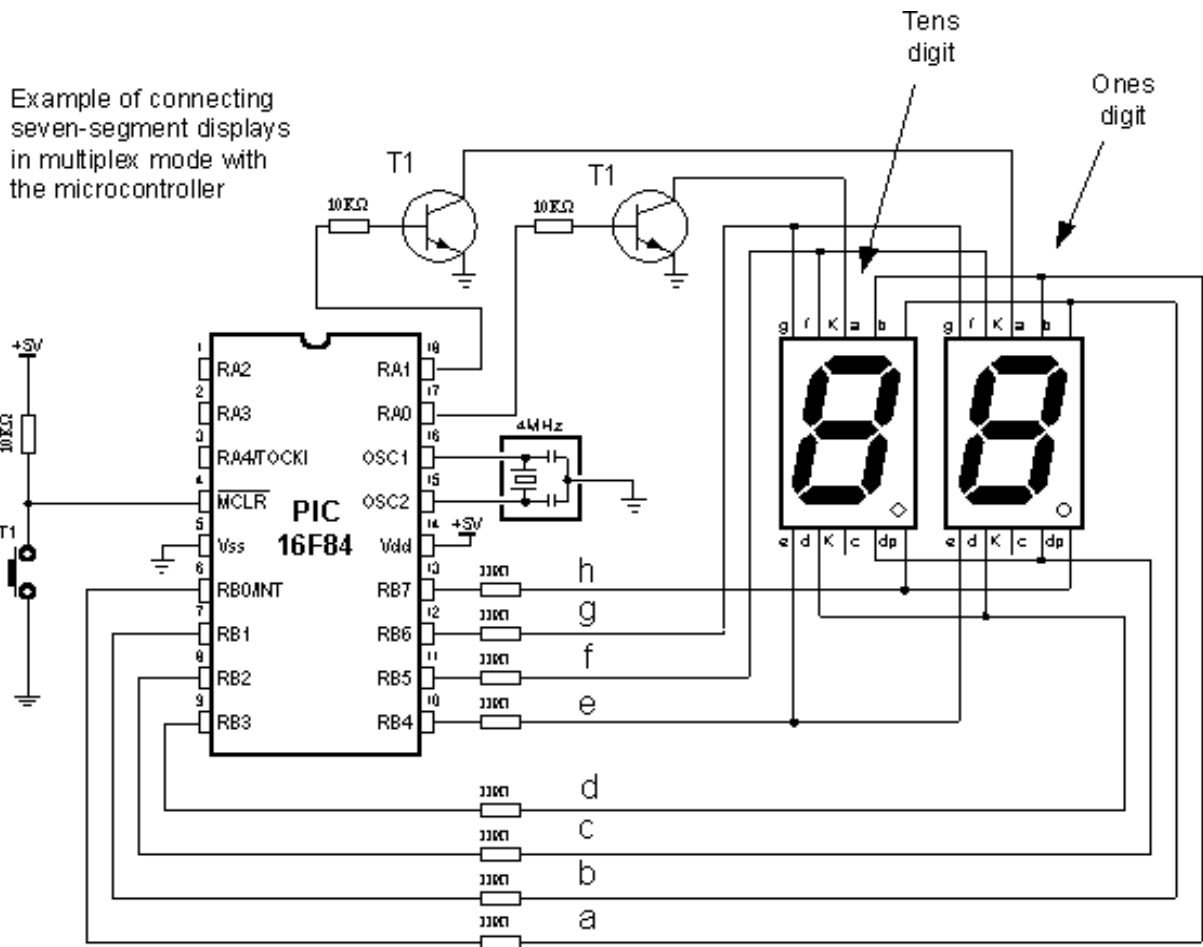
Most common form of communication between the microcontroller system and a man is, of course, the visual communication. The simplest form is the LED diode, while seven-segment digits represent more advanced form of visual communication. The name comes from the seven diodes (there is an eighth diode for a dot) arranged to form decimal digits from 0 to 9. Appearance of a seven-segment digit is given on a picture below.



As seven-segment digits have better temperature features as well as visibility than LCD displays, they are very common in industrial applications. Their use satisfies all criteria including the financial one. Simple application would be displaying value read from a certain sensor.



Digits can have a shared cathode (K) or anode (A). In the first case the segment is turned on by a logical one and in the second case, by logical zero.



One of the ways to connect seven-segment display to the microcontroller is given on a picture above. System is connected to use seven-segment digits with common cathode. This means that segments emit light when logical one is brought to them, and that output of all segments must be a transistor connected to common cathode, as shown on the picture. If transistor is in conducting mode any segment with logical one will emit light, and if not no segment will emit light, regardless of its pin state.

If we use the scheme from the picture above, one of the ways to realize the display in BASIC could be the following program code :

```

Program: Displays.BAS
Digit var byte      ' Value of number to be displayed
Maska var byte      ' Mask of number to be displayed
i var byte          ' temporary variable
LEDDis1 var PORTA.1 ' Transistor for ones digit
LEDDis2 var PORTA.0 ' Transistor for tens digit
TRISA=%00000000    ' all pins of port A are output
TRISB=%00000000    ' all pins of port B are output
LEDDis2=0          ' Digit on PA1 (ones) is off
LEDDis1=1          ' Digit on PA0 (tens) is on

Main:
  for i=0 to 9
  Cifra=i
  Lookup Digit, [$3F,$06,$5B,$4F,$66,$6D,$7D,$07,$7F,$6F],Maska
  PORTB=Maska      ' Send mask of a number to port B
  pause 500      ' Pause allowing to see the change
  next i          ' Increase i by one
  goto Main      ' Repeat the loop
  end           ' End of program

```

Variables LEDDisp1 and LEDDisp2 are actually pins 1 and 0 of port A, which bases of transistors T1 and T2 are connected to. Setting logical one on those pins turns on the transistor, allowing every segment from "a" to "h", with logical one on it, to emit light. If there is logical zero on transistor base, none of the segments will emit light, regardless of the pin state. Tens digit is disabled at the very beginning of program, ahead of label *Main* (LEDDisp2=0).

Purpose of the program is to display figures from 0 to 9 on the ones digit, with 0.5 seconds pause in between. In order to display any number, it's mask must be sent to port B. For example, if we need to display "1", segments "b" and "c" must be set to 1 and the rest must be zero. If (according to the scheme above) segments b and c are connected to the first and the second pin of port B, values 0000 and 0110 should be set to port B. These values which are set to port are commonly called "masks". Mask for number "1" is value 0000 0110 or \$06 (hexadecimal). The following table contains corresponding mask values for numbers 0-9 :

Digit	Seg. h	Seg. g	Seg. f	Seg. e	Seg. d	Seg. c	Seg. b	Seg. a	HEX
0	0	0	1	1	1	1	1	1	\$3F
1	0	0	0	0	0	1	1	0	\$06
2	0	1	0	1	1	0	1	1	\$5B
3	0	1	0	0	1	1	1	1	\$4F
4	0	1	1	0	0	1	1	0	\$66
5	0	1	1	0	1	1	0	1	\$6D
6	0	1	1	1	1	1	0	1	\$7D
7	0	0	0	0	0	1	1	1	\$07
8	0	1	1	1	1	1	1	1	\$7F
9	0	1	1	0	1	1	1	1	\$6F

Program uses the instruction *Lookup* to apply an appropriate mask to numerical value. Instruction *Lookup* works very simply - it puts a character from a sequence, its position defined by numerical value *Digit*, to variable *Mask*. For example, *Mask* will take value \$5B if *Digit* has value 2. In that manner, we can easily get mask for any decimal digit.

Continual display of *Mask* (PORTB=Mask) for appropriate value of variable *Digit*, with 0.5sec pause, will produce an effect of digits rotating from 0 to 9.

Problem with multiplexing occurs when displaying more than one digit is needed on two or more displays. It is necessary to put one mask on one digit quickly enough and activate its transistor, then put the second mask and activate the second transistor (of course, if one of the transistors is in conducting mode, the other should not work because both digits will display the same value).

New program differs from the one above in converting 2-digits value to 2 masks, which are displayed in a way that human eye gets impression of simultaneous existence of both figures (this is the reason for calling it "multiplexing" - only one display actually emits in any given moment).

Let's say we need to display number 35. First, the number should be separated into tens and ones (in this case, digits 3 and 5) and their masks sent to port B. This separation can be done with instruction *Dig*. For example, *Digit1= W dig 0* will extract ones digit from variable *W* and store it into variable *Digit1*. If 0 is substituted with 1, tens digit will be extracted. Following the same logic, 2 extracts number of hundreds, 3 number of thousands, etc.

Program: Display2.EAS

```

Digit var byte      ' Value of number to be displayed
Mask var byte      ' Mask of number to be displayed
W var byte         ' temporary variable
LEDDis1 var PORTA.1 ' Transistor for ones digit
LEDDis2 var PORTA.0 ' Transistor for tens digit

TRISA=%00000000    ' all pins of port A are output
TRISB=%00000000    ' all pins of port B are output
LEDDis1=0          ' ones digit is off in the start
LEDDis2=0          ' tens digit is off in the start

Main:
W=35
Digit=W dig 1      ' Put tens to variable Digit
Gosub bin2seg      ' Call the conversion of binary number
                  ' to a code of appropriate 7seg digit

PORTB=Mask         ' Set the mask of a digit to port B
LEDDis2=1         ' Print the tens digit
pause 1           ' Hold it printed for 1 ms
LEDDis2=0         ' Turn off the tens digit

Digit=W dig 0      ' Put ones to variable Digit
Gosub bin2seg      ' Call the conversion of binary number
                  ' to a code of appropriate 7seg digit

PORTB=Digit
LEDDis1=1         ' Print the ones digit
pause 1           ' Hold it printed for 1 ms
LEDDis1=0         ' Turn off the ones digit

Goto Main         ' Again, for achieving the effect that
                  ' both digits are on simultaneously

bin2seg:
Lookup Digit, [%3F,%06,%5B,%4F,%66,%6D,%7D,%07,%7F,%6f],Mask
Return
End

```

This part of program code prints value 35 on two seven-segment displays. The rest of the program is very similar to the last example, except for having one transition caused by displaying one digit after another. This transition can be spotted when LEDDisp1 is being turned off and LEDDisp2 turned on with a new mask. Lookup table is still the same and may be called as a subroutine when needed.

The multiplexing problem is solved for now, but the program doesn't have a sole purpose to print values on displays. It is commonly just a subroutine for displaying certain information. However, this kind of solution for printing data on display will make essence of the program much more complicated. This newly encountered problem may be solved by moving part of the program for refreshing the digits (part of the program code for handling the masks and controlling the transistors) to interrupt routine. The following program shows how to use interrupt for refreshing the display. Main program increases the value of variable *W* from 0 to 99 and that value is printed on displays. After reaching the value of 99, counter begins anew.

```

Digit var byte      ' Value of number to be displayed
Mask var byte      ' Mask of number to be displayed
W var byte         ' temporary variable
i var byte         ' temporary variable
LEDDis1 var PORTA.1 ' Transistor for ones digit
LEDDis2 var PORTA.0 ' Transistor for tens digit

TRISA=%00000000    ' all pins of port A are output
TRISB=%00000000    ' all pins of port B are output
LEDDis1=0          ' ones digit is off at the start
LEDDis2=0          ' tens digit is off at the start

INTCON = %00100000 ' Enable interrupt TMRO
OPTION_REG = %10000000 ' Initialization of prescaler

On Interrupt Goto ISR ' Interrupt vector
INTCON = %10100000 ' Enable interrupts
W=0 ' Initialization of variable W

Main: ' Beginning of the program
for i=1 to 99 ' Print values from 0 to 99
W=W+1 ' Increase variable W
Gosub Prepare ' Prepare value from W to be displayed
pause 500 ' Pause to see the digits
next i
goto Main ' Print values from 0 to 99 again

Prepare:
Digit=W dig 1 ' Value of ones is put to var. Digit
Gosub bin2seg ' Converting digit to mask
Mask1=Digit ' Mask 1 contains the mask of ones

Digit=W dig 0
Gosub bin2seg ' Converting digit to mask
Mask2=Digit ' Mask 1 contains the mask of tens
Return ' Return from subroutine

bin2seg:
Lookup Digit,[%3F,%06,%5B,%4F,%66,%6D,%7D,%07,%7F,%6f],Cifra
Return

Disable ' Disable interrupts while ISR is
' executing

ISR:
PORTB=Mask1 ' Put a mask of tens digit to port B
LEDDis2=1 ' Print the ones digit
pause 1 ' Hold it printed for 1 ms
LEDDis2=0 ' turn off the tens digit

PORTB=Mask2 ' Put a mask of ones digit to port B
LEDDis1=1 ' Print the ones digit
pause 1 ' Hold it printed for 1 ms
LEDDis1=0 ' turn off the ones digit

INTCON.2 = 0 ' Clear TOIF flag
Resume ' Return to program
Enable ' Interrupts are enabled again

End ' End of program

```

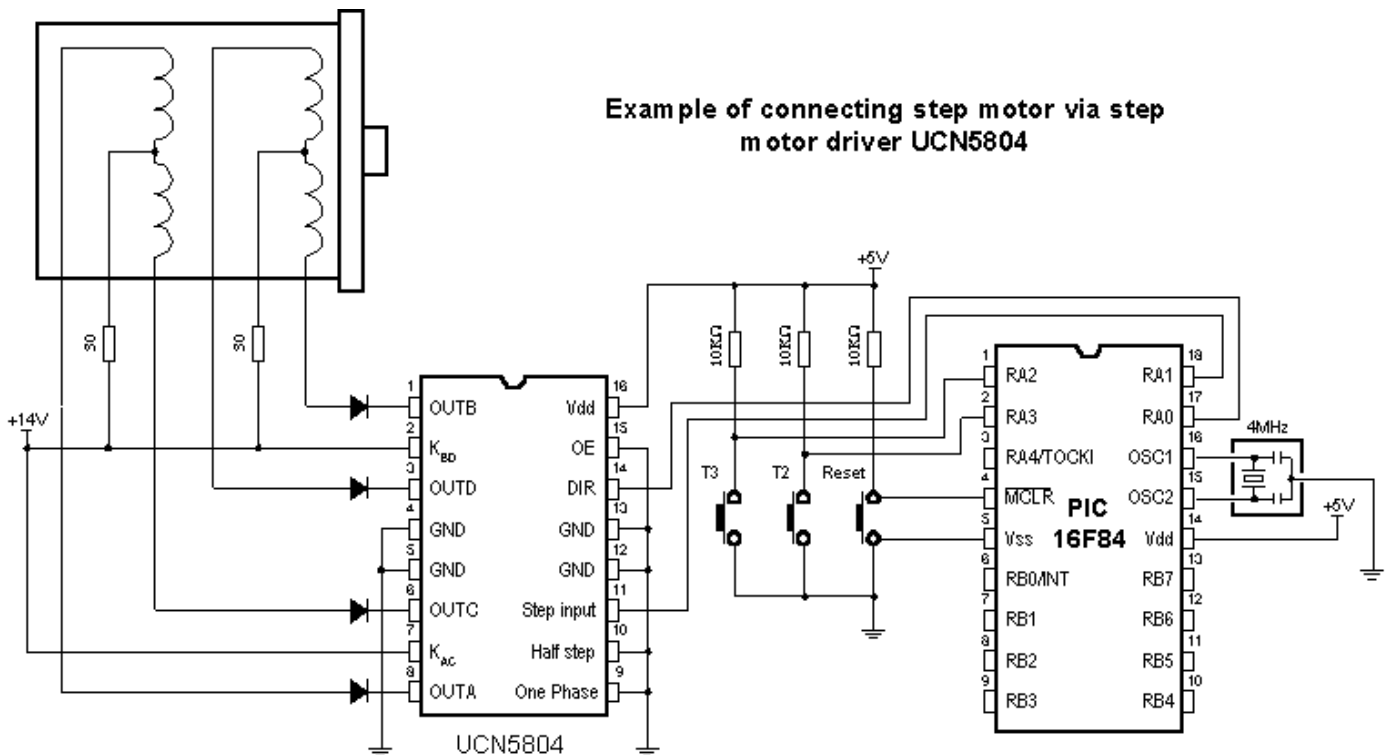
Interrupt initialized in this way will generate interrupt every time TMR0 timer changes state from 255 to 0. Every time interrupt takes place, interrupt routine will be executed so that human eye gets impression that both displays print values simultaneously. As can be seen from the program code, everything tied to displaying digits is moved to interrupt routine. However, part of the code for forming the masks to be displayed is in the special subroutine (Gosub Prepare) in order to make interrupt routine code as short as possible. Another reason for this kind of organization is also the need to create masks only when variable W is changed and not every time interrupt takes place.

In the course of main program, programmer doesn't have to take care of refreshing the display nor anything about displays whatsoever. It is only necessary to call subroutine "Preparation" every time value that will be displayed changes.

As 2-digit values don't satisfy most needs, the following step is adding two additional digits. Program for realization of 4 seven-segment displays is just an expansion of the program above. The main difference is in the part for separating values to ones, tens, hundreds and thousands.

6.6 Step motor

Of all motors, step motor is the easiest to control. It's handling simplicity is really hard to deny - all there is to do is to bring the sequence of rectangle impulses to one input of step controller and direction information to another input. Direction information is very simple and comes down to "left" for logical one on that pin and "right" for logical zero. Motor control is also very simple - every impulse makes the motor operating for one step and if there is no impulse the motor won't start. Pause between impulses can be shorter or longer and it defines revolution rate. This rate cannot be infinite because the motor won't be able to "catch up" with all the impulses (documentation on specific motor should contain such information). The picture below represents the scheme for connecting the step motor to microcontroller and appropriate program code follows.





```
Include "modedefs.bas"      ' Modes of data transfer used
                             ' by instruction SHIFTOUT

TxData var word            ' Variable where from data is
                             ' sent to shift register

B0 var byte                ' Variables used by
                             ' instruction BUTTON

B1 var byte

symbol Dir_in = PORTA.0    ' Clk line is connected to RA0
symbol Step_in = PORTA.1   ' Din line is connected to RA1
symbol Button2 = PORTA.2   ' Button T2 is connected to RA2
symbol Button3 = PORTA.3   ' Button T3 is connected to RA3
TRISA = %11111100         ' Configuring I/O ports
TRISB = 0
PORTB = 0

low Dir_in
low Step_in

Main:
    B0 = 0
    button Button2,0,255,0,B0,1,Left
    B0 = 0
    button Button3,0,255,0,B0,1,Right
    goto Main              ' Jump to the beginning

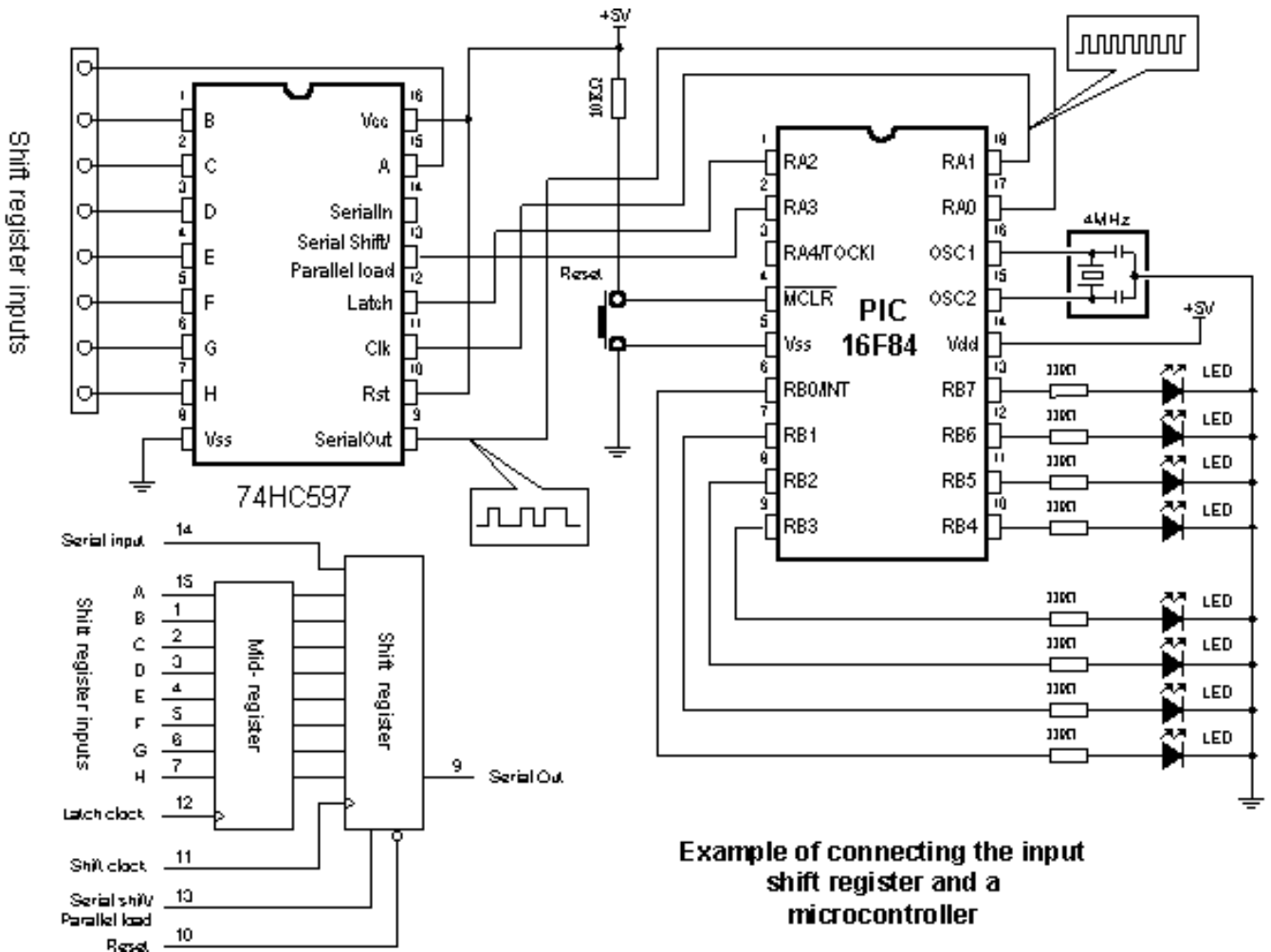
Left:
    low Dir_in            ' Set direction left
    gosub Make_circle     ' Make a full circle to left
    goto Main             ' Jump to the beginning

Right:
    high Dir_in           ' Set direction left
    gosub Make_circle     ' Make a full circle to right
    goto Main             ' Jump to the beginning

Make_circle:
    for B1 = 0 to 199     ' Motor used in the sample
    toggle Step_in        ' makes full circle in 200 steps
    pause 50              ' 50 mS pause between two steps
    next B1
    return
End                        ' End of program
```

6.7 Input shift register

Insufficient number of microcontroller input lines might represent a problem. Instead of switching to another, more expensive microcontroller model, input shift register 74HC597 could be an answer. For connecting input shift register it is necessary to take 4 of microcontroller I/O lines and one Latch line for every next register. This provides you 8 input lines per shift register. Input shift register transfers states of input pins to one mid-register using the Latch signal. After that, Load signal transfers data from mid-register to shift register, where from it is sent to the microcontroller via SerialOut and Clk lines.



Data transfer between shift registers and a microcontroller is serial. The following program reads states of input shift registers, transfers them to variable RxData and then displays the contents of RxData on diodes connected to port B. For data transfer between shift register and a microcontroller, BASIC instruction SHIFIN is used.



```
' Modes of data transfer used by instruction SHIFTLIN
Include "modedefs.bas"

' Variable for taking data from shift register
RxData var byte

' SerialIN line is connected to RA0
symbol HC_Data = PORTA.0
' Clk line is connected to RA1
symbol HC_Clk = PORTA.1
' Latch line is connected to RA2
symbol HC_Latch = PORTA.2
' Load line is connected to RA3
symbol HC_Load = PORTA.3

TRISA = %00010001 ' Initialization of port A
PORTA = 0
TRISB = 0          ' Initialization of port b
PORTB = 0

Main:
gosub Get_Data      ' Read the inputs of shift register
PORTB=RxDATA        ' Display input states on LED diodes
goto Main          ' Repeat the loop

End                ' End of program

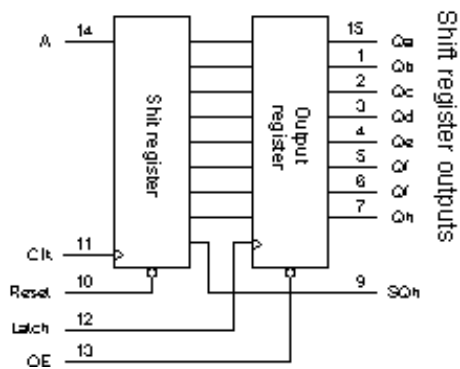
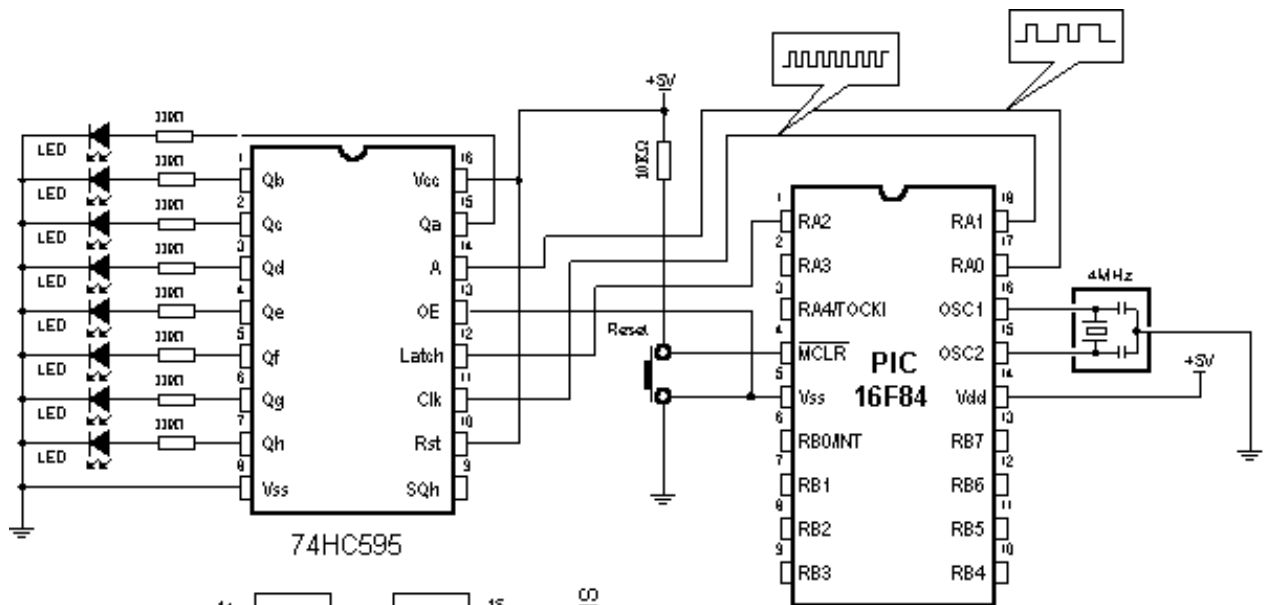
Get_Data:

    HC_Latch=1      ' Transfer states of input lines to
                    ' mid-register.
pauseus 5
    HC_Latch=0
    HC_Load=0       ' Transfer data from mid-register to
                    ' shift register
pauseus 5
    HC_Load=1

    ' Transfer data from shift register to variable RxData
SHIFTLIN HC_Data,HC_Clk,MSBPRES,[RxData]
return
```

6.8 Output shift register

Insufficient number of output lines might represent a problem with microcontrollers commanding small number of I/O lines, such as PIC16F84. If this is the case, output shift register 74HC595 could be used as an expansion. For connecting it, it is necessary to take 3 of microcontroller I/O lines, thus getting 8 additional output lines. Data transfer between shift register and a microcontroller is serial and commences via the Clk and A lines.



Example of connecting the output shift register and a microcontroller

When data is transferred to shift register, Latch signal sends it to output pins. The following program alternately turns on upper and lower LED diodes. For data transfer between shift register and a microcontroller, BASIC instruction SHIFTOUT is used.

```

' Modes of data transfer used by instruction SHIFTOUT
Include "modedefs.bas"

' Variable for supplying data to shift register
TxData var byte

' A line is connected to pin RA0
symbol HC_Data = PORTA.0
' Clk line is connected to pin RA1
symbol HC_Clk = PORTA.1
' Latch is connected to pin RA2
symbol HC_Latch = PORTA.2

TRISA = 0      ' Pins of port A are output
PORTA = 0

Main:
TxData=$F0      ' Turn on the upper 4 LED diodes
gosub Send_data ' Send the data to shift register

pause 500      ' 500 msec pause

TxData=$0F      ' Turn on the lower 4 LED diodes
gosub Send_data

pause 500

goto Main      ' repeat the loop

End           ' End of program

Send_data:

' Send the contents of TxData to shift register
shiftout HC_Data,HC_Clk,MSBFIRST,[TxData]

' Transfer data from shift register to output register
' i.e. turn on LED diodes

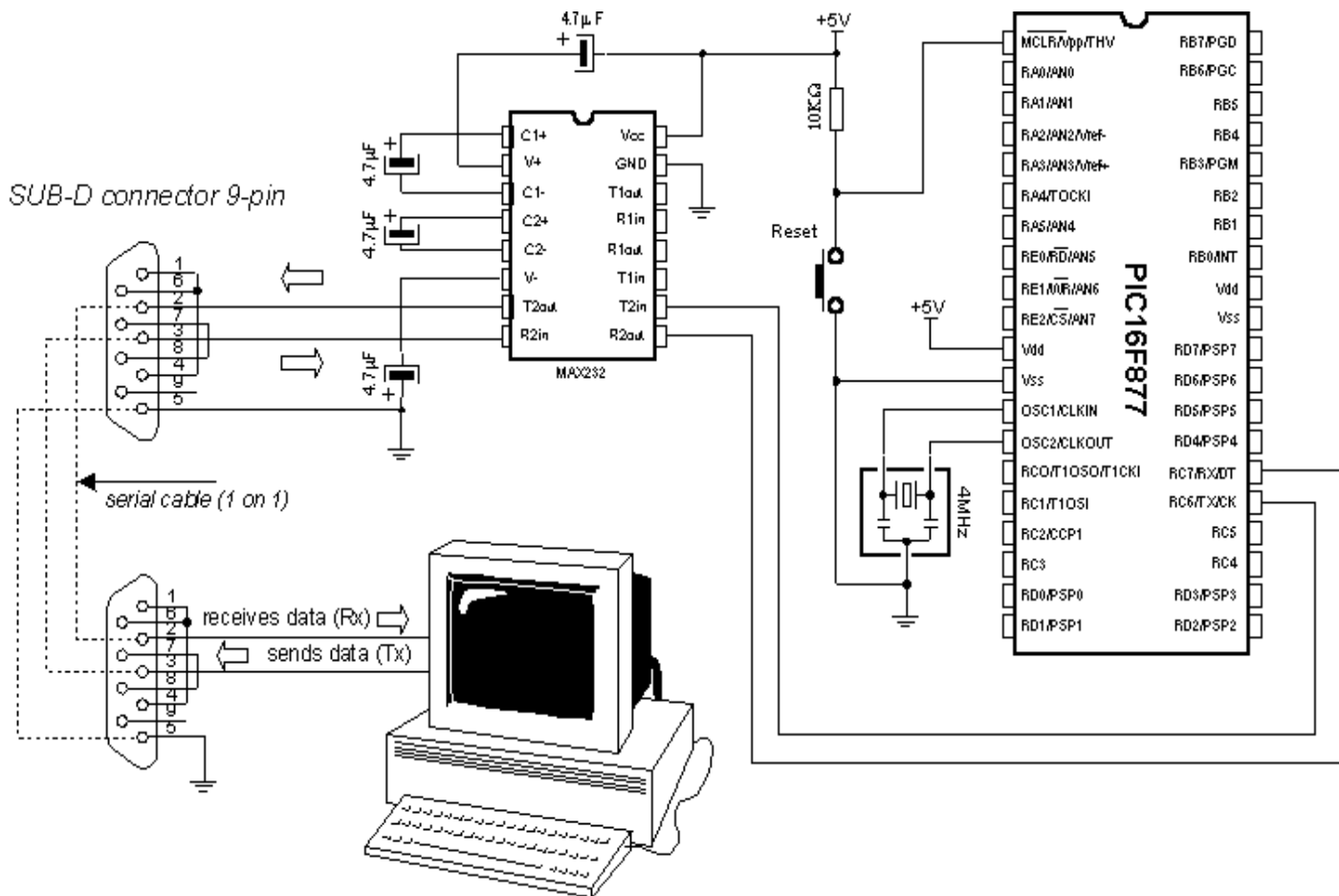
HC_Latch=1
pauseus 5
HC_Latch=0

return

```

6.9 Software serial communication

The easiest way to transfer data between the microcontroller and some other device (for example, PC or another microcontroller) is via RS-232 communication port. This type of communication provides serial asynchronous data transfer over 2 lines (Tx for transmitting and Rx for receiving) within 10m range. In this sample, instructions *Serin* and *Serout* are used for creating the software serial communication. Besides voltage level of the signal (RS-232 line interface MAX232 has a role to adjust signal levels on the microcontroller side, i.e. to convert RS-232 voltage levels +/- 10V to TTL levels 0-5V and vice versa) RS-232 features format and transfer rate. Transfer format is 8 data bits, no parity bit and one stop bit, while transfer rate is 2400 bauds.



```

Program: UARTsw.BAS

Include "modedefs.bas" ' Modes of transfer used by
                        ' instructions SERIN and SEROUT

symbol S0 = PORTA.3 ' Pin used for sending data
symbol SI = PORTB.0 ' Pin used for receiving data

B0 var byte ' Variable used by instructions SERIN and
            ' SETOUT for storing the data

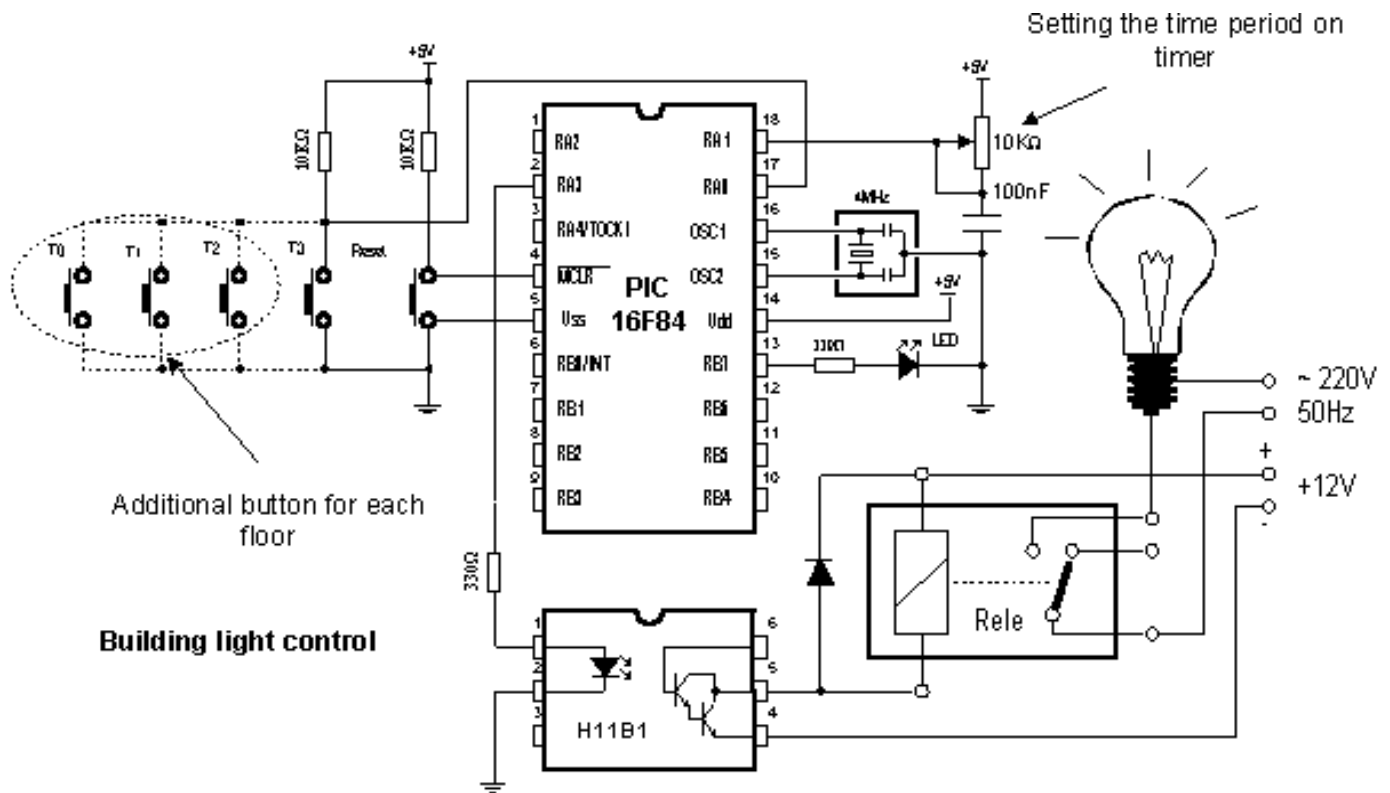
Main:
  Serin SI,T2400,B0 ' Received data is in B0
  Serout S0,T2400,[B0] ' Send the data from B0
  Goto Main ' repeat loop
end ' End of program

```

The program above uses *Serin* and *Serout* instructions for sending and receiving data. Data received via *Serin* instruction is stored into variable B0 and sent back to PC via *Serout* as a confirmation of successful transfer. Any microcontroller I/O pin can be used for described data transfer.

6.10 Building light control

Building light control is a very simple device that is realized using the microcontroller technology lately. The principle is simple - pressing the button turns on the light in the building for a time period T . Upon that time, all lights turn off. Variable T is defined with potentiometer. It is possible to determine for how long will the light be on by reading the potentiometer.



ATTENTION!!!

This sample works with public power supply, so all the necessary measures of precaution must be taken.

```

symbol Button = PORTA.0      ' Buttons su na pinu RA0
symbol Time_in = PORTA.1     ' Potentiometer for setting the
                               ' timer
symbol Light = PORTA.3       ' Output for light
symbol LED = PORTB.7         ' Output for control LED diode
B0 var byte                 ' Temporary variable used by instr. POT
B1 var byte                 ' Temporary variable used by instr. BUTTON
i var byte                  ' Variable in FOR...NEXT instruction
TRISA = %00000111           ' Pins RA0,1,2 are input
TRISB = %01111111           ' Pin RB7 is output
low Light                   ' Turn off the light
low LED                      ' Turn off the control LED diode
Main:                          ' Beginning of the program
B0 = 0
pot Time_in,255,B0          ' Time period for which the light is
                               ' on
B1 = 0
button Button,0,255,0,B1,1,Lite      ' If Button=0
                                       ' turn on the light
pause 50                     ' 50 ms pause
goto Main                     ' Jump to beginning
Lite:
high Sijalica                 ' Turn on the light
high LED                      ' Turn on the control LED
for i=0 to 60 + B0           ' If B0 = 0 Light is on for 1 min
pause 1000                   ' If B0 = 255 Light is on for 5 min
next i
low Light                     ' Turn off the light
low LED                      ' Turn off the control LED diode
goto Main                     ' Jump to beginning
End                          ' End of program

```

Chapter 7

SAMPLES WITH PIC16F877 MICROCONTROLLER

Introduction

[7.1 Keyboard](#)

[7.2 Driver for seven-segment displays - MAX7912](#)

[7.3 LCD display](#)

[7.4 Serial EEPROM](#)

[7.5 RS-485](#)

[7.6 12-bit A/D converter LTC1290](#)

[7.7 12-bit D/A converter LTC1257](#)

[7.8 16-bit electrical current D/A converter AD421](#)

[7.9 Real time clock PCF8583](#)

[7.10 Digital thermometer DS1820](#)

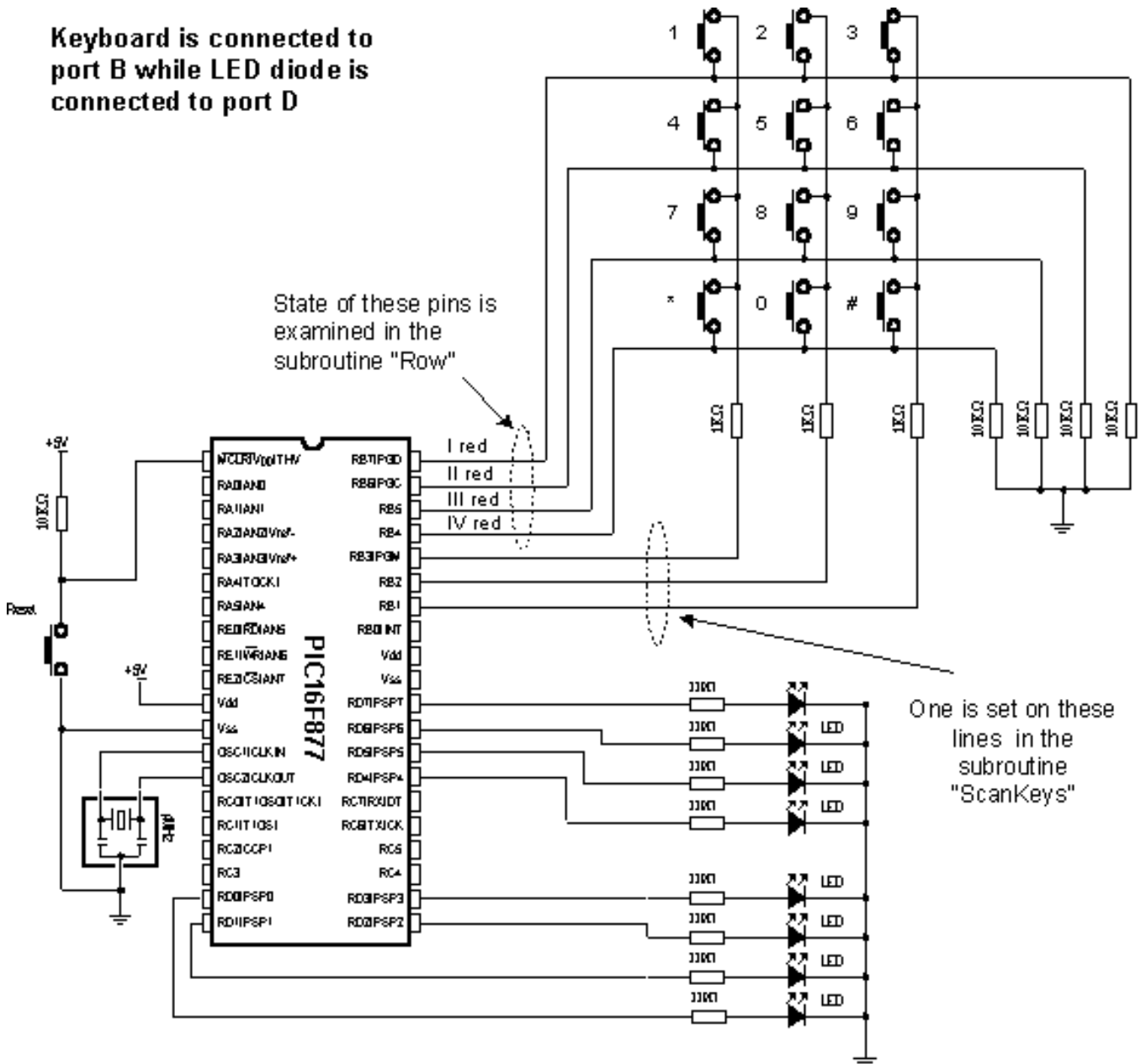
Introduction

This chapter gives detailed examples of connecting PIC16F877 microcontroller to peripheral components and appropriate programs written in BASIC. All of the examples contain electrical connection scheme and program with comments and clarifications. You have the permission to directly copy these examples from the book or download them from the web site <http://www.mikroelektronika.co.yu/>.

7.1 Keyboard

In more demanding applications that require greater number of buttons, it is possible to use buttons connected in matrix to keep microcontroller I/O lines free. The following sample includes scheme of connecting the keyboard and accompanying program which reads keyboard keys and prints the read value on LED diodes of port D.

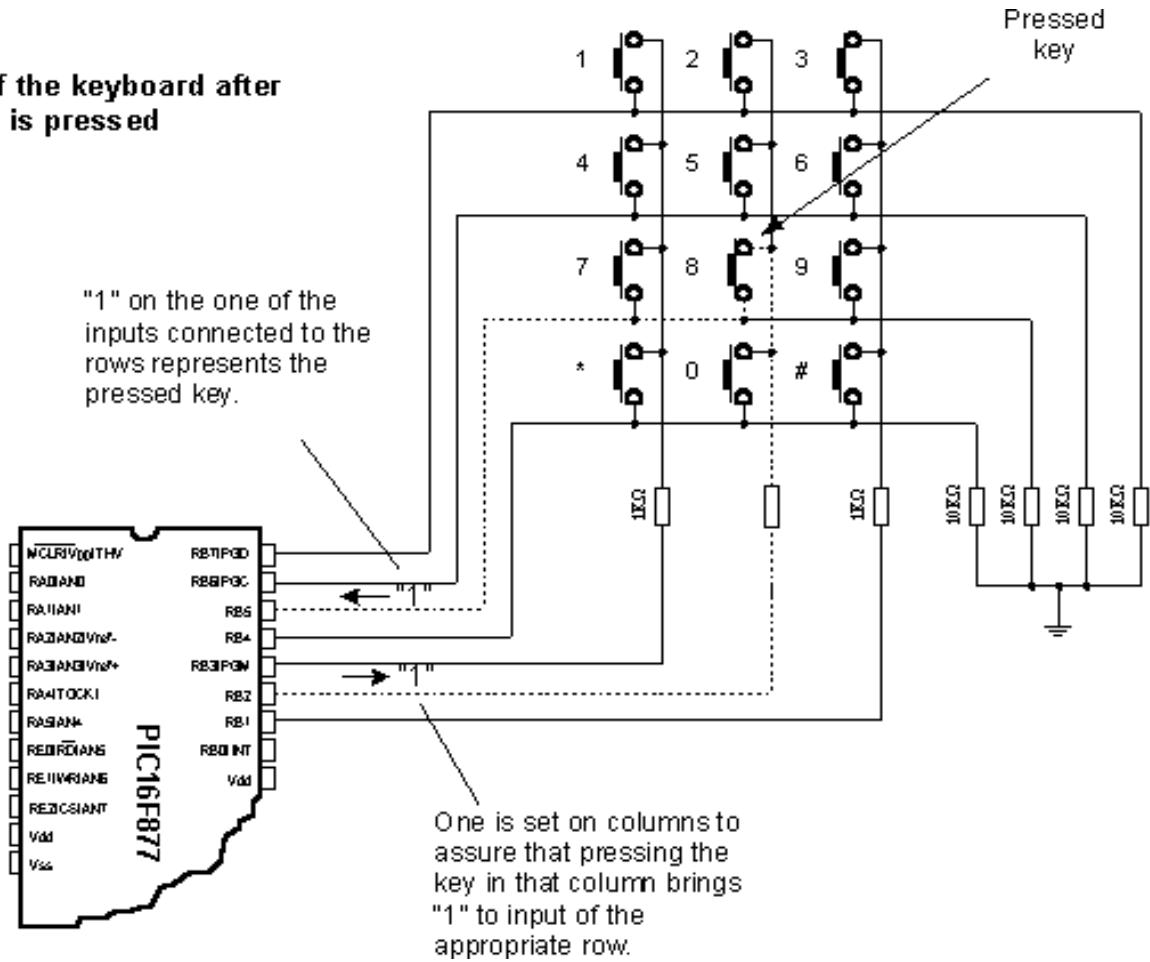
Keyboard is connected to port B while LED diode is connected to port D



The keys are connected into shared rows and columns. 10K resistors between input pins and the ground determine the state of input pins when the key is not pressed. It means that the logical zero is on input pins when the keys are not pressed. In order to avoid short-circuits between two pressed keys, 1K resistor is added to each row.

Reading the keyboard is done by subroutine "ScanKeys". The keyboard is connected to port B, it's pins being designated as input for rows (RB7, RB6, RB5 and RB4) and output for columns (RB3, RB2 and RB1).

State of the keyboard after the key is pressed



The program sets value of the last read key on port D. If none of the keys is pressed all diodes of port D are on. "*" and "#" are represented with values 10 and 11.

The greatest task is on the subroutine *ScanKey*. It sets logical one on keyboard columns and then calls the subroutine *Row* which checks if any of the 4 keys in that columns is pressed (which is signaled by variable *Flag*).

In case that one of the keys from the column is pressed, variable *KeyPress* takes value from 0 to 3 (zero for the first row of that column, one for the second row of that column, etc.). By calling the appropriate *Lookup* table, real value of the key is stored into variable *Result* and then to variable *OldResult* where from it is displayed on port D. In case that no key is pressed value of variable is 12.



```
OldResult    Var   Byte   ' Previously read character
Flag         Var   Bit    ' temporary var.
KeyPress     Var   Byte   ' Pressed key
Result       Var   Byte   ' The read character
TRISB=%11110000      ' Pins from RB0 to RB3 are output
                  ' Pins from RB4 to RB7 are input

PORTD=%11111111      ' All diodes are on at start
Result=$FF
OldResult=$FF

Main:
                  ' Beginning of the program
PORTD=OldResult     ' Display the last pressed key
                  ' on port D
gosub ScanKeys      ' Read the keys of the keyboard
if Result = OldResult then Main ' Same character ?
if Result = 12 then Main      ' None is pressed
goto Main           ' Repeat loop

ScanKeys:
KeyPress=0          ' clear keypress
PORTB=%1000         ' Choose the 1st column RB3 = 1
gosub Row           ' check the rows
if Flag=1 then FirstColumn ' If the key is pressed assign it
                  ' a value from look up table
PORTB=%0100         ' Choose the 2nd column RB2 = 1
gosub Row           ' check the rows
if Flag=1 then SecondColumn

PORTB=%0010         ' Choose the 3rd column RB1 = 1
gosub Row           ' check the rows
if Flag=1 then ThirdColumn

Result=12           ' None of the keys is pressed
return

FirstColumn:
                  ' If key is in the 1st column
lookup KeyPress,[1,4,7,10],Result
OldResult = Result
return

SecondColumn:
                  ' If key is in the 2nd column
lookup KeyPress,[2,5,8,0 ],Result
OldResult = Result
return

ThirdColumn:
                  ' If key is in the 3rd column
lookup KeyPress,[3,6,9,11],Result
OldResult = Result
return

Row:
Flag=1              ' set flag to 1 in case that the
```

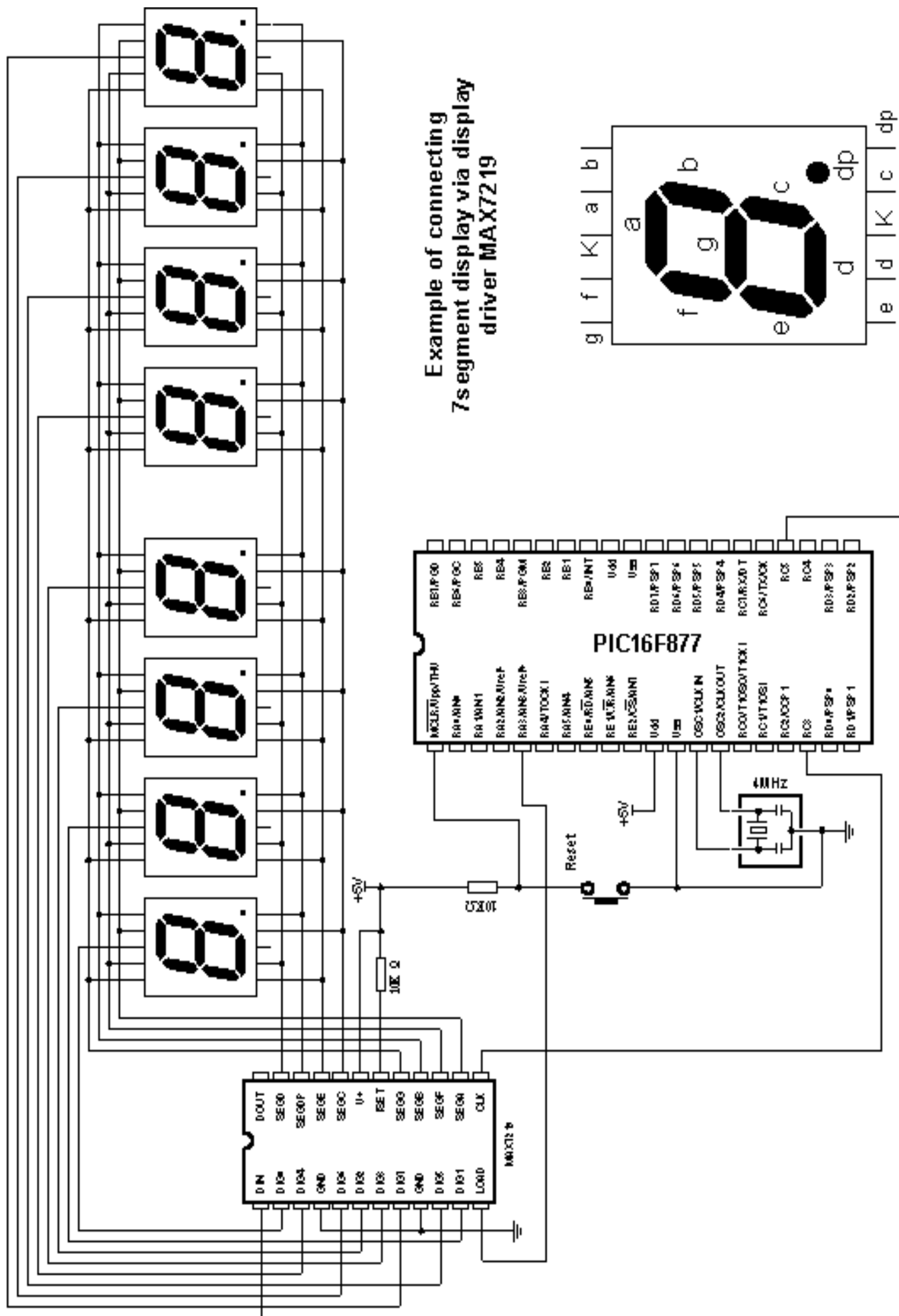
7.2 Driver for seven-segment displays - MAX7912

If a PIC16F84 or some similar microcontroller is programmed only to work with seven-segment displays (in multiplex mode) then it could be called "driver". If we supply it with option to communicate, we have a complete driver. If all that is realized directly in silicon while creating the "driver", we get full-fledge drivers that can be sold as independent electronic components.

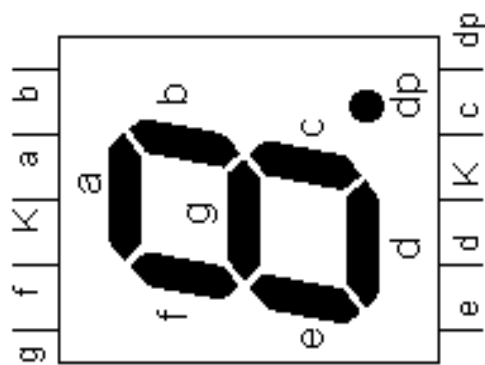
Question "why use drivers and not multiplexing the digits" is easy to answer with another question "what in case that we need 6 groups of 4 digits display?". It would require programmer to take care of multiplexing $4 \times 6 = 24$ digits. If the program in question is complicated, time necessary to write and adjust such a program might be more expensive solution than buying a separate driver.

There is a great variety of drivers and we will use MAX7912 in this sample. It can refresh 8 displays with option of configuring light intensity, while data transfer is serial, requiring small number of microcontroller pins. Anyhow, using the driver minimizes the work with seven-segment displays.

Working with driver is simple. There are certain registers which get necessary values via SPI communication. Address value is stored into variable *TxAddr* and data is stored into *TxData*. Subroutine *Send_Data* transfers address and data to driver. Before the first transfer, driver should be initialized by subroutine *Init_MAX* which is called only at the beginning of the program. The picture below shows the connection scheme and the sample program for printing the numbers 12345678 on displays follows.



Example of connecting 7s segment display via display driver MAX7219





```

Include "modedefs.bas"      ' Modes of data transfer used by
                              ' instruction SHIFTOUT
TxAddr var byte             ' Variable for storing the
                              ' address of reg. in MAX7219
TxData var byte            ' Var. for sending data
WO var word                 ' Temp. var. of word type
symbol MAX_Data = PORTC.5   ' Line for data input is connected
                              ' to pin RC5
symbol MAX_Clk = PORTC.3    ' Clk line is connected to pin RC3
symbol MAX_Load = PORTA.3   ' Load line is connected to pin RA3

TRISA = 0                    ' All pins of port A are output
TRISC = %11010011           ' 0,1,4,6,7 input: 2,3,5 output

MAX_Load = 1                 ' Disable access to MAX7219
Main:
gosub Init_MAX              ' Initialize MAX7219
WO = 1234                   ' Number displayed on the first
                              ' 4 digits
gosub Displ                 ' Print the data on the first 4
                              ' digits
WO = 5678                   ' Number displayed on the second
                              ' 4 digits
gosub Disp2                 ' Print the data on the second
                              ' 4 digits
Loop: goto Loop              ' remain in the loop

Init_MAX:                    ' Initialization MAX7219
TxAddr = $09                 ' BCD mode for decoding the digits
TxData = $ff
gosub Send_Data
TxAddr = $0a                 ' Intensity of display light
TxData = $0f
gosub Send_Data

TxAddr = $0b                 ' Refreshing the display
TxData = $07
gosub Send_Data

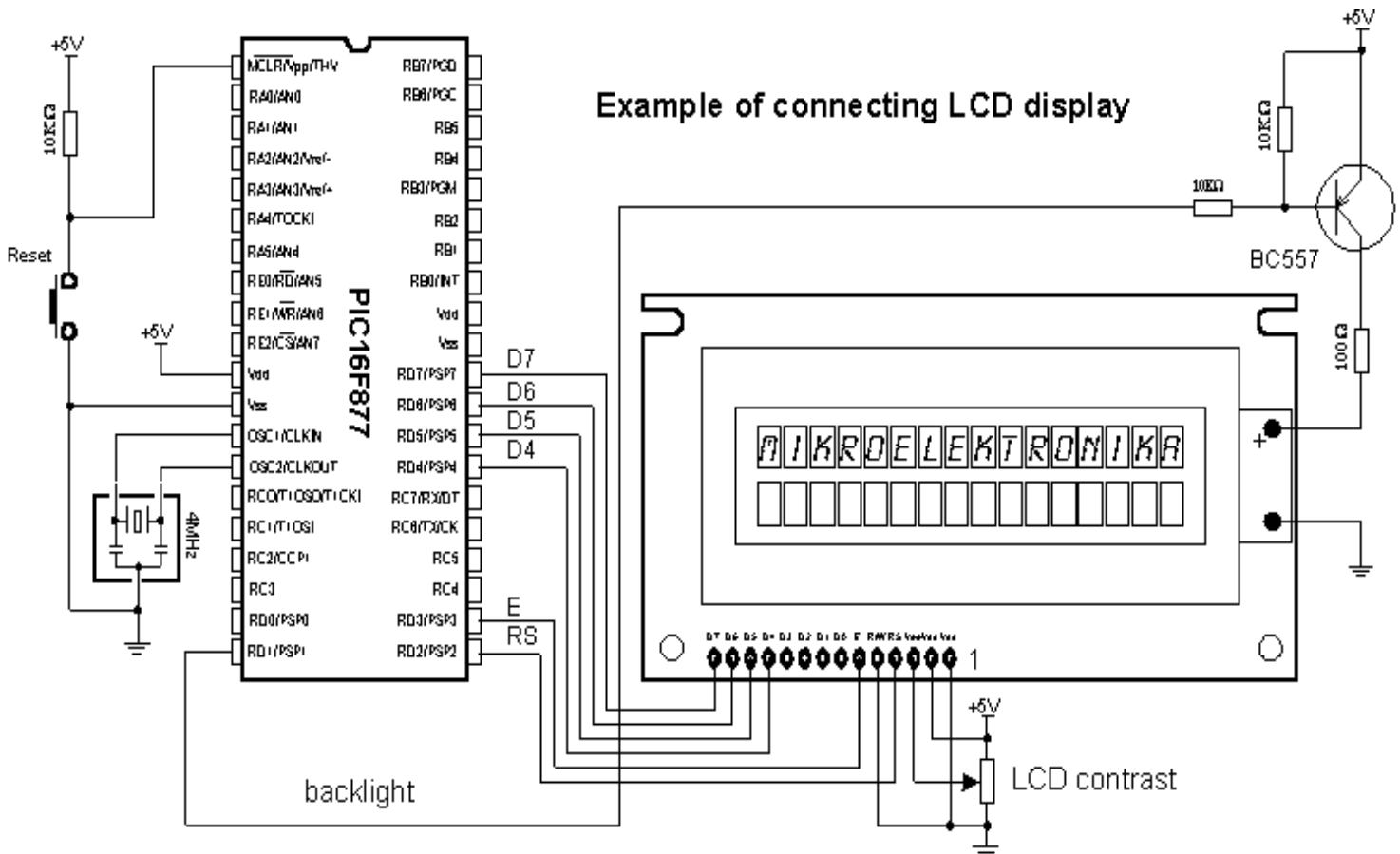
TxAddr = $0c                 ' Turn on the display
TxData = $01
gosub Send_Data

TxAddr = $00                 ' No test
TxData = $ff
gosub Send_Data
return

Displ:                        ' Print data on the first 4 digits
TxAddr = $01                 ' Print data on the first digit
TxData = WO dig 3           ' "thousands" digit
gosub Send_Data
```

7.3 LCD display

One of the best solutions for devices that require visualizing the data is the "smart" LCD display. Printing the data on this type of display is performed on the dot segments arranged to form a row. Segment dimensions are 7x5 dots and one row can consist of 8, 16, 20 or 40 segments. LCD display can have 1, 2 or 4 rows. LCD can be connected to a microcontroller via 8-bit or 4-bit bus (4 or 8 lines).



Besides these, there are control lines E (enable), R/W (read/write) and RS (register select) for a total of 7 lines. R/W signal is on the ground, because there is one-way communication toward the LCD display. Some displays feature built-in back-light that can be turned on with RD1 pin via PNP transistor BC557.

The following program uses the BASIC instruction *Lcdout* for printing the data on LCD display. At the beginning of the program DEFINE directives are used to configure the LCD display.

```

DEFINE LCD_DREG    PORTD  \ I/O port where LCD is connected
DEFINE LCD_DBIT    4
DEFINE LCD_RSREG   PORTD
DEFINE LCD_RSBIT   2      \ Register select pin
DEFINE LCD_EREG    PORTD
DEFINE LCD_EBIT    3      \ Enable pin
DEFINE LCD_BITS    4      \ 4-bit data bus
DEFINE LCD_LINES   2      \ LCD has two character lines

high    PORTD.1        \ turn on backlight

Main:                                          \ Beginning of the program
low     PORTD.1        \ turn on backlight
lcdout  $fe,1          \ Clear the LCD screen
\ Print "mikroElektronika" in the first line
lcdout  "mikroElektronika"
pause   2000           \ 2 sec pause

lcdout  $fe,1          \ Clear the LCD screen
\ Print "LCD example" in the first line
lcdout  "LCD example"
lcdout  $fe,$c0        \ Move to second line
\ Print " Second line " in the second line
lcdout  " Second line "

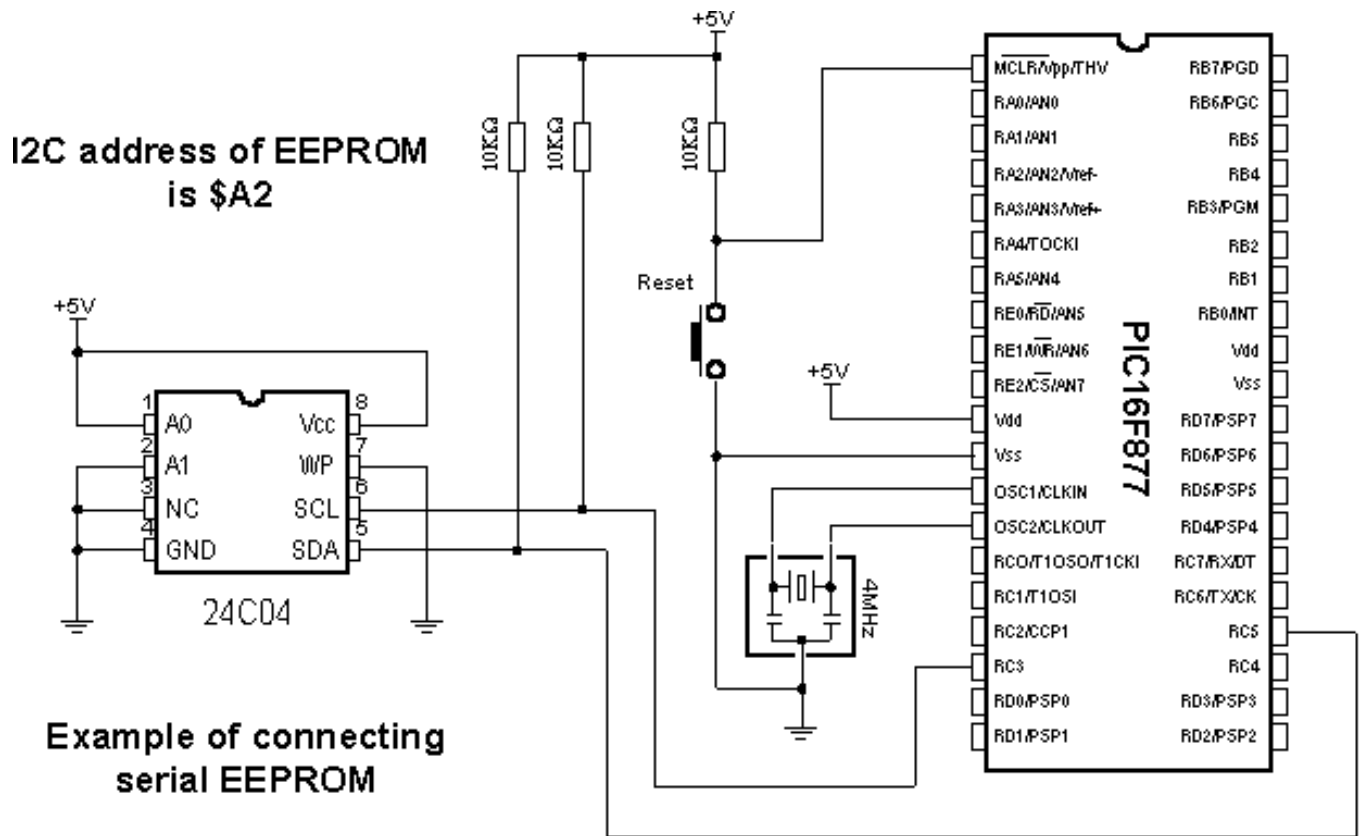
pause   2000           \ 2 sec pause
high    PORTD.1        \ turn off backlight
pause   2000

goto   Main           \ repeat the loop
end                                          \ End of program

```

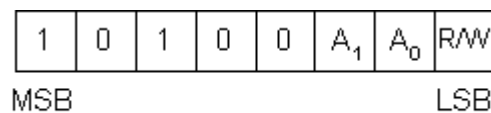
7.4 Serial EEPROM

Data (variables) used by microcontroller are stored in the RAM memory as long as there is microcontroller power supply. When the power is off, contents of RAM memory is gone. If it is necessary to keep the data for later use, it must be stored in permanent memory. One of the memories that can be used for this purpose is serial EEPROM. EEPROM stands for *Electrically Erasable and Programmable Read Only Memory*. Serial means that EEPROM uses one of the serial protocols (I2C, SPI, microwire) for communication with a microcontroller. This sample will deal with EEPROM from 24Cxx family which uses two lines and I2C protocol for communication with a microcontroller.



EEPROM is connected to microcontroller via SCL and SDA lines. SCL line is clock for synchronizing the data sent via SDA line. Frequency for transferring the data via SCL and SDA lines goes up to 1MHz.

BASIC instructions I2CWRITE and I2CREAD make reading and writing data to EEPROM pretty simple. To write or read data from EEPROM it is required to write an address of EEPROM (in this case \$A2) and address of data in memory as parameters of instruction. EEPROM address is formed in the following way :



EEPROM address

I2C communication allows connecting multiple devices on one line. Therefore, bits A1 and A0 have an option of assigning addresses to certain I2C devices by connecting the pins A1 and A0 to the ground and +5V (one I2C line could be EEPROM on address \$A2 and, say, real time clock PCF8583 on address \$A0). R/W bit of address byte selects the operation of reading or writing data to memory. More detailed data on I2C communication can be found in the technical documentation of any I2C device.

The following program uses the instruction I2CWRITE to write data from variable *EE_byteOut* to EEPROM, and then uses the instruction I2CREAD to transfer the same data from EEPROM to variable *EE_byteIn* and display it on port D diodes.


```

DEFINE LCD_DREG    PORTD  \ I/O which LCD is connected to
DEFINE LCD_DBIT    4
DEFINE LCD_RSREG   PORTD
DEFINE LCD_RSBIT   2      \ Register select pin
DEFINE LCD_EREG    PORTD
DEFINE LCD_EBIT    3      \ Enable pin
DEFINE LCD_BITS    4      \ 4-bit data bus
DEFINE LCD_LINES   2      \ LCD has two character lines

' Modes of data transfer used by instructions I2CWRITE i I2CREAD

symbol SDA = PORTC.5      \ I2C data pin

' 8-bit address of memory location in EEPROM
Addr      Var byte
' Var. containing data for writing in EEPROM
EE_ByteOut Var Byte
' Var. for containing data read from EEPROM
EE_ByteIn  Var Byte

TRISD = 0      \ Port D is output

Main:

lcdout "Write in EEPROM "
Addr = 2      \ Write data $55 to
EE_ByteOUT = $55      \ address 2
Gosub EWrite      \ Write data to EEPROM

Pause 1000      \ Brief pause

lcdout "Contents of ",Addr

Gosub ERead      \ Read data from address 2
lcdout $fe,$c0      \ Move to second line
lcdout EE_ByteIn      \ Print the read data

Loop: goto Loop      \ Remain in loop

' Write one byte in EEPROM

EWrite:      \ Write data
I2CWRITE SDA,SCL,%10101110,Addr,[EE_ByteOut]
Pause 10      \ 10 msec pause
Return

' Reading one byte from EEPROM

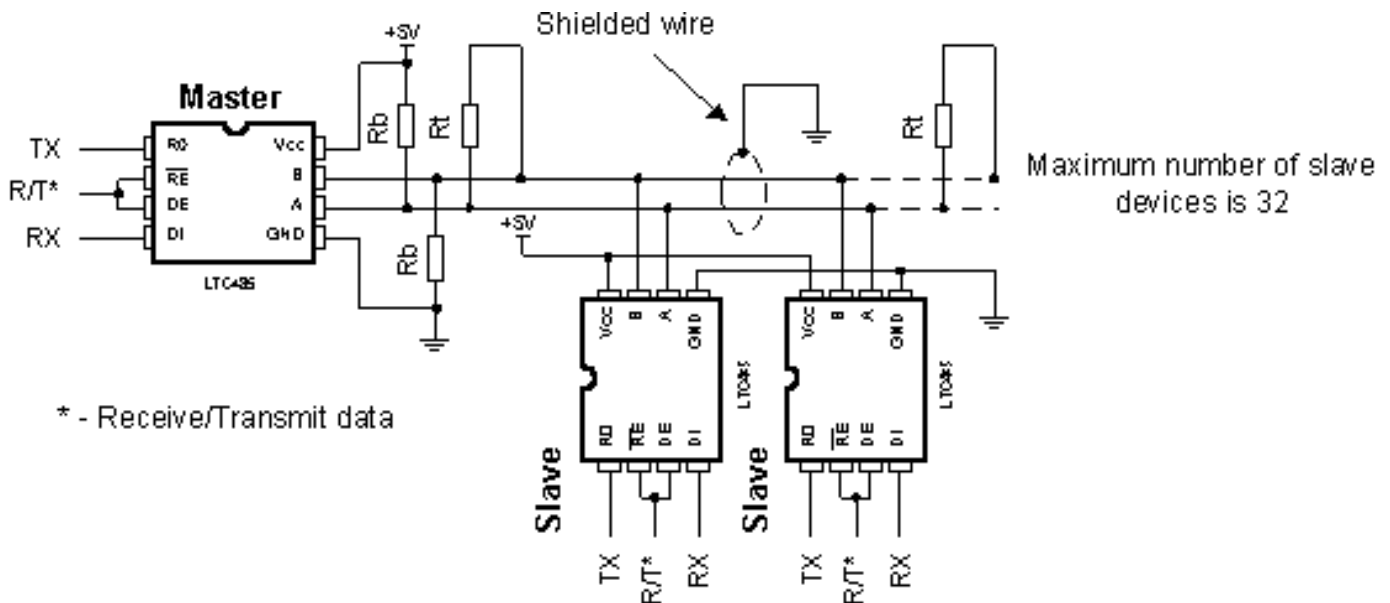
ERead:      \ Read data
I2CREAD SDA,SCL,%10101110,Addr,[EE_ByteIn]
Return
End      \ End of program

```

7.5 RS-485

Communication between two devices is easiest to achieve via serial RS232 communication. However, there are certain limitations of this type of communication. First of all, it is meant for local devices in 10-15m radius. The second drawback is that it can be used with only one device (for example, only one mouse can be connected to one COM port of PC).

RS-485 communication goes past these shortcomings, supporting up to 32 devices (even 128) with maximal network length of up to 1500m. The example of connecting 485 interface is shown on the picture below.

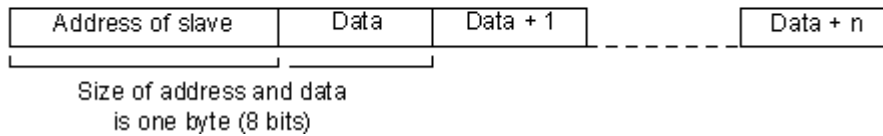


The role of the line interface LTC485 is similar to MAX232 interface with serial communication (adjusting the level). Using the shielded wire is not necessary, but is recommended for reducing the glitch.

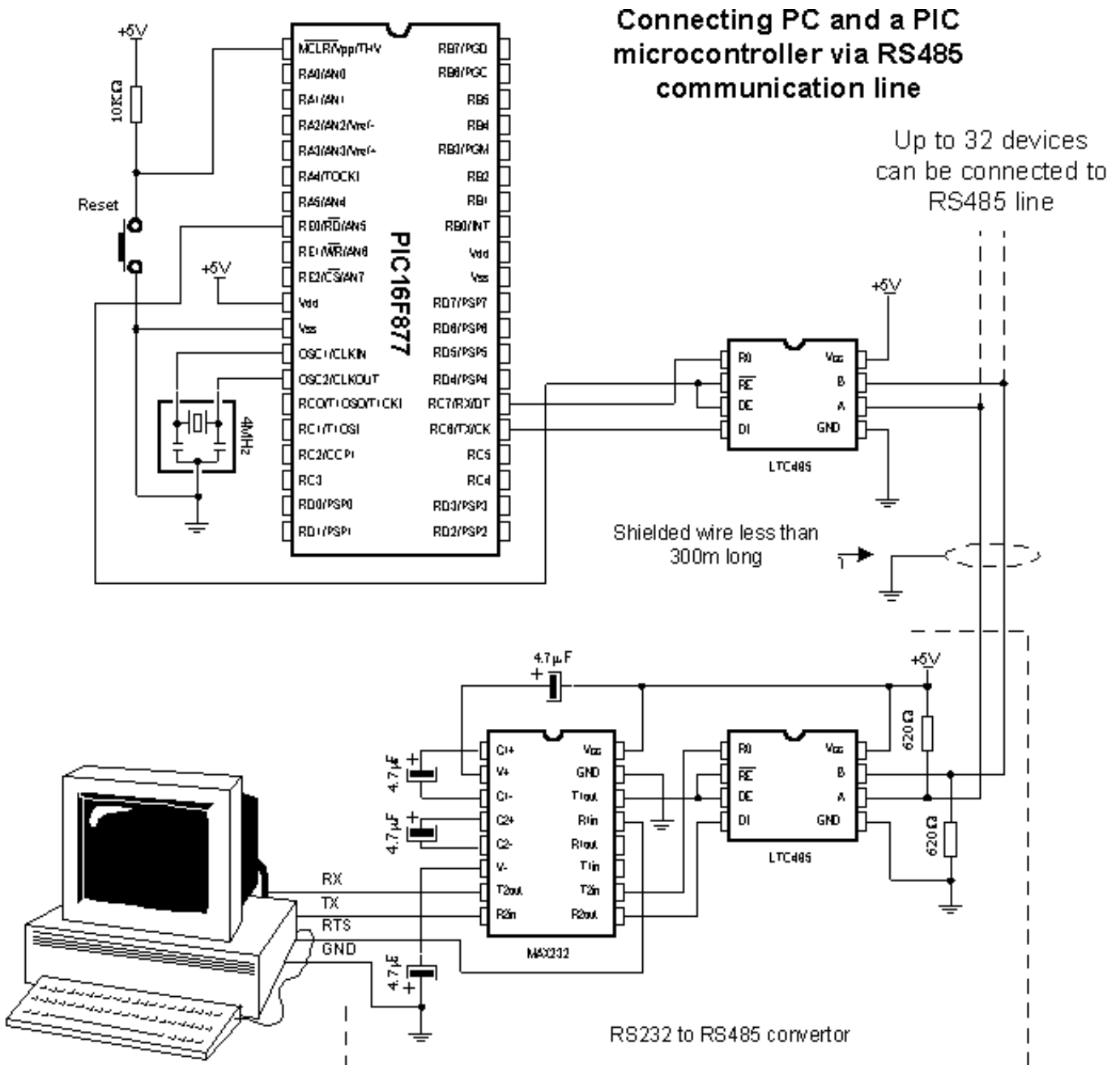
Ending resistor R_t should set the specific impedance on the line and usually has value about 100-200 ohms. With lines shorter than 300m, there is no need for ending resistor. Resistors R_b are placed in order to avoid incorrect reading when all the drivers are in receiving mode. Value of these resistors should be 680 ohms if they are placed on single device or 4K7 if they are placed on more than 8 devices in the network. RS485 is a *half-duplex* communication which uses 2 lines (marked as A and B) for data transfer. The information if data on the receiver side equals binary one or not can be read from the voltage difference of lines A and B. If voltage difference between lines A and B exceeds 200mV data is binary one and if it is lower than -200mV data is binary zero. This method proved very efficient for eliminating great deal of glitch which can occur on the line. As data transfer commences via two lines, organization of sending and receiving the data must be regulated to avoid interference of data on the line. Direction of data (sending or receiving) is determined by state of input pins RE and DE of line interface RS485. When states of pins are RE=0 and DE=1, line interface is set to the receiving mode. When states change to RE=1 and DE=0 line interface changes to the sending mode.

One of the typical solutions for RS485 communication is *master/slave* communication, where one *master* device takes over the control while other *slave* devices get the address which upon they can be called. Communication is initiated by *master* device sending the same N byte message to all *slave* devices and going to receiving mode. This N byte sequence contains the address of the specific *slave* device that is to receive data. Address is commonly located at the beginning of the sequence and consists of one byte (8 bits). When all of the *slave* devices receive the sequence, they extract the address from it and compare it to their own identification address. *Slave* device with matching address sends its address to the *master* device as a confirmation of successful transfer and receives data, while other *slave* devices return to the receiving mode. *Master* device receives the address of the mentioned *slaved* device and that ends the transfer between *master* and *slave* devices. In case that *slave* device does not send its address in some specified (*Time out*) period of time, *master* will declare that *slave* device inactive. This time out period ranges from 10ms to 100ms. The following picture shows format of the sequence that *master* sends to *slave* devices.

Sequence sent by *master* to *slave* devices via RS485 line



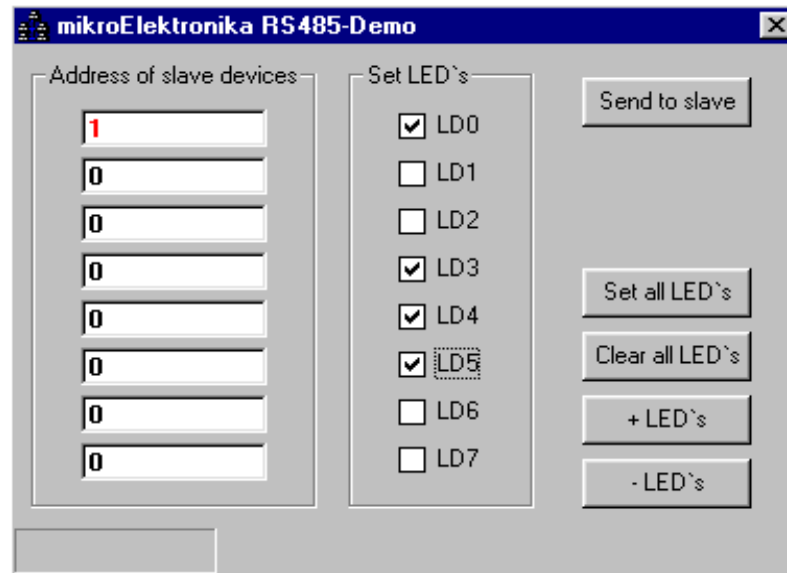
Newer systems consist of PC as *master* and multiple microcontroller *slave* devices. One such network is shown on the picture below.



As there is no output for RS485 on PC it is necessary to create RS232 to RS485 converter, while the microcontroller can be directly connected to LTC485 line interface. For physical connection of *master* and *slave* devices in RS485 network, RS485 interfaces other than LTC485 may be used - for example, SN75176 manufactured by *Texas Instruments*.

The following program is an example of connecting one *master* and up to 8 *slave* devices. Role of the *master* is given to the PC which is connected to the network via RS232/RS485 converter. *Slave* devices are PIC16F877 microcontrollers which are connected to the network via line RS485 interface LTC485 and each one of them owns unique address given as number in 0-7 range. In order to make data received from the *master* "visible", diodes are

connected to port B of every PIC. In the sample, *master* changes the states of LED diodes on the *slaves* devices. The following picture represents a window of PC software for communication with PICs via RS485 line.



When the button "Send to slave" is clicked, PC sends the address of *slave* device from the field "Address of slave device" to the RS485 network and it also sends the states of LED diodes as they are set in the group "Set LED's" (total of 2 bytes). All of the PICs receive the address and LED states, but only one of them actually changes LED diode state. That PIC answers by sending its address to PC in 50ms as confirmation of successful receipt and marks the end of the transfer between PC (*master* device) and PICs (*slave* devices). In case that PIC does not answer in the expected 50ms, PC will interpret it as if PIC with specified address is not present in the network and will report an error (data transfer is serial, asynchronous at 9600 bauds). Communication with PICs is software based via instructions SERIN2 and SEROUT. The instruction SERIN2 expects a data that matches the address of slave device (in this case, "1" ASCII). Upon receiving the address, next received data is stored into variable B0, its contents being displayed on port B LED diodes.

```

Program: RS485.BAS

' Modes of transfer used by instructions SERIN2 and SEROUT
Include "modedefs.bas"
' Pin used for sending data
symbol S0 = PORTC.6
' Pin used for receiving data
symbol SI = PORTC.7
' Pin for defining direction of data transfer
symbol RT = PORTE.0
B0 var byte ' Temp. var. of byte type
TRISE = $00 ' LED diode may be connected to port B
Main:
low RT      ' Microcontroller is in receiving mode
' If address received is "1", store data into B0
serin2 SI,84,20,Main,[wait ("1"),B0]
high RT     ' Microcontroller is in sending mode
pause 50    ' Wait for 50ms
' Return the address as confirmation of successful transfer
serout S0,T9600,["1"]
PORTB = B0  ' Display data on LED diodes
goto Main   ' Repeat loop
end         ' End of program

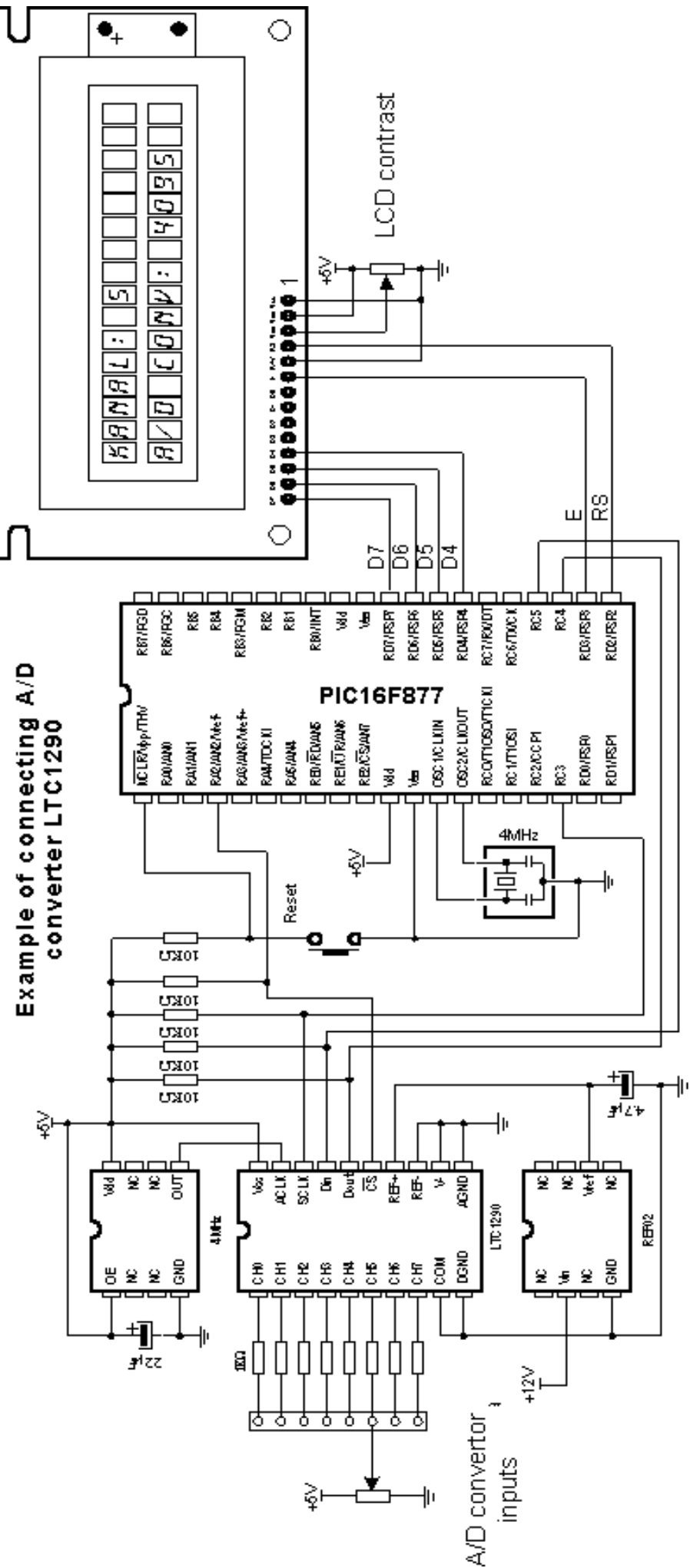
```

7.6 12-bit A/D converter LTC1290

LTC1290 is a 12-bit AD converter with serial access. It has 8 inputs and thus can measure up to 8 different signals and send them to microcontroller via SPI communication. SPI is abbreviation for *Serial Peripheral Device* and it is form of master-slave communication.

Microcontroller and AD converter are connected over 4 lines: MOSI, MISO, SS and SCK. The first line is named *Master Out Slave In* and it is used for instructions that master sends to slaves. The second line is reverse - *Master In Slave Out* for data master receives from slaves. Line SS (*Select Slave*) is used for defining which slave will master address, because there can be several slave devices connected to the same communication lines, but with different SS lines. The last line is named *Clock* indicating that it brings clock to communication between master and selected slave.

Data is sent in form of binary sequence via MISO or MOSI lines and master or slave reads it in the middle of CLK impulse creating synchrony between sent and received message. Although it might sound complicated, it is a pretty simple form of communication for creating software.





```
DEFINE LCD_DREG PORTD ' I/O port which LCD is connected
                        ' to
DEFINE LCD_DBIT 4
DEFINE LCD_RSEG PORTD
DEFINE LCD_RSBIT 2 ' Register select pin
DEFINE LCD_EREG PORTD
DEFINE LCD_EBIT 3 ' Enable pin
DEFINE LCD_BITS 4 ' 4-bit data bus
DEFINE LCD_LINES 2 ' LCD has two character lines

symbol CS = PORTA.2 ' ADC select pin
symbol Dout = PORTC.4 ' sending data from ADC to MCU
symbol Din = PORTC.5 ' sending data from MCU to ADC
symbol Sclk = PORTC.3 ' clock
ADval_MSB var byte ' higher bajt
ADval_LSB var byte ' lower bajt
B0 var byte ' temporary variable
B1 var byte ' temporary variable
W0 var word ' temporary variable

' I N I T I A L I Z A T I O N

PORTC = 0 ' port C is on log. 0 level
TRISA = %11111011 ' All pins are output except RA2
TRISC = %10 ' All pins are output except RC4
TRISD = 0 ' All pins are output
PIE1 = 0 ' SPI communication interrupt
                        ' disabled
INTCON = 0 ' All interrupts disabled
SSPCON = %21 ' SPI communication register
ADval_LSB = 0
ADval_MSB = 0
lcdout $fe,1 ' Print the text in the first line

lcdout "Channel: 5"

lcdout $fe,$c0 ' Print the text in the second line
lcdout "A/D conv:"
high CS ' Disable further LCD instructions

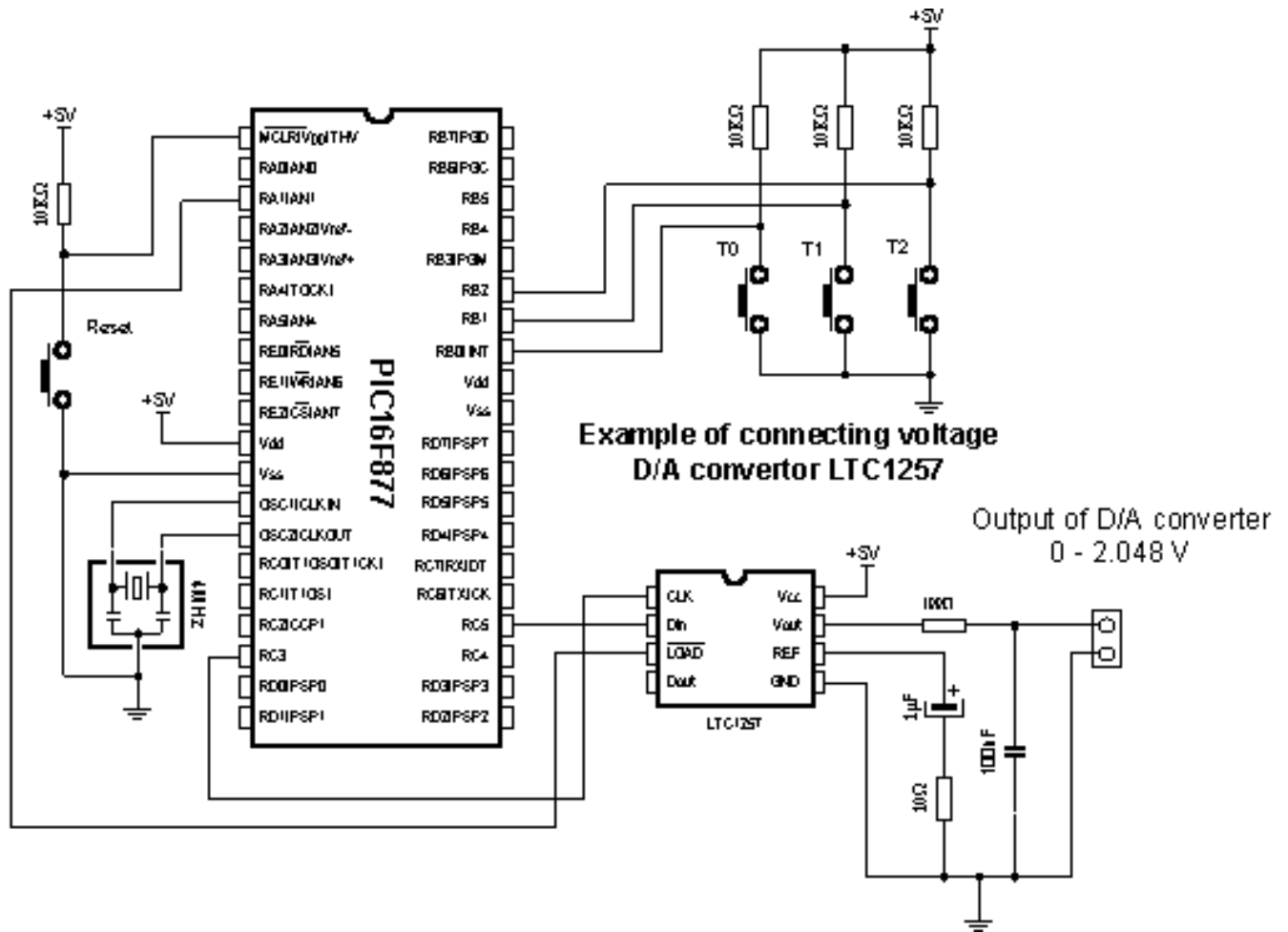
Main:
gosub Read_AD ' Read the value of AD input
gosub Display_ADval ' and print it on LCD
pause 50
goto Main ' Repeat all

Read_AD:
low CS ' Enable sending to ADC
SSPBUF = %11101111 ' Channel no.5 selected

L1: if SSPSTAT.0 = 0 then goto L1 ' Wait until transferred
ADval_MSB = SSPBUF ' Store higher byte of converted result
SSPBUF = 0 ' Send zero
```

7.7 12-bit D/A converter LTC1257

DA converters convert digital information into voltage (they have much more restricted application than AD converters). In this sample we use LTC1257, 12-bit DA converter with serial access, manufactured by *Linear Technology*. Type of communication with DA converter is SPI, which is explained in the previous sample. The program sends data containing the desired voltage to the AD converter which translates that data into voltage. We can check the program by measuring voltage on converter's output.





```
Include "modedefs.bas"      ' Modes of data transfer used
                             ' by instruction SHIFTOUT
TxData var word            ' Variable for transferring data
                             ' to shift register
BO var byte                ' Variable used by instruction
                             ' BUTTON
symbol LTC_Data = PORTC.5  ' Din line is connected to pin RC5
symbol LTC_Clk = PORTC.3   ' Clk line is connected to pin RC3
symbol LTC_Load = PORTA.1  ' Load line is connected to RA1
symbol Button0 = PORTB.0   ' Button T0 is connected to RB0
symbol Button1 = PORTB.1   ' Button T1 is connected to RB1
symbol Button2 = PORTB.2   ' Button T2 is connected to RB2
TRISA = 0                  ' Configuring I/O ports
TRISB = %11111111
TRISC = %11010111
LTC_Load = 1              ' Disable writing in DA converter

Main:
BO = 0 ' If button T0 is pressed jump onto Full
button Button0,0,255,0,BO,1,Full
BO = 0 ' If button T1 is pressed jump onto Half
button Button1,0,255,0,BO,1,Half
BO = 0 ' If button T2 is pressed jump onto Zero
button Button2,0,255,0,BO,1,Zero
goto Main                ' Jump to beginning
' Full scale on D/A coverter output has value 2.048 V

Full:
TxData = $fff           ' Write $fff in variable TxData
gosub Send_data         ' Write data in DA coverter
goto Main               ' Jump to beginning

Half: ' DA converter output value is 1.024 V which is exactly
      ' half of the full scale (2.048 V)
TxData = $7ff
gosub Send_data
goto Main

Zero: ' Voltage on DA converter output is 0 V
TxData = $0
gosub Send_data
goto Main

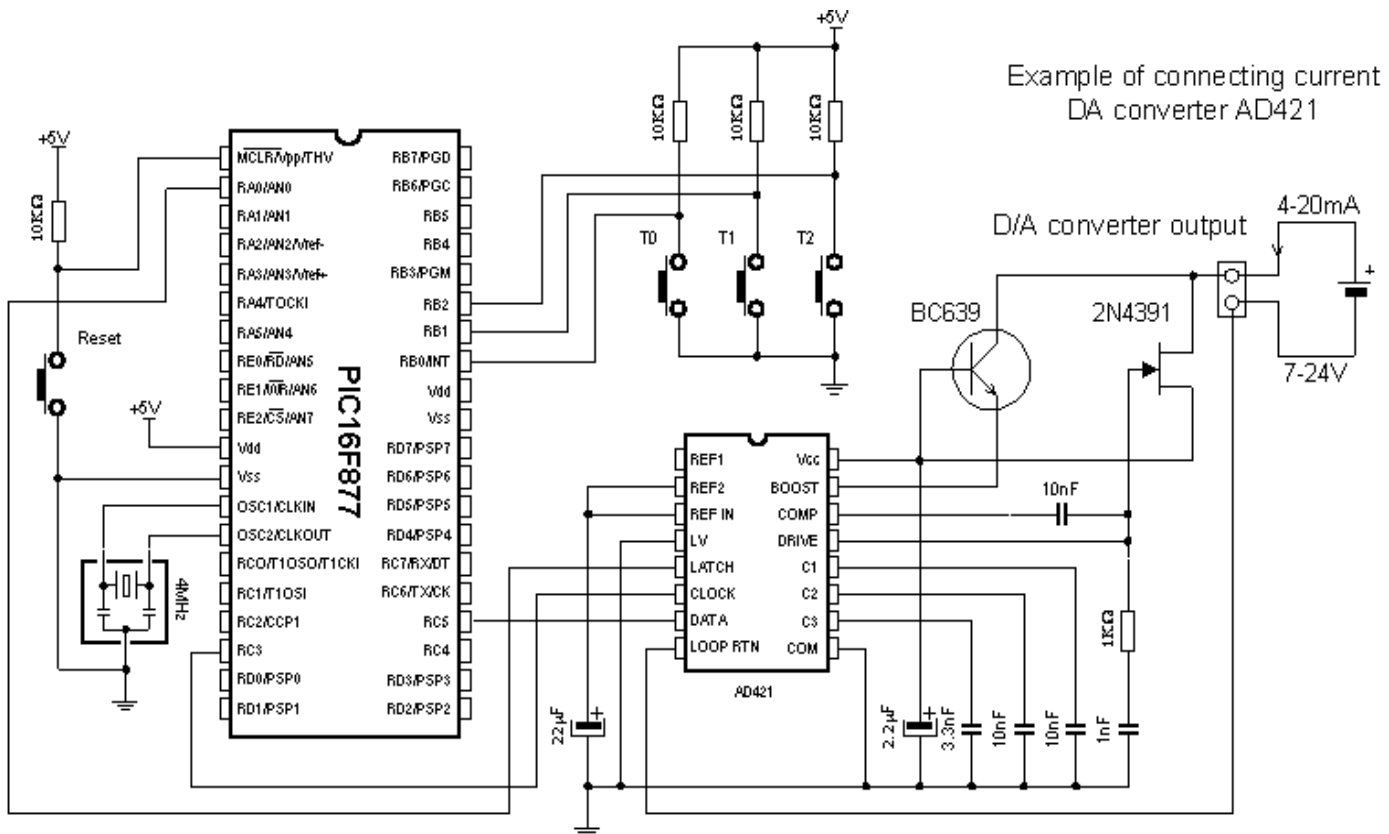
Send_Data:
' Transfer 12-bit data to SHIFT register of DA converter
shiftout LTC_Data,LTC_Clk,MSBFIRST,[TxData\12]

LTC_Load=0              ' Transfer data from SHIFT register
                        ' to LATCH of DA converter

Pauseus 10
LTC_Load=1
Return
End                    ' End of program
```

7.8 16-bit electrical current D/A converter AD421

When talking about DA converters we usually refer to voltage converters. However, there are electric current converters - they simply change the current at output instead of voltage. Current signals are very frequent in industrial applications due to their reliability in information transfer. Common range for electrical current signal in the industrial applications is 4-20mA. Lower limit value (4mA) allows line break or similar malfunctions to be noticed instantly and thus increases safety of signal transfer (when the line breaks or electronics fails, signal drops to 0 mA signaling the supervising system that the malfunction has occurred). In this sample we used 16-bit DA converter AD421 manufactured by *Analog Devices*. The program itself is quite simple and it's purpose is to set the lowest, medium and the highest value on the output of DA converter via 3 buttons.





```
Include "modedefs.bas"      ' Modes of data transfer used by
                             ' instruction SHIFFOUT
TxData var word            ' Variable for sending data to
                             ' shift register
B0 var byte                ' Variable used by instruction
                             ' BUTTON

symbol AD_Data = PORTC.5
symbol AD_Clk = PORTC.3
symbol AD_Cs = PORTA.0
symbol Button0 = PORTB.0   ' Button T0 is connected to RB0
symbol Button1 = PORTB.1   ' Button T1 is connected to RB1
symbol Button2 = PORTB.2   ' Button T2 is connected to RB2
TRISA = 0                  ' Configuring I/O ports
TRISB = %11111111
TRISC = %11010111
AD_Cs = 1                  ' Disable writing in DA converterer

Main:
B0 = 0                    ' If button T0 is pressed jump onto label Full
button Taster0,0,255,0,B0,1,Full
B0 = 0                    ' If button T1 is pressed jump onto label Half
button Taster1,0,255,0,B0,1,Half
B0 = 0                    ' If button T2 is pressed jump onto label Zero
button Taster2,0,255,0,B0,1,Zero
goto Main                ' Jump to the beginning

Full: ' Full scale of DA converter produces 20mA current at output
TxData = $ffff          ' Write $ffff in variable TxData
gosub Send_data         ' Write data in D/A converter
goto Main              ' Jump to the beginning

Half: ' At half of the scale current at output is 12 mA
TxData = $8000
gosub Send_data
goto Main

Zero: ' At beginning of the scale, current at the output is 4 mA
TxData = $0
gosub Send_data
goto Main

Send_Data:
AD_Cs=0                ' Enable writing in D/A converter
                       ' Transfer data from TxData to
                       ' SHIFT register of D/A converter
shiftout AD_Data,AD_Clk,MSBFIRST,[TxData]
AD_Cs=1                ' Transfer data from SHIFT register to
                       ' D/A converter Latch

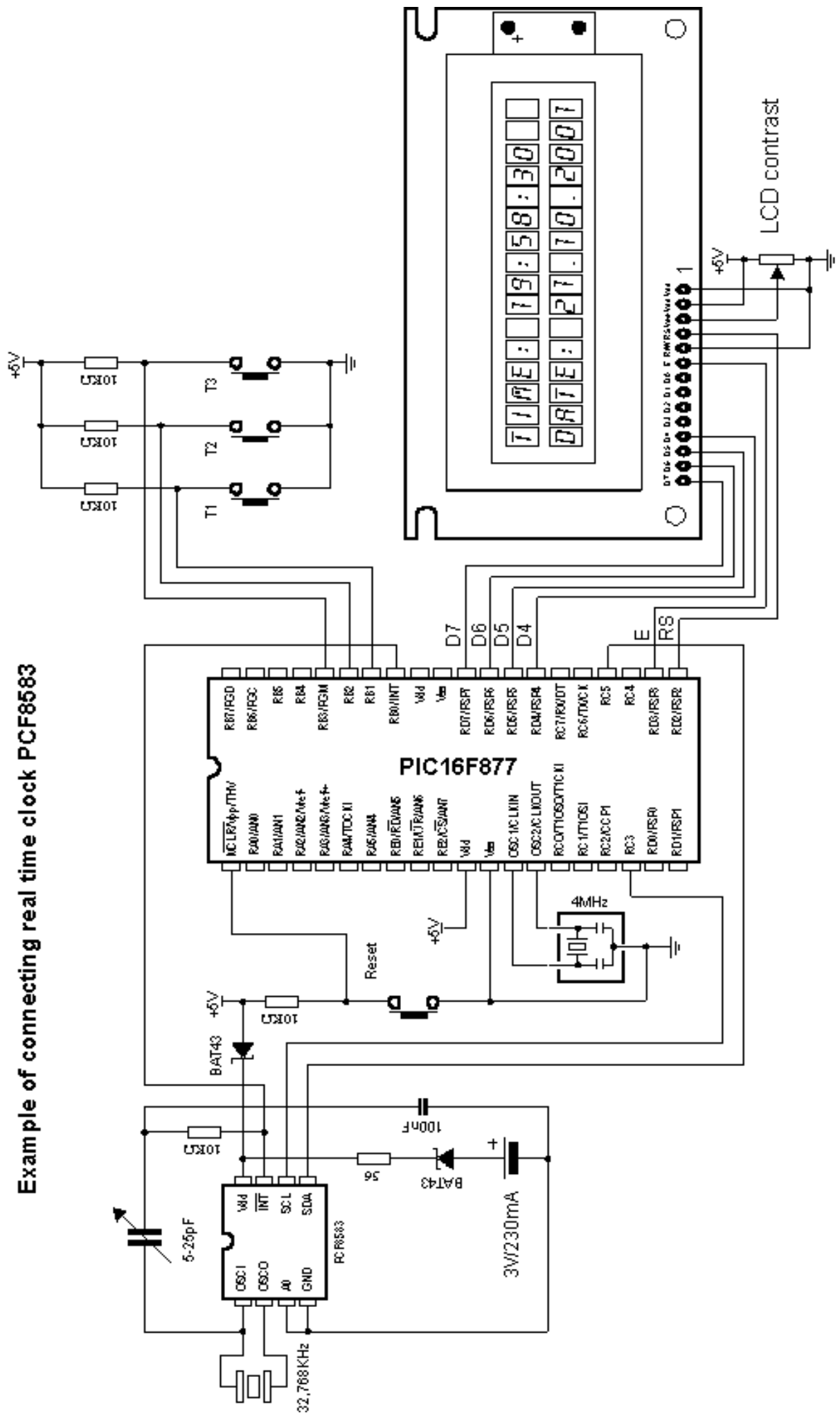
Return
End                    ' End of program
```

7.9 Real time clock PCF8583

Knowing the real time is of utmost importance in some projects. Let's take for example meteorological observations where we measure temperature, humidity and atmospheric pressure twice a day - these information, without the date and precise time when the measuring took place, would be incomplete. Unless we used real time clock, we would have to utilize microcontroller internal timers to create a second, minute, day, month and a year. Another drawback is the significant cumulative delay that will certainly amount until the end of the year. Even chips intended for special purpose (like the one used in the sample) exhibit significant delays.

For such purpose it is much more efficient to use special chip with it's own power supply, so that time always equals the real time. Accuracy of PCF8583 is about 1 second daily, so that maximal yearly deviation does not exceed 6 minutes (most applications do not require higher accuracy than this).

Example of connecting real time clock PCF8583



```

DEFINE LCD_DREG    PORTD ' I/O port with LCD
DEFINE LCD_DBIT    4
DEFINE LCD_RSEG    PORTD
DEFINE LCD_RSBIT   2    ' Register select pin
DEFINE LCD_EREG    PORTD
DEFINE LCD_EBIT    3    ' Enable pin
DEFINE LCD_BITS    4    ' 4-bit data bus
DEFINE LCD_LINES   2    ' LCD has two character lines
symbol Up = PORTB.3    ' Button Up is connected to RB3
symbol Down = PORTB.2  ' Button Down is connected to RB2
symbol Set = PORTB.1   ' Button Set is connected to RB1
symbol SCL = PORTC.3   ' I2C data pin
symbol SDA = PORTC.5   ' I2C clock pin

Sec    var bit
B0     var byte
S      var byte      ' Seconds
M      var byte      ' Minutes
H      var byte      ' Hours
D      var byte      ' Day
Mn     var byte      ' Month
Y      var byte      ' Year

OPTION_REG = $7f    ' Enable pull-up resistors for
                   ' PORTB

lcdout $fe,1      ' Clear the display

Main:
toggle Sec
I2CREAD SDA,SCL,%10100001,2,[S]
I2CREAD SDA,SCL,%10100001,3,[M]
I2CREAD SDA,SCL,%10100001,4,[H]
I2CREAD SDA,SCL,%10100001,5,[D]
I2CREAD SDA,SCL,%10100001,6,[Mn]
I2CREAD SDA,SCL,%10100001,6,[Y]

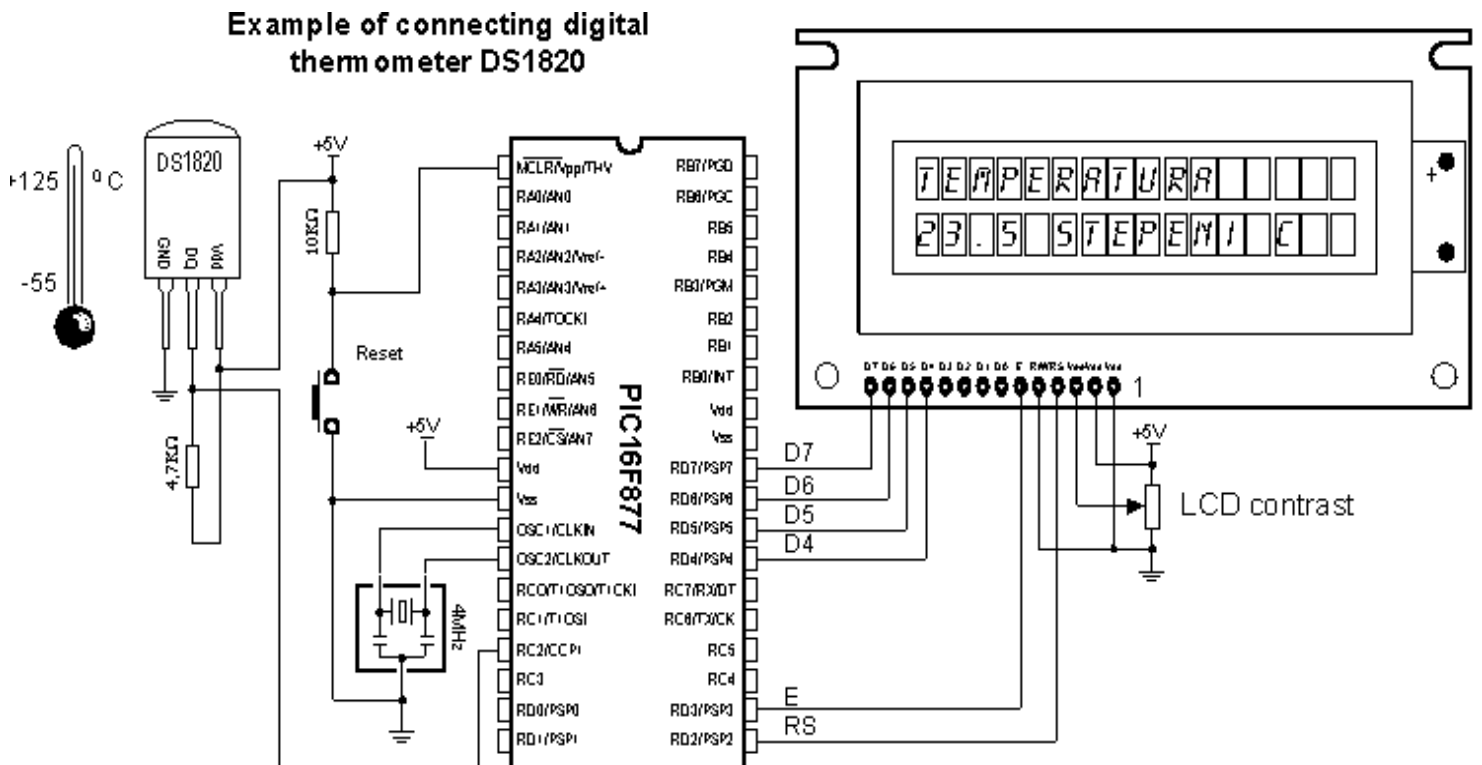
lcdout $fe,2
lcdout "Time: ", hex(H),":", hex(M)," ", hex(S)
lcdout $fe,$c0
lcdout "Date: ", hex(D),".", hex(Mn),".20",hex(Y)

if Sec = 0 then
    lcdout $FE,$8B," "
else
    lcdout $FE,$8B,":"
endif
goto Main
End                ' End of program

```

7.10 Digital thermometer DS1820

Temperature measuring is still one of the most common observations. Despite various means of measuring temperature (starting from temperature couples like PT100 all the way to NTC and PTC resistors) new, easier ways are being discovered. One of these new ways was presented by company *Dallas* with their new chip DS1820. It is a digital temperature sensor with only three pins. The advantage over the mentioned PT100 and NTC sensors which have output in millivolts or ohms (both signals must be converted into digital values in order to be handled and displayed) is that DS1820 has digital information in degrees of Celsius, practically solving the problem of analog electronics. Its only limitation is the temperature range, in this case $-55\text{ C} \sim +125\text{ C}$. Precision is 0.5 C and the measuring takes 200ms which is more than sufficient, considering the fact that the temperature change is a slow process.



```

DEFINE LCD_DREG      PORTD  \ I/O port with LCD
DEFINE LCD_DBIT      4
DEFINE LCD_RSREG     PORTD
DEFINE LCD_RSBIT     2      \ Register select pin
DEFINE LCD_EREG      PORTD
DEFINE LCD_EBIT      3      \ Enable pin
DEFINE LCD_BITS      4      \ 4-bit data bus
DEFINE LCD_LINES     2      \ LCD has two character lines
symbol DQ = PORTC.2      \ Dq line is connected to RC2

temperature Var Word      \ Var.for storing measured value
count_remain Var Byte    \ Count remaining
count_per_c Var Byte    \ Count per degree C
ADCON1 = 7                  \ Set PORTA and PORTE to digital
pause 100                  \ Pause for starting LCD

Main:
OWOut DQ, 1, [%CC, %44]    \ Start measuring temperature
Wait: OWIn DQ, 4, [count_remain] \ Check if it is finished
If count_remain = 0 Then Cekaj
OWOut DQ, 1, [%CC, %BE]    \ Read the measured temperature
OWIn DQ, 0, [temperature.LOWBYTE, temperature.HIGHBYTE,
Skip 4, count_remain, count_per_c]

' Display temperature in DEC

temperature = (((temperature >> 1) * 100) - 25) +
(((count_per_c - count_remain) * 100) / count_per_c)
Lcdout %fe, 1, DEC (temperature / 100), ".", DEC2
temperature, " C"

Pause 1000                \ Measure every second
Goto Main                  \ Repeat all
End                       \ End of program

```


Appendix A

PIC BASIC AND MPLAB

[Introduction](#)

- [A.1 Installation of the program / MPLAB](#)
- [A.2 Connection of PIC BASIC and MPLAB](#)
- [A.3 Toolbar](#)

Introduction

MPLAB is a Windows programming package that facilitates writing and the development of the program. The easiest way to describe it would be to characterize it as a development environment for some standard programming language intended for PC programming. Using MPLAB technically facilitates some of the operations which all the way up to the appearance of the IDE environment, were operating out of the command line with very big number of parameters. Nevertheless, out of different tastes, some programmers even today prefer standard editors and compilers operating out of the command line. In any case the written code is very manifest and provided with a relatively well-provided HELP menu (the abbreviation IDE was born out of the initials *Integrated Development Environment*).

A.1 Installation of the program / MPLAB



MPLAB is composed out of several different entities

- The grouping of the files belonging to the same project (*Project Manager*)
- The creation of the program and its elaboration (*Text Editor*)
- Simulator of the code whereby its work on the microcontroller is simulated.

Besides there exist support for Microchips products such as PICStart Plus i ICD (*In Circuit Debugger*). As this book doesn't rely upon them, they'll be mentioned as options only.

The minimal requirements in order to start up MPLAB on your computer are:

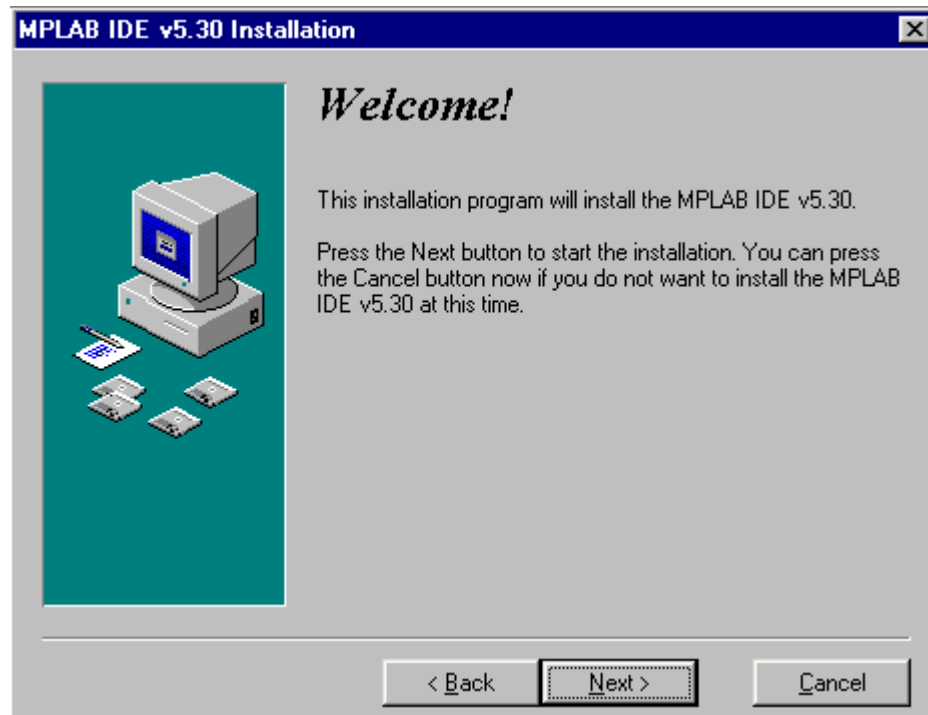
- Compatible PC of 486 class or higher
- Microsoft Windows 3.1x or Windows 95 and more recent Windows OS versions
- VGA graphic card
- 8MB of memory space (32 MB recommended)
- 20MB space on hard disk
- The mouse

To start MPLAB it is necessary to install it first, which is understood as a process of copying of MPLAB files from CD onto the hard disk of the PC. On each newly opened window there is button for going back to the previous window so mistakes should not represent any problem. The installation itself flows similarly as those of almost all Windows programs. The welcome screen pops up first and then you have the option choice and the installation menu in order to finally get the message that your installed program is ready to be started.

Steps in the installation:

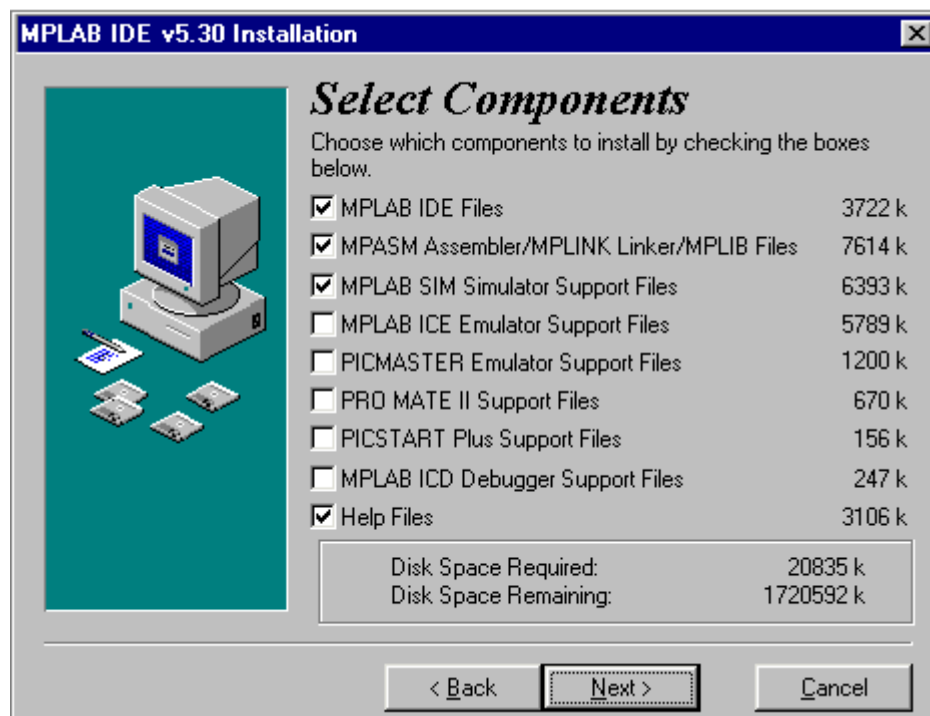
1. The starting of the Microsoft Windows
2. Put the Microchip CD disk into the CD ROM
3. Click onto the START in the lower left corner of the screen and choose the RUN option
4. Click onto the BROWSE and select CD ROM drive for your PC
5. On the CD ROM find the directory under the name of MPLAB
6. Click onto the SETUP.EXE and then on the OK button
7. Click once again on OK button in the RUN window

After these seven consecutive steps the installation will start. The following pictures explain the meaning of single steps in the installation process.



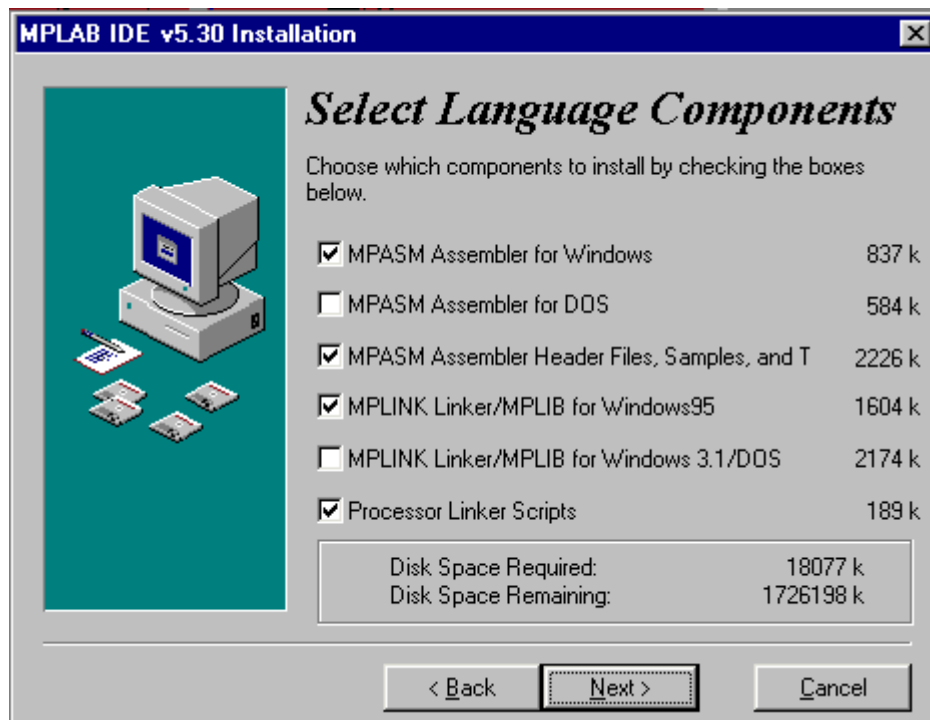
The WELCOME screen at the beginning of the installation

At the very beginning it is necessary to choose those components of MPLAB with which we are going to work. As it is supposed that there are no original Microchip's hardware additions such as programming devices or emulators, only the MPLAB environment, Assembler, Simulator and the instructions for use will be installed.



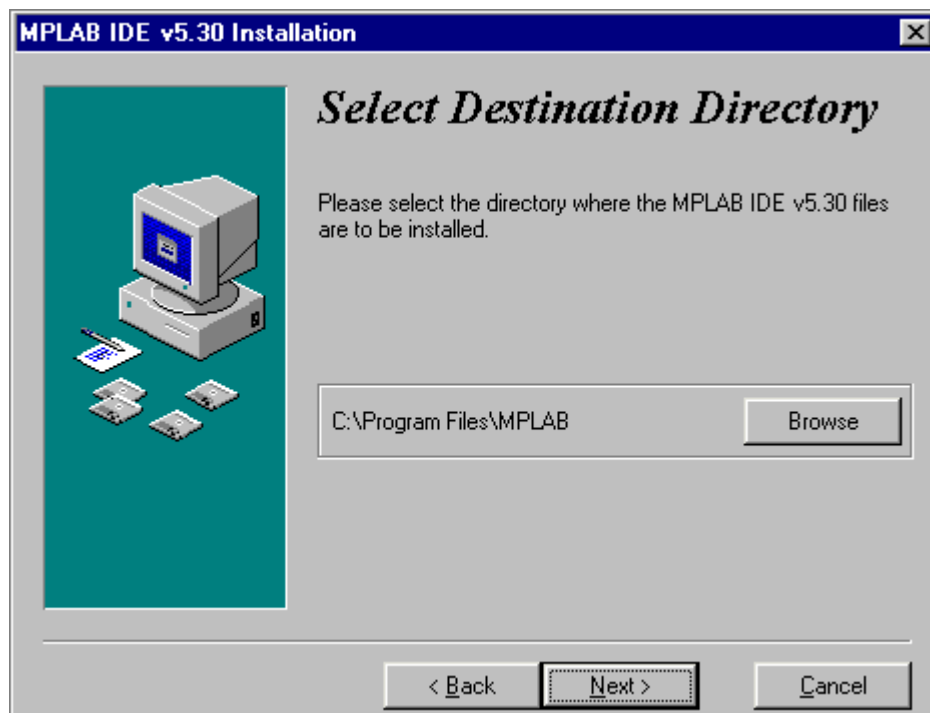
Selection of the components of the MPLAB development environment

The second supposition is that the OS will be Windows 95 (or some more recent version), so that in the selection of the assembler language is taken out everything that is connected to DOS operating system. However if you nevertheless wish to work in DOS, it is necessary to perform the deselection of all the options connected with Windows, and choose the corresponding DOS components.



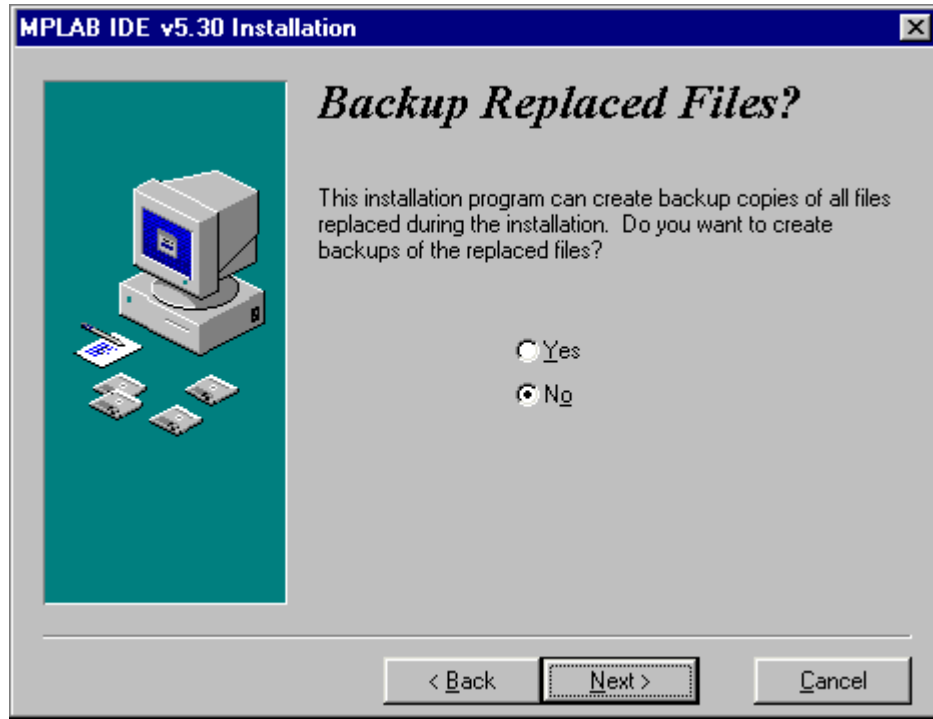
Selection of the assembler and OS

As it is normal for any program, MPLAB should be installed into a defined directory. This option can be changed into any directory on any hard disk of your PC. Unless you have some specific reason, it should be left on the selected location.



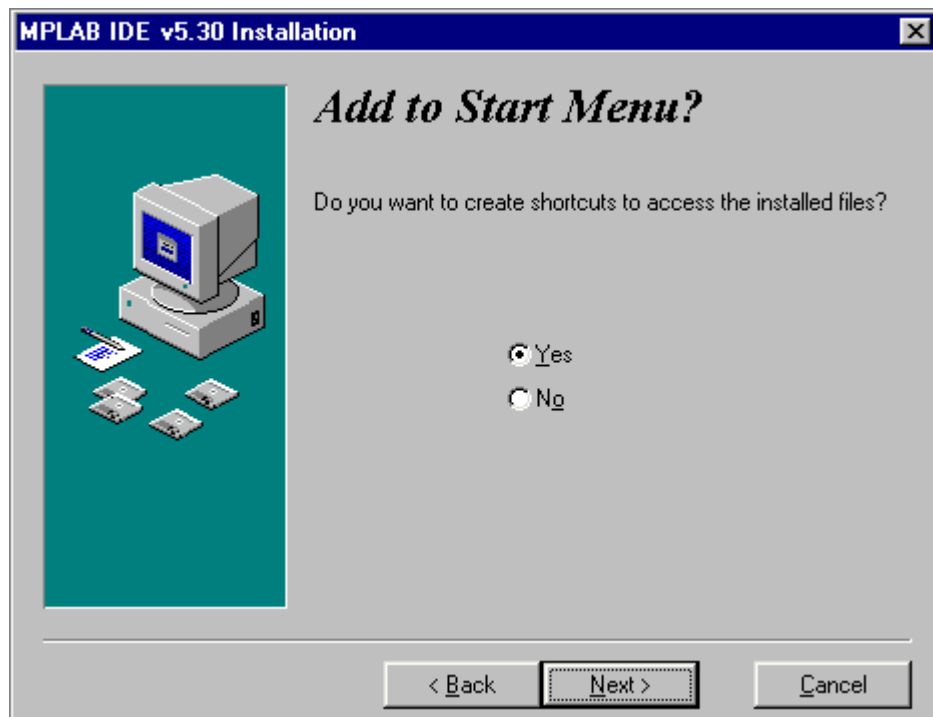
Selection of the directory for the MPLAB installation

The next option is necessary for the users who already had some previous MPLAB version (different from one that is being installed). Its purpose is to save all the file copies that are subject to change upon the transition to an updated version. In our case the selection of NO assumes that the installation in course is the first one.



The option necessary to the users who install the new version of MPLAB over some already existing installation

The start menu is the set of the pointers onto the programs opened by the click onto the START button in the lower left corner of the screen. It is necessary to leave this option exactly as it is offered, since MPLAB is going to be started from here.



Adding MPLAB into the START menu

Location mentioned next is related to the part of MPLAB which will not be explained here as it is insignificant for users. By selecting an apposite directory, MPLAB will keep all the files in connection with the linker in that directory.



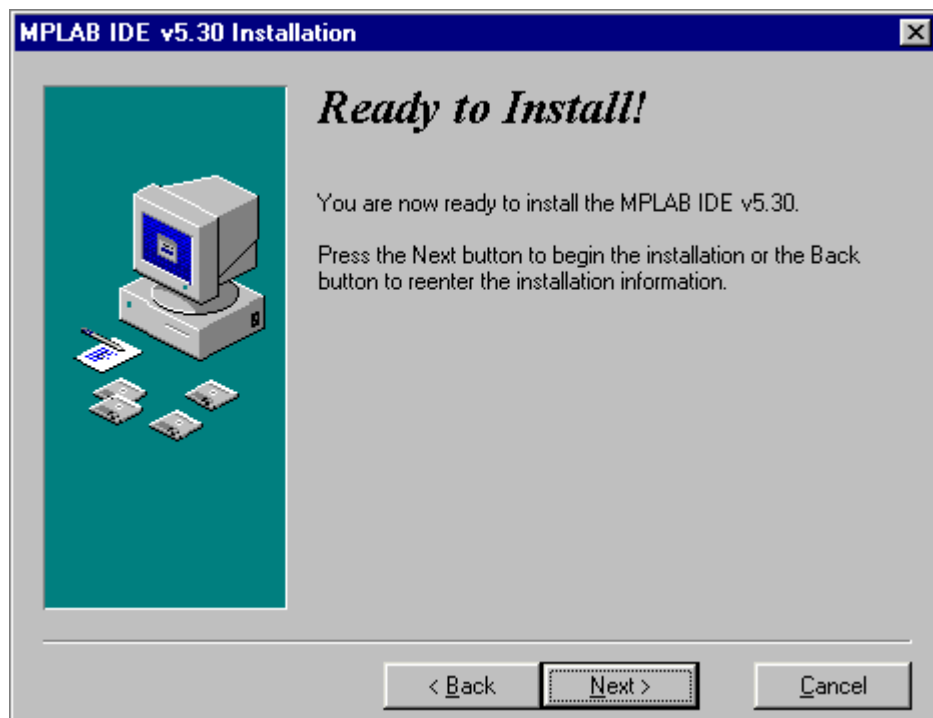
Selection of the directory for the linker files

Every Windows program has the system files, usually stored in the same directory as the Windows itself. After numerous installations, the Windows directory has a tendency of becoming too big and encumbered. Therefore, some of the programs permit their system files to be kept in the same directory as the program itself. MPLAB is one such program so that the option below should be selected.



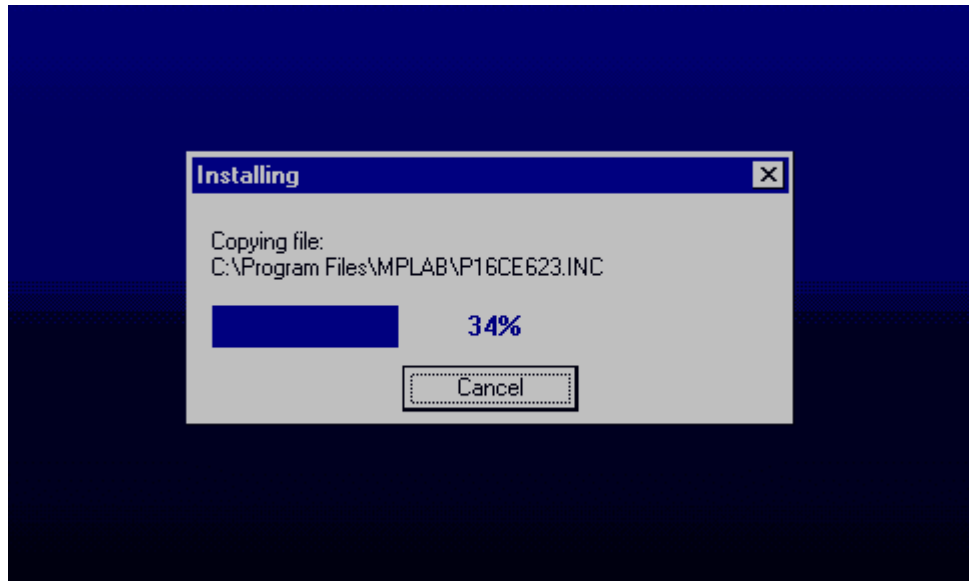
Selection of the system files directory

Following all steps up to now after pressing the button 'Next' the installation is under way



The screen exactly before the installation

The installation itself is brief and the course of the copying can be monitored on the small screen in the right corner.



The installation in course

When the installation is terminated, two dialog boxes are present on the screen – one for the last information concerning corrections and the version of the program, the other greeting one. If the text files (Readme.txt) are opened they should be closed.



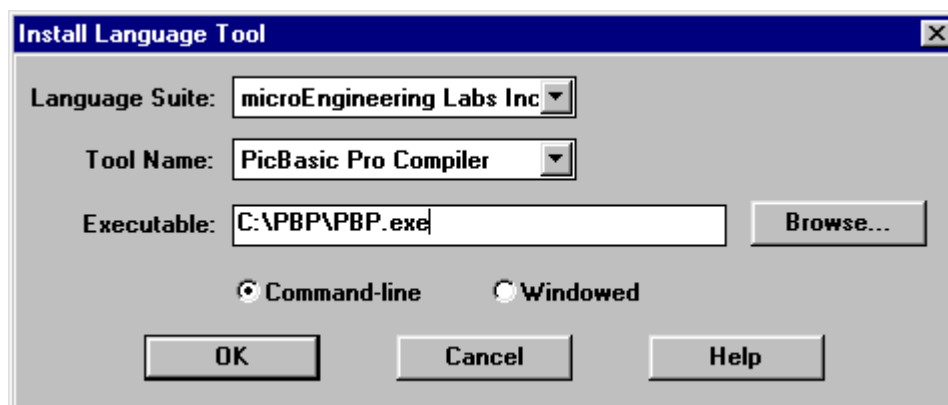
The last information concerning version and the corrections on the program

By clicking on the *Finish* button the installation of the program is thereby terminated.

A.2 Connection of PIC BASIC and MPLAB

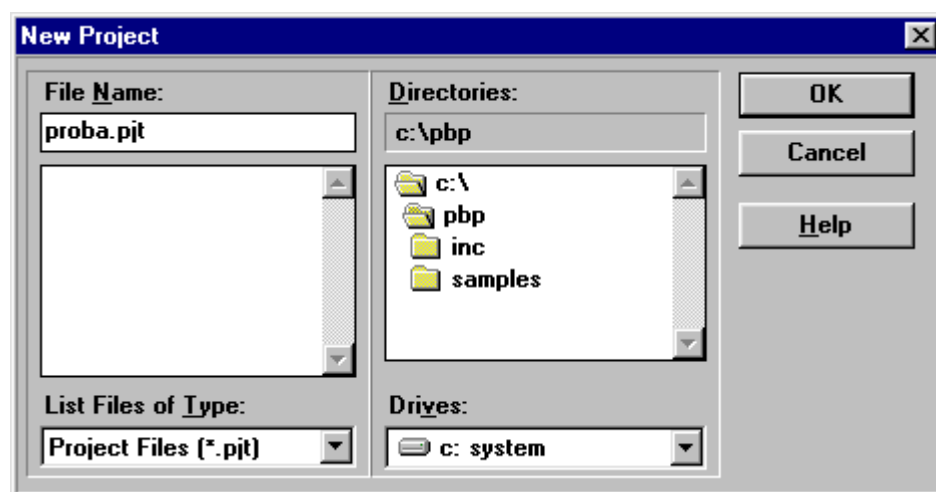
To make work as easy as possible to those who already got used to the assembler's compiler and MPLAB, *Microchip* has left the option of using, besides its proper, the compilers of the other manufacturers in its MPLAB development tool. Before starting to write a program, it is necessary to undertake some adjustments. Let's assume that, for example MPLAB is installed in directory: C:\ *Program Files* \ MPLAB and PIC BASIC Pro compiler in C:\ PBP.

You just start the MPLAB and choose *Install Language Tool* from the *Project* menu. The dialog box where the corresponding options is to be set, the manufacturer first, (whereby directly in the next option comes the list of compilers by the same manufacturer) and accordingly the compiler itself – in our case *Pic Basic Pro Compiler*- and exactly as the one on the picture bellow will appear then. At the end on should click at the option “browse” and find PBP.EXE file on the disk (in this case C: PBP\). By clicking on OK the basic settings are completed.



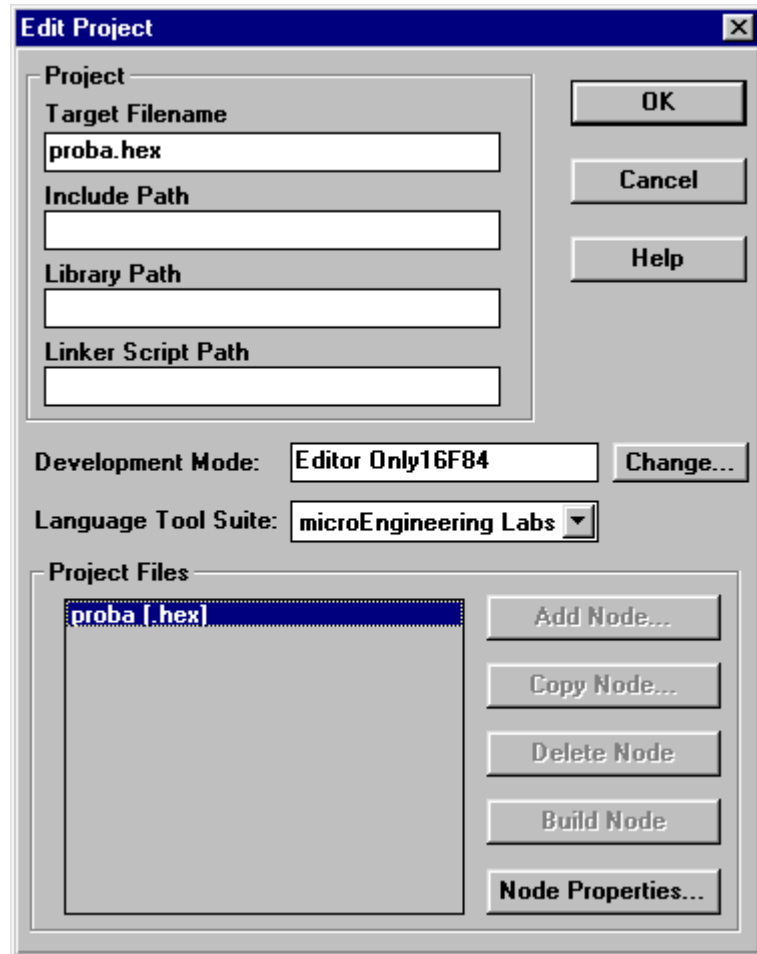
Start MPLAB and choose the *Install Language Tool* from the *Project* menu.

Next step is the creation of the project that is done in a standard way by selecting *New Project* from the *Project* menu and by assigning the project name e.g. “probe.pjt”. A special care is to be given to the project storage location. The new project and all its components must be located in the same directory as PicBasic Pro! For this case, the project must be stored in C:/PBP.



Creating project by selecting *New Project* from *Project* menu and assigning the project name as, e.g. “probe.pjt”.

By clicking *OK* the new window *Edit Project* appears. In *Language Tool* “microEngineering Labs” is to be selected (answer the incoming question with *OK*). It is, hence, necessary to click on ‘probe [.hex]’ in the lower part of the window whereby the option *Node Properties* is activated.

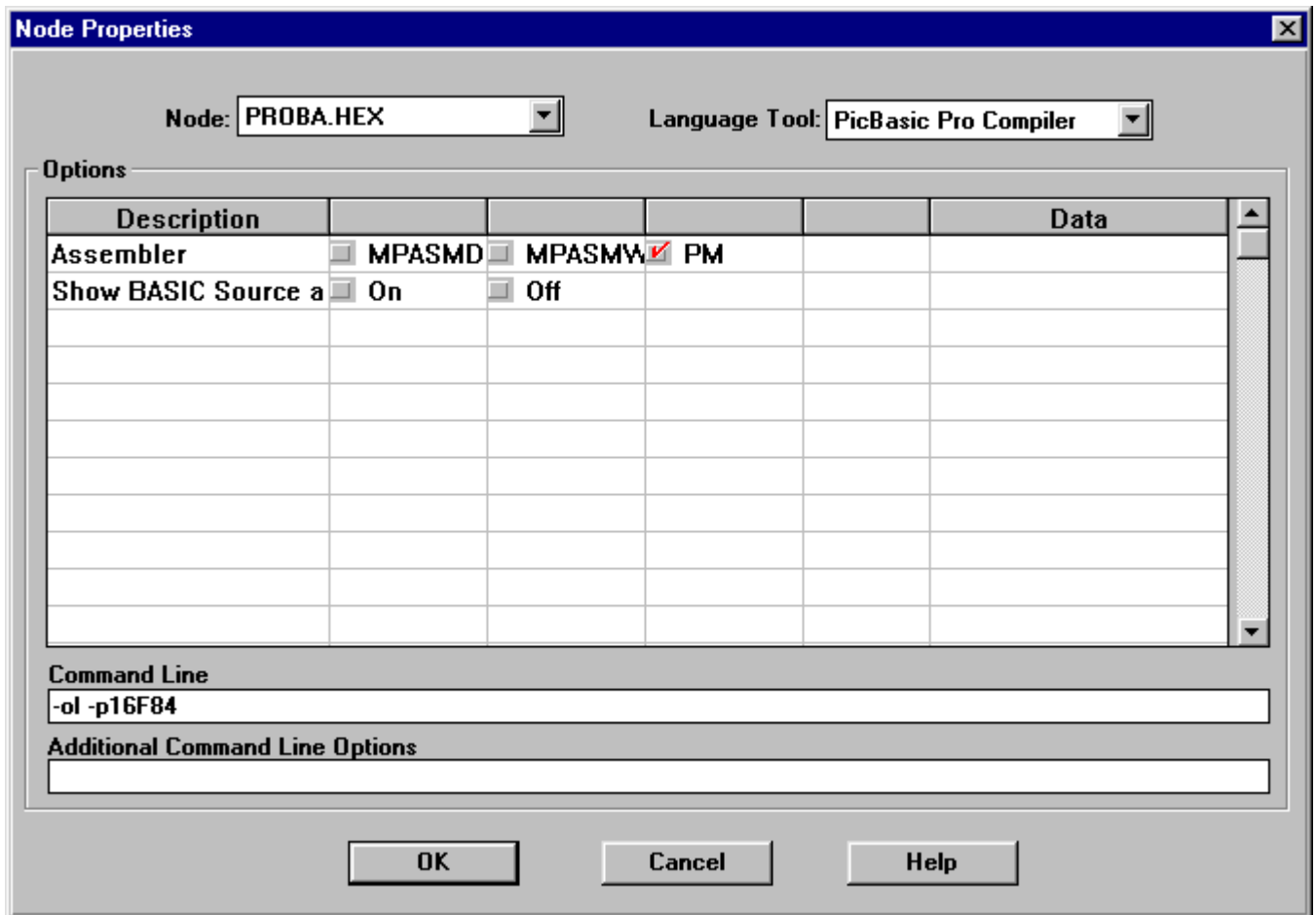


The New window Edit Project for the definition of the manufacturer. Choose “microEngineering Labs”

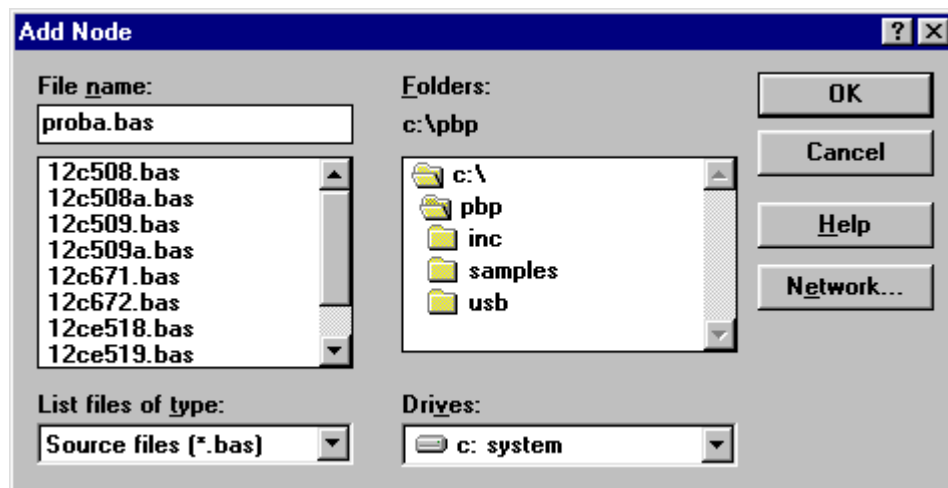
The purpose of this window is to set the microcontroller for which the program is written.

By clicking *Change* button, the new window for choosing the available microcontrollers appears. As an option, *Editor only* is to be left in the absence of any available Microchip’s tools (this option states the use of MPLAB as a shell for PIC Basic compiler).

By clicking *Node Properties* the window shown on the picture below appears. Choose “PM” version in the assembler selection. Clicking the *OK* returns us to the previous window.



The *Add Node* button is active now, and through it the name to the file with basic program is assigned. It is in our case, 'proba.bas'. It is to take notice that the present action is only assigning name of the file into the project. Its actual creation is done in next step.



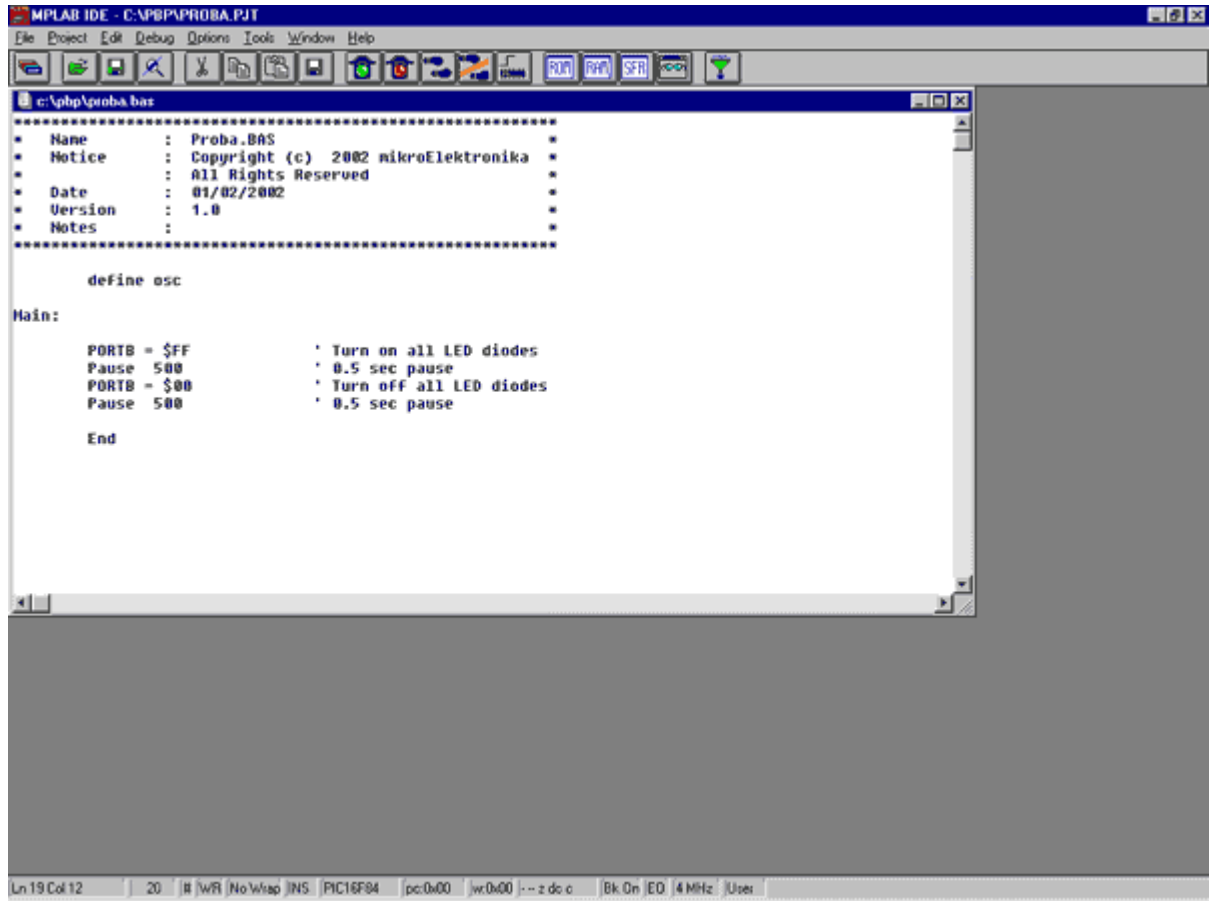
Window for naming the program in writing. Opening of the file is done in next step.

So far we defined microcontroller and the programming language. It still remains to open the file, write the code and register it under the name given in previous step. (proba.bas).

By clicking *File-> New* the window in which the basic program will be written appears.

Before we start the program writing, file must be registered with the command *File-> Save as*, file

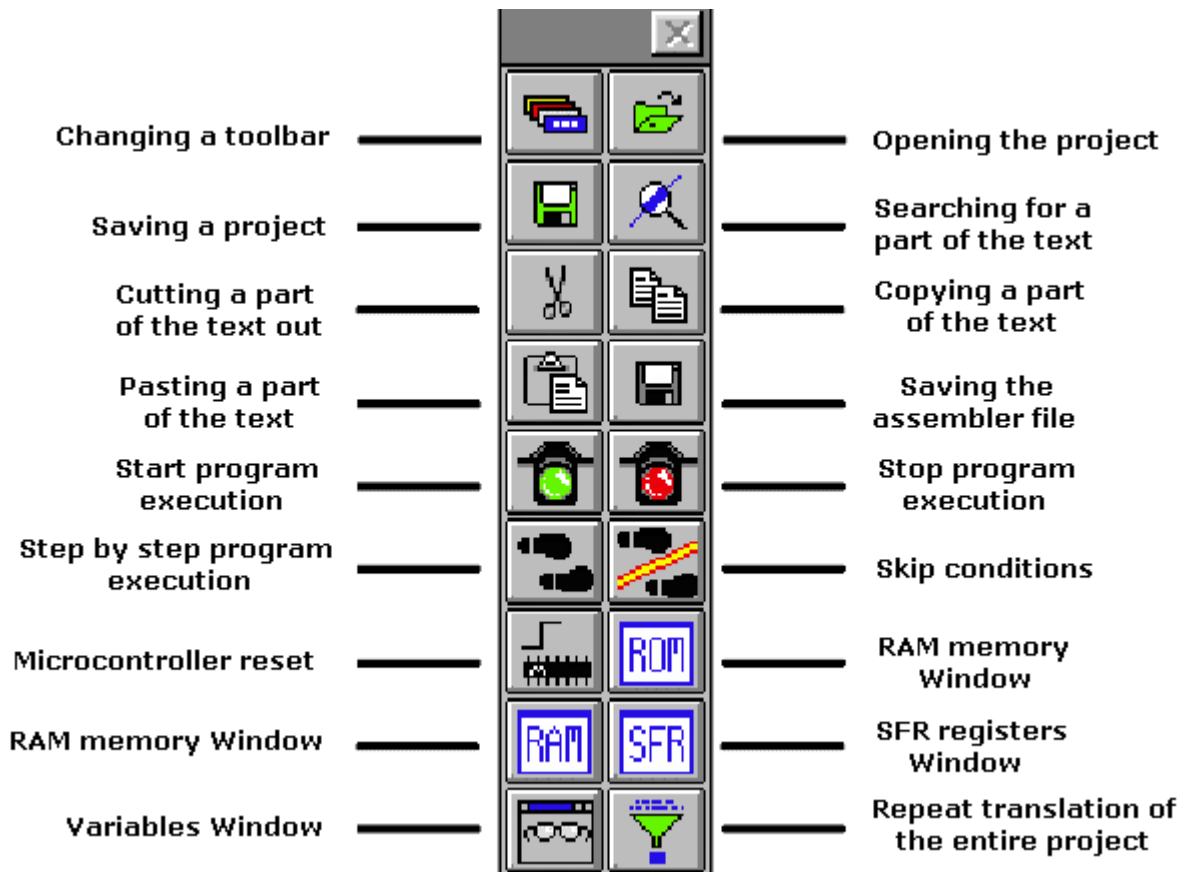
name being obviously “proba.bas”. The code writing can start now. The program here serving as an example is a very simple one and its only function is to make the diode on a port B twinkle.



The window for writing Basic program



















Upon finishing the code writing, the click on *PROJECT-> Build All* is performing the compilation of the program. Unless there have been some errors, the obtained file is C:/PBP/probe.hex readable into the microcontroller.

A.3 Toolbar



Since MPLAB is composed of several separate parts, each of them possesses its own toolbar. However, there exists a toolbar being a sort of a combination of all the others, which may be considered as a common one. This toolbar is sufficient for our needs so it will be explained in details. On the picture below this toolbar is given with the brief explanations of the icons. Out of the limited format of this book, the basic toolbar is displayed as the free one and in a standard position is always below the menu, displaced horizontally along the entire screen.

If, for whatever reason, currently used toolbar does not respond, upon clicking this icon the next toolbar becomes available. The change goes into circle so that upon the 4th click, the same toolbar is obtained again.

	If the current toolbar for some reason does not respond to a click on this icon, the next One appears. Changeover is repeated so that on the fourth click we will get the same toolbar again.
	Icon for opening a project. Project opened in this way contains all screen adjustments and adjustment of all elements which are crucial to the current project.
	Icon for saving a project. Saved project will keep all window adjustments and all Parameter adjustments. When we read in a program again, everything will return to the screen as when the project was closed.
	Searching for a part of the program, or words is operation we need when searching Through bigger assembler or other programs. By using it, we can find quickly a part of the program, label, macro, etc.
	Cutting a part of the text out. This one and the following three icons are standard in all Programs that deal with processing textual files. Since each program is actually a common text file, those operations are useful.
	Copying a part of the text. There is a difference between this one and the previous icon. With cut operation, when you cut a part of the text out, it disappears from the screen (and from a program) and is copied afterwards. But with copy operation, text is copied but not cut out, and it remains on the screen.
	When a part of the text is copied, it is moved into a part of the memory which serves For transferring data in Windows operational system. Later, by clicking on this icon it can be 'pasted' in the text where the cursor is.
	Saving a program (assembler file).
	Start program execution in full speed. It is recognized by appearance of a yellow status line. With this kind of program execution, simulator executes a program in full speed until it is interrupted by clicking on the red traffic light icon.
	Stop program execution in full speed. After clicking on this icon, status line becomes gray again, and program execution can continue step by step.
	Step by step program execution. By clicking on this icon, we begin executing an instruction from the next program line in relation to the current one.
	Skip requirements. Since simulator is still a software simulation of real work, it is possible to simply skip over some program requirements. This is especially handy with instructions which are waiting for some requirement following which program can proceed further. That part of the program which follows a requirement is the part that's interesting to a programmer.
	Resetting a microcontroller. By clicking on this icon, program counter is positioned at the beginning of a program and simulation can start.
	By clicking on this icon we get a window with a program, but this time as program memory where we can see which instruction is found at which address.
	With the help of this icon we get a window with the contents of RAM memory of a microcontroller.
	By clicking on this icon, window with SFR register appears. Since SFR registers are used in every program, it is recommended that in simulator this window is always active.
	If a program contains variables whose values we need to keep track of (ex. counter), a Window needs to be added for each of them, which is done by using this icon.
	When certain errors in a program are noticed during simulation process, program has to be corrected. Since simulator uses HEX file as its input, so we need to translate a program again so that all changes would be transferred to a simulator. By clicking on this icon, entire project is translated again, and we get the newest version of HEX file for the simulator.

Appendix B

MicroCODE STUDIO

[Introduction](#)

- [B.1 Installation of the PIC Basic Pro compiler](#)
- [B.2 Installation of a MicroCODE studio](#)
- [B.3 Connecting MicroCODE Studio and PBP compiler](#)
- [B.4 Connecting MicroCODE Studio and the programmer](#)
- [B.5 Code writing and compilation in MicroCODE studio](#)

Introduction

Although the code writing can be done with the simplest editor and compiled in command line (those who had programmed in DOS probably remember well those acrobatics) using special “editors” appropriate for programming language is far better.

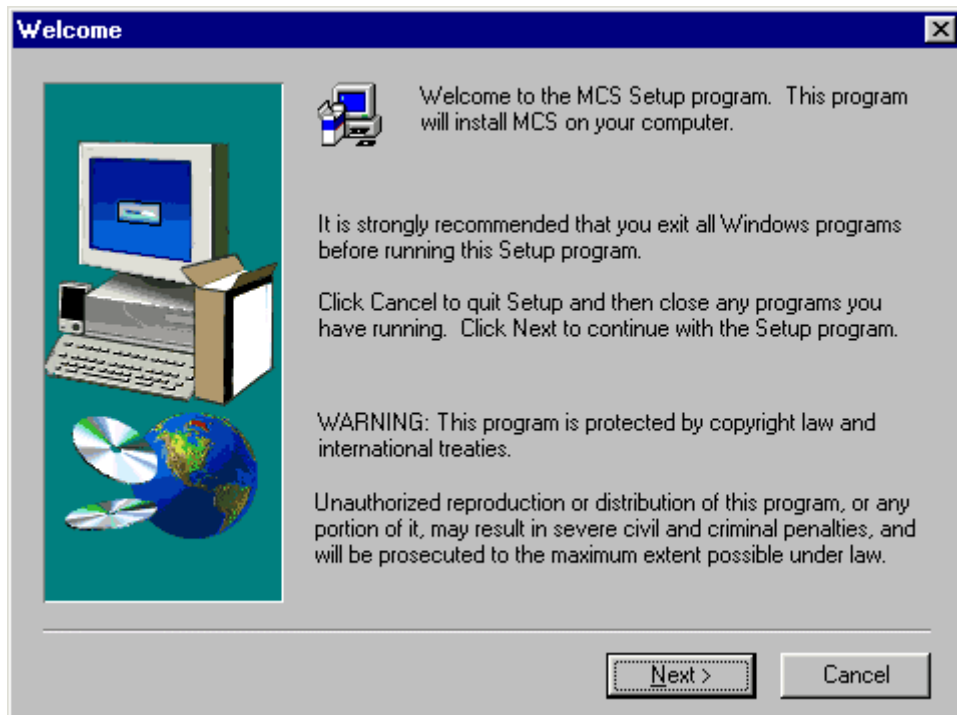
Such specialized editors are called “Integrated Development Environments” - *IDE*. Using them makes code writing easier as the programmer is able to supervise which variables, labels or similar program elements have already been used. At the same time, they make command words bold and even write them in another color rendering thereby program more intelligible. The option for automatic call up of the programmer is also available together with many other facilities. Simply put, having those facilities without using them is like climbing on foot to the 13th floor of a building with elevator.

B.1 Installation of the PIC Basic Pro compiler

The first thing to be done is to create a new directory into which the compiler will be stored. Let it be the directory C:/PBP. Then follows the copying of data file PBP240.EXE into that directory and its unpacking (compiler enters in the form of unpacking archive)? by double-clicking it. Unless the compiler is unpacked it is enough to copy it into the desired directory.

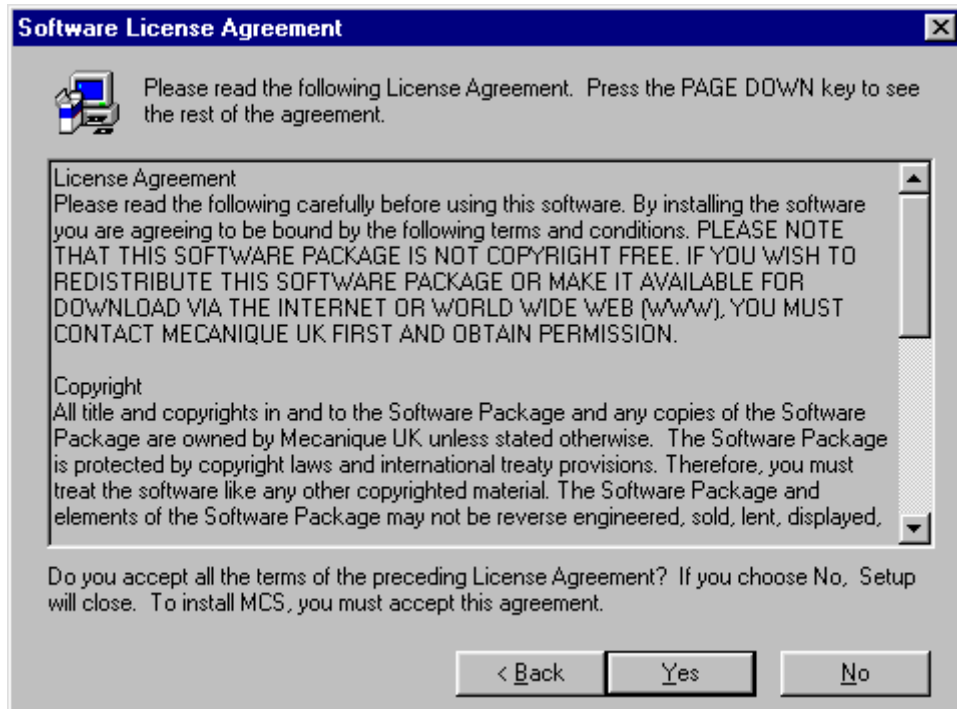
B.2 Installation of a MicroCODE studio

Installation of the editor starts by double-clicking on MCSTUDIO. Afterwards, the standard setup process is started where the computer location for the editor’s installation can be chosen. The setup process starts with the usual warning to close all other active windows. By clicking on button *Next*, the setup continues.

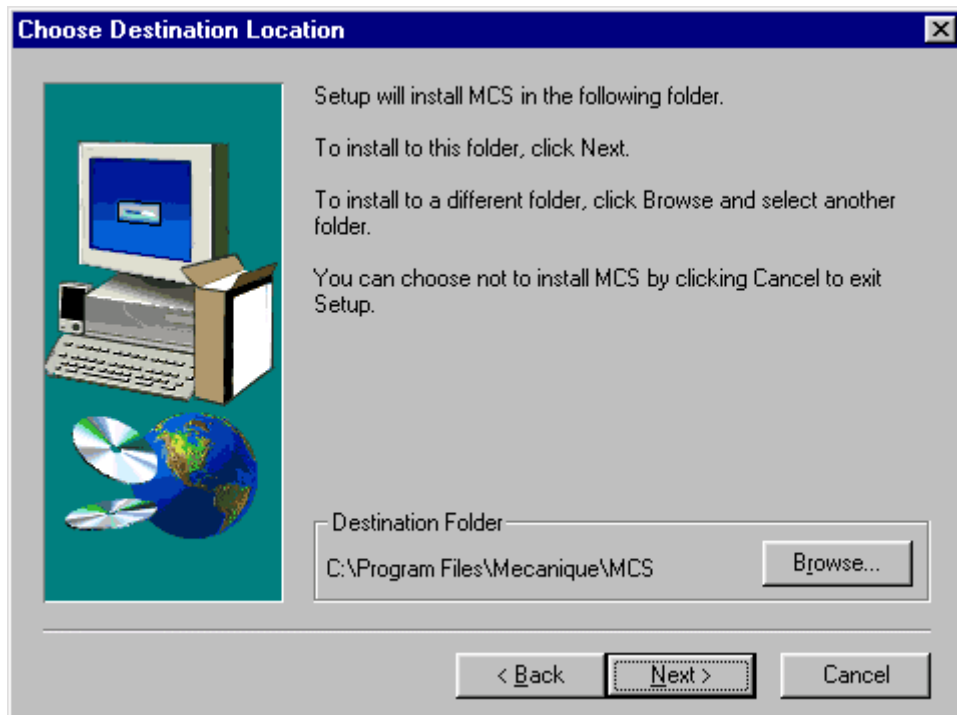


The first window after the installation starts. It is necessary to click on button Next

Next question is whether you accept the license and copyright rules or not. By accepting these rules by clicking on the Yes button, the installation goes forward. The next image corresponds to that phase of the installation.



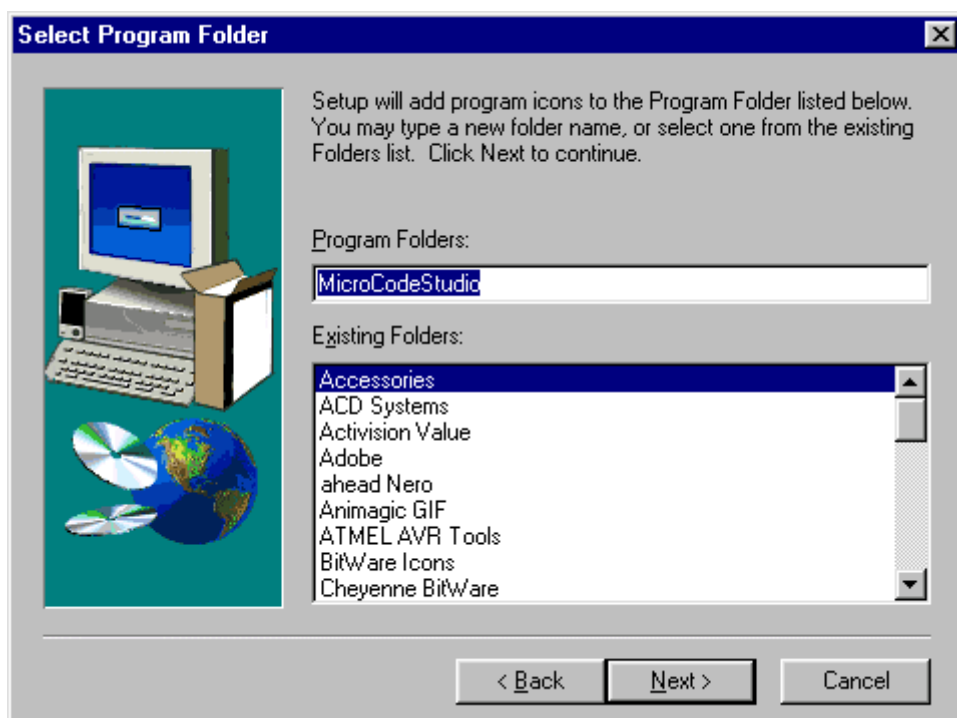
The directory for editor location is the next question. In case of failed statement of the directory, the installation is to be effectuated in <C:\ProgramFiles\Mecanique>.



The choice of an installation directory. The best choice is to leave the option by default. It is necessary to click on OK button in order to proceed

The name and address of directory is without any special meaning for further programming. The real issue is the available memory space on the hard disk or on the need for keeping all items associated with a single program in the same directory.

The next question refers to the name of programming group. The name already offered corresponds to the program name so it should be left as such.



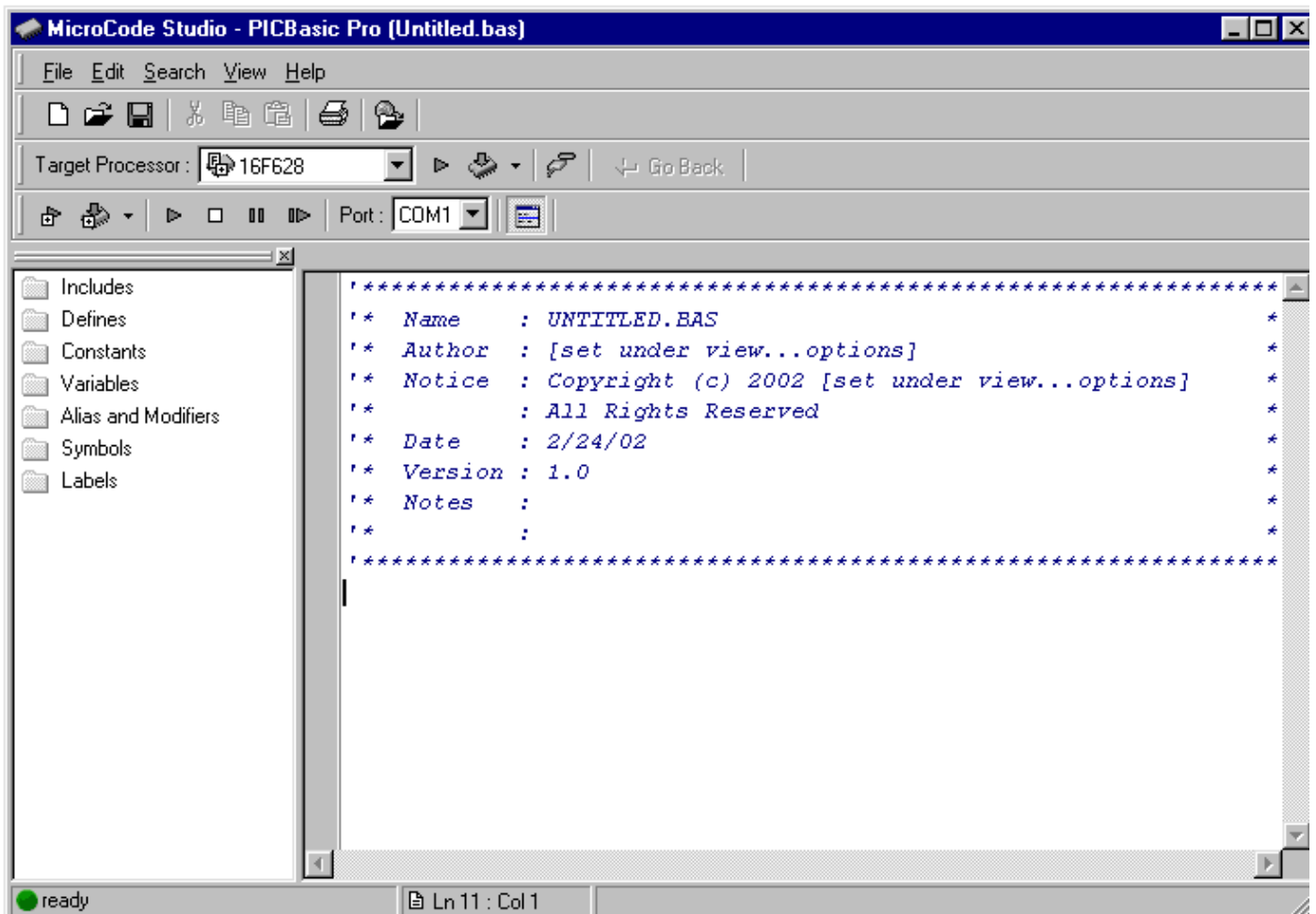
The program group is to be named MicroCodeStudio. Clicking on Next, the installation goes on

Finally, the window appears confirming the successfully performed installation.

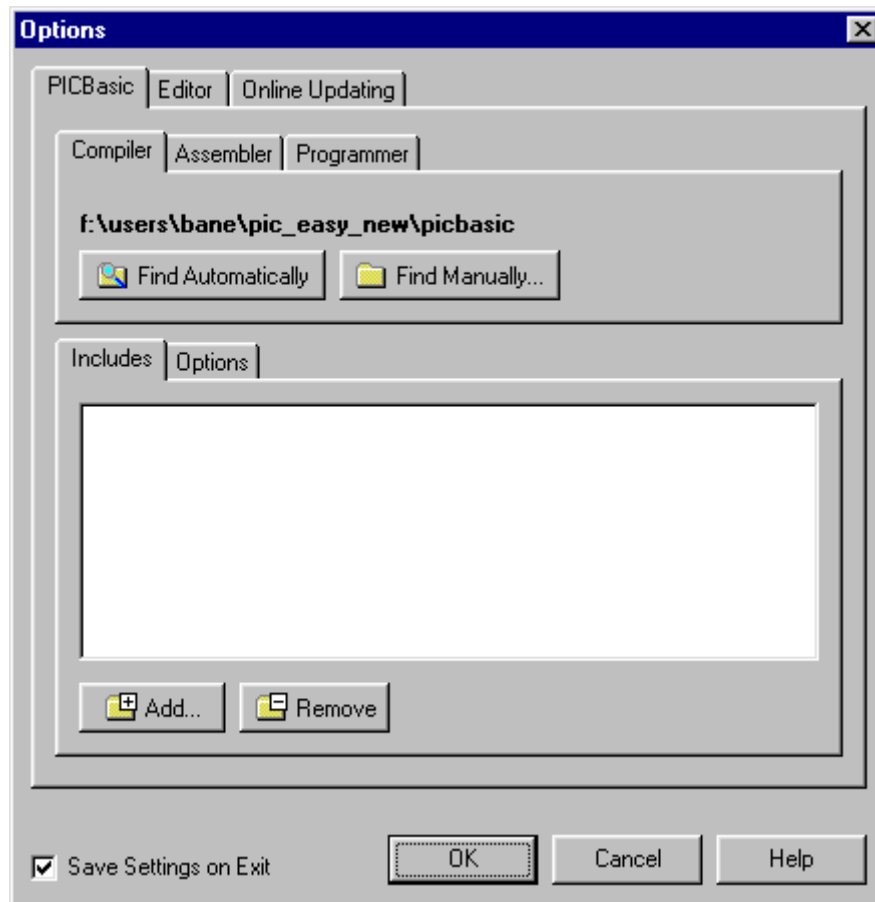


B.3 Connecting MicroCODE Studio and PBP compiler

Clicking on *Start-Programs-MicroCode Studio* starts up the just installed MicroCode Studio and the window from the picture bellow will appear.

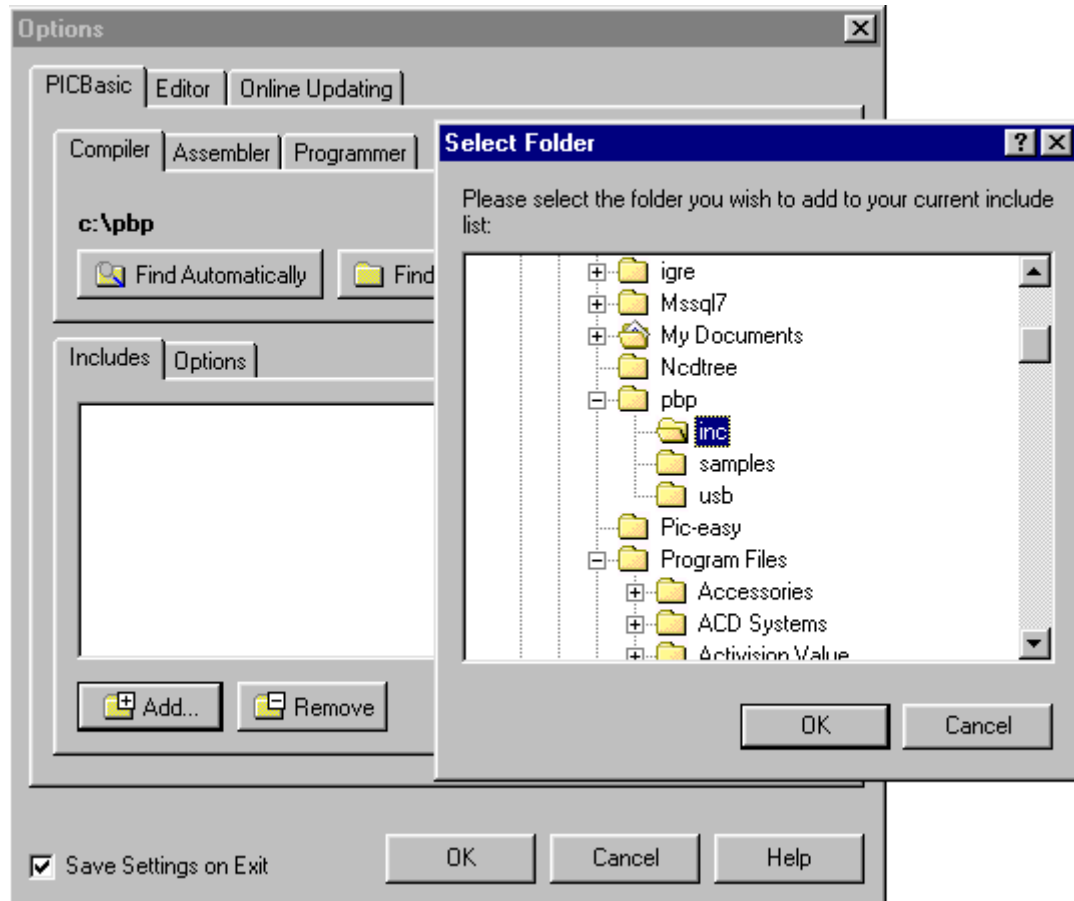


To connect MicroCode Studio and PBP compiler a new window is to be opened. It's done by clicking on the *Options* from the *View* menu. If the compiler is already copied into a hard disk directory clicking on the *Find Automatically* button whereupon will the program itself search for the directory with compiler through the hard disk. When the program finds the compiler, above the button the path "C:\PBP" will appear above the button *Find Automatically*.

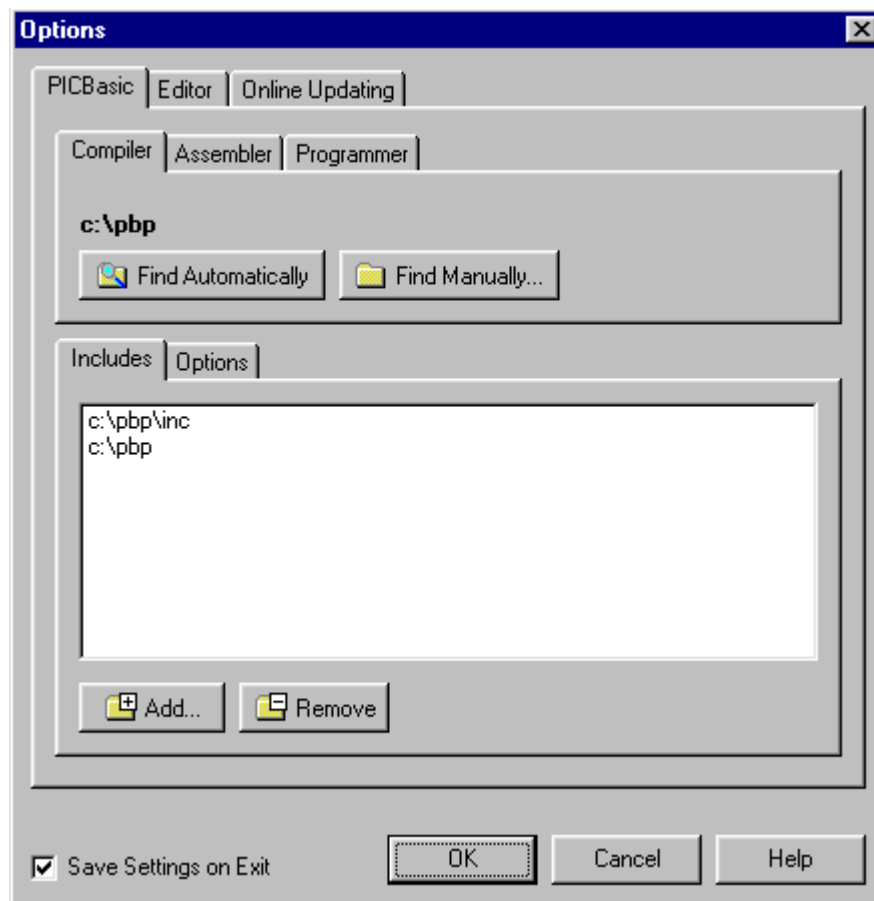


Connecting MicroCode studio and PBP compiler. If the PBP compiler is already copied into a directory on a hard disk, it is enough to click on the Find Automatically button and the program will find it on its own

Beside the path to the compiler, it is still necessary to define the path to the include data file. By clicking on *Add* the paths C:/PBP and C:/PBP/inc are added within *Includes*.



Include data files are necessary for successful compilation of the program. Clicking the Add, the new window appears with the inc directory into which the PBP compiler is copied



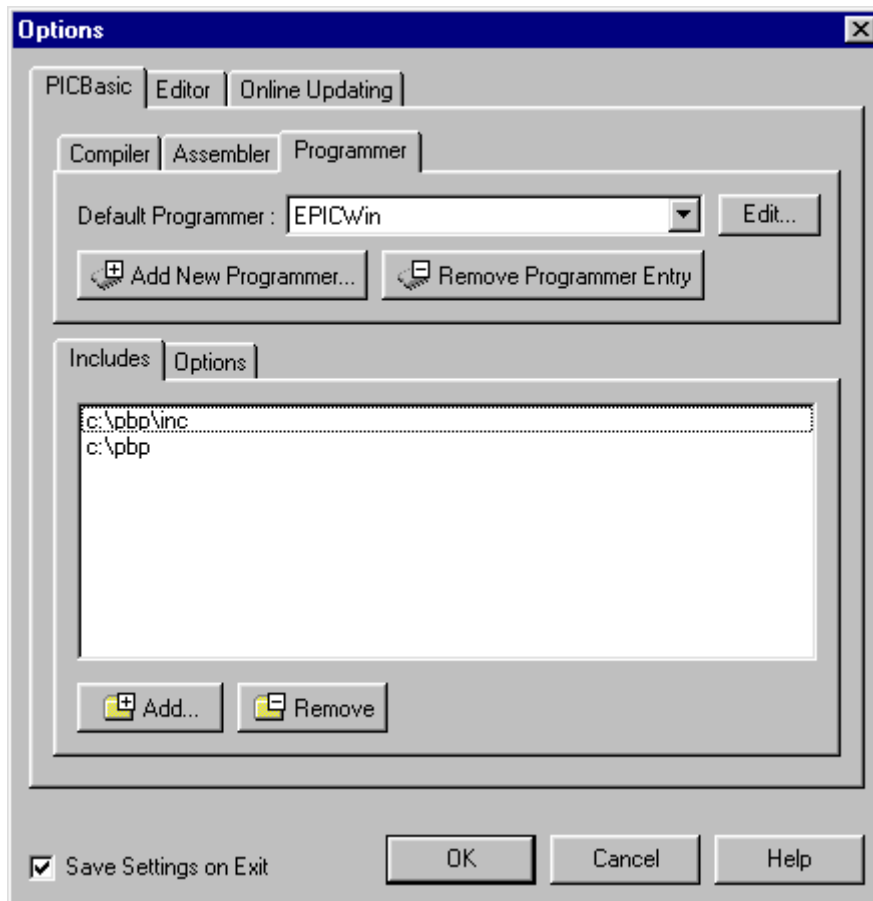
Options window after setting the path to the compiler and include data files. Notice that there are include data files in the very C:\PBP directory so that their path should be specified as well

This step finishes the setting part referring to the compiler. MicroCode studio is now ready for program reading and compiling.

B.4 Connecting MicroCODE Studio and the programmer

The installation of the programmer that *MicroCode* will call upon successfully accomplished program compilation is to be undertaken only if the user possesses some development environment or some of the programmers that will read in the compiled program into the microcontroller. In lack of any of these tools this part of MicroCode studio setting is to be omitted.

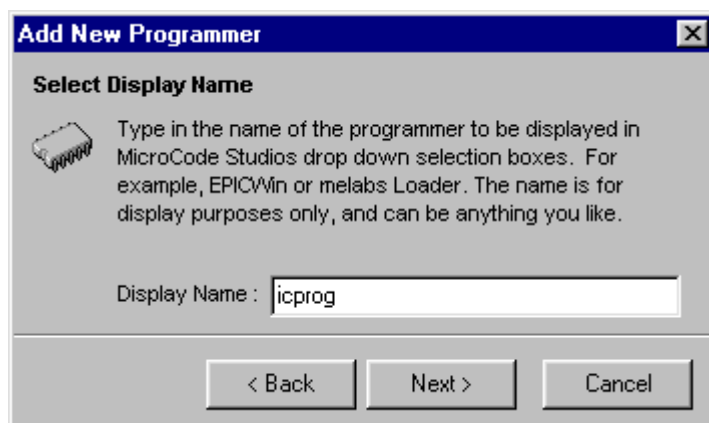
The setting of the programmer starts by clicking on *Programmers* whereupon two distinct options appear, one for adding of programmer into the list and another for their removal. The programmer that is to be used here ranks as the simplest economic programmers of PIC microcontrollers that are available at the moment. The name of this programmer is *ICprog* and it uses the serial pin of the computers port in order to communicate with the microcontroller (more details can be found in the special appendix contained in this book).



By clicking Programmers the part for setting the programmer appears

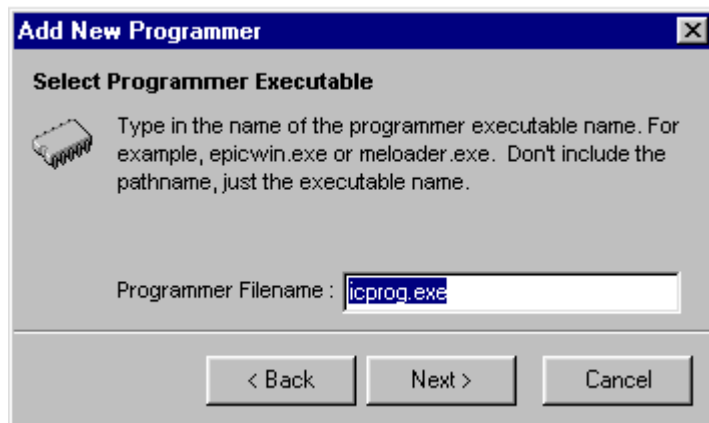
Before installing the programmer, it has to be copied in a directory on the hard disc, e.g. "C:\Programmer". Clicking the "Add new Programmer...", the brief procedure of selecting the path to programmer begins.

The first step is writing the name of the programmer or any abbreviation that could bear resemblance to it. As Icprog programmer is used it is logical to name it "ICprog".



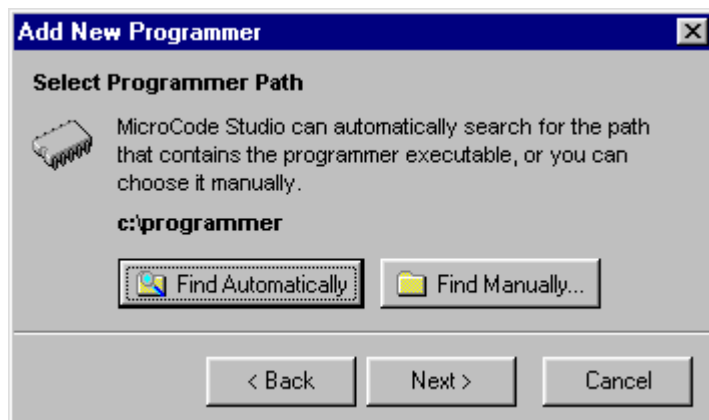
In this option the name of the programmer is to be written. It can well be any of the names bearing resemblance to the programmer we wish to install

The next step is the writing of the exact name of the programmer. It is very important not to make any mistake; otherwise the program will not be able to locate it on a hard disk.



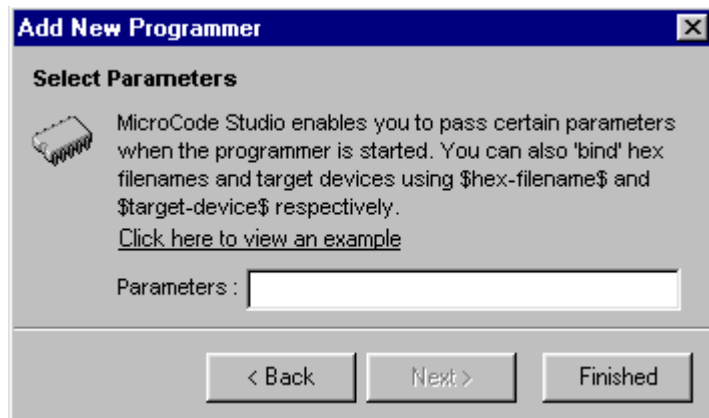
In this option, the exact name of the executive data file of the programmer is to be indicated. In this case it's icprog.exe

Finally, by clicking on *Find Automatically*, the program then finds on its own the path towards the programmer.



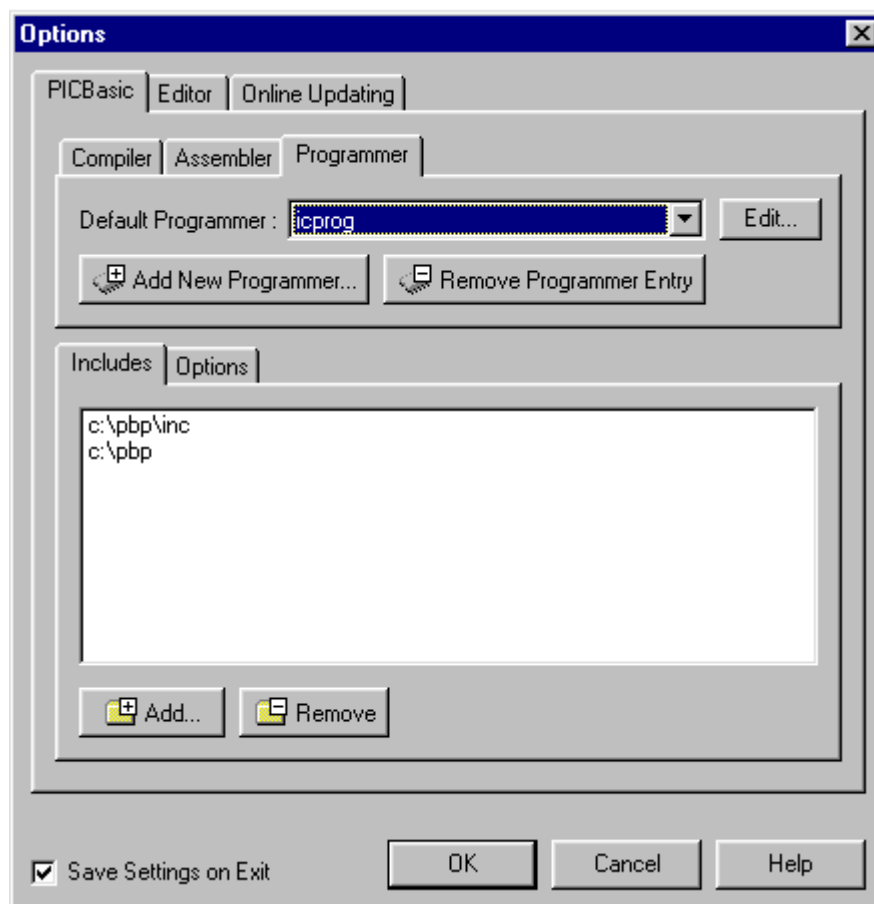
By clicking on Find Automatically the program finds the path to the programmer on its own

Option to define additional parameters is next. Nevertheless, it is to be omitted due to the fact that it will be used in a later phase of the operation when the longer programs are written and the program name is not changed very often. Clicking on *Finished* overrides this option.



The option to define the additional parameters of the programmer is not to be used here; therefore it is to be omitted by clicking on Finished

The window *Option* out of the *View* menu with the set parameters for the compiler and the programmer now looks like exactly as on the image bellow. Thereby all relevant settings of the MicroCode Studio are finished.

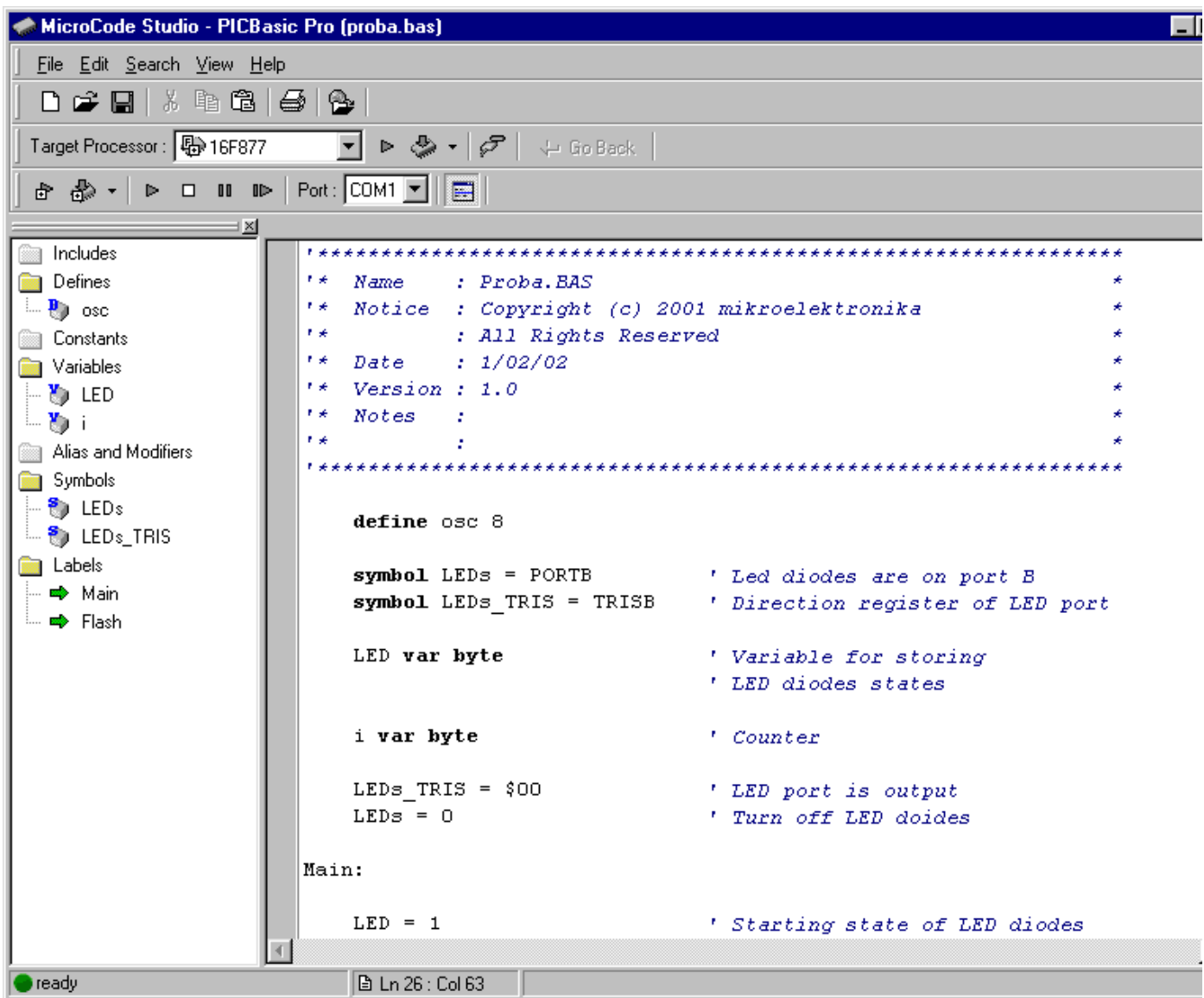


Window Option with all the parameters for the compiler and the programmer set

Besides the setting of the compiler and the programmer, there are somewhat less important settings as that of an editor. Since those parameters are already well set we will not take them into consideration now.

B.5 Code writing and compilation in MicroCODE studio

The MicroCode studio looks like most of the Windows programs. Above the working area there are menu lines, toolbars and the line connected to the compilation and reading of a program into the microcontroller.



The menu line contains all standard submenus as File, Edit, Search, View and Help.

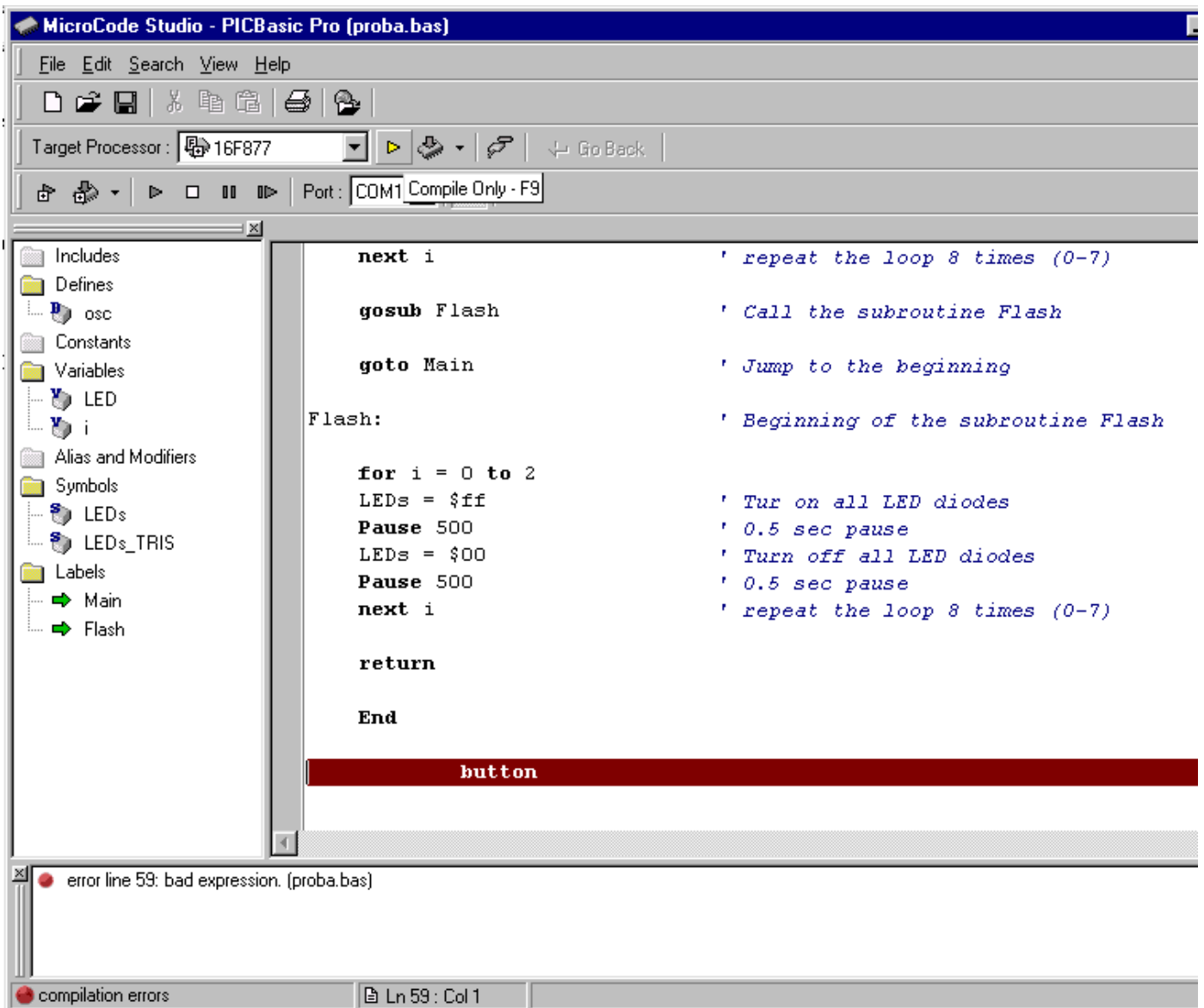
The toolbar contains but a few basic icons and their purpose we will not explain in details.

What separates the MicroCode studio from the other development environments is its simplicity and legibility. Its most important part is located in the left part by the name *Code Explorer*. When necessary, that part of the window can be shut down by clicking on *View – Code Explorer...*

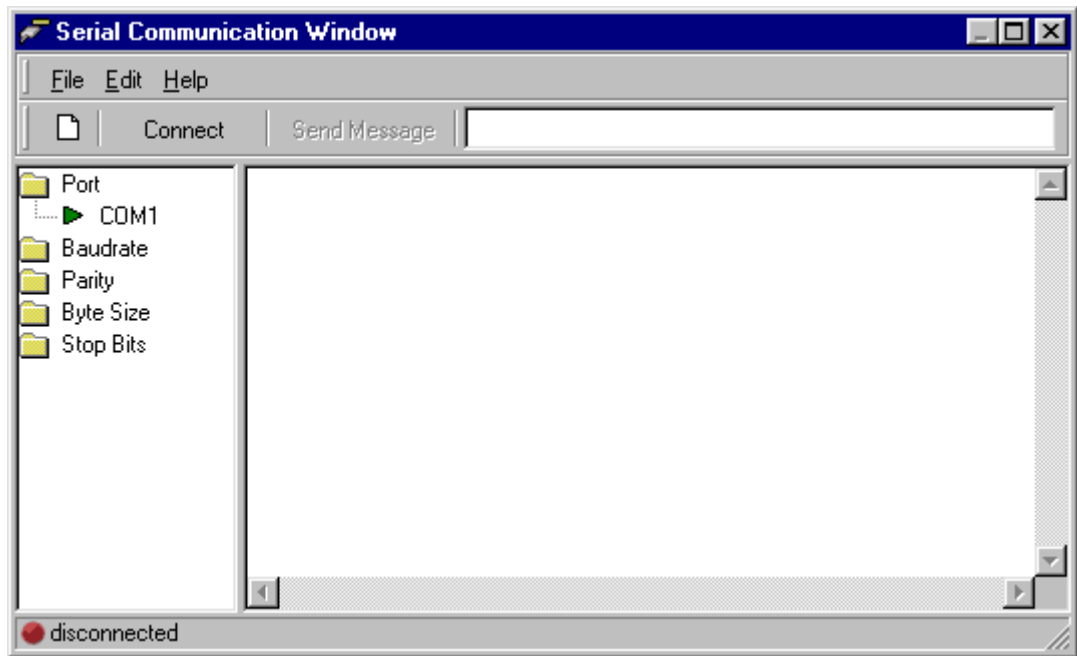
although it is recommended to leave it as it is for it contributes to the better legibility and organization of the program. The code writing is done in the right part of the window. The process of code writing itself is largely facilitated by thickening of the commands, and by the excellent solution for the complicated commands with the greater number of parameters as “button” command is. Namely, after writing of this command and the first empty (blank) character, the yellow frame with all parameters of the respective command appear.

Upon having written the code, by clicking on icon *Compile Only* (in triangular shape on the right side) the compilation of program starts. If an error occurs, it's reported in a special part at the bottom of the window. By clicking on Error, the cursor is positioned exactly at the row in which the error occurred. After correction, the program is compiled as long as the compilation process becomes successful.

If the programmer is already configured, then the icon right next to the *Compile Only* can be used instead, which will, upon a successfully accomplished compilation, call the programmer.



Clicking on the icon in the port form, the special window for examining the serial connection with the microcontroller opens. The *Serial communication window* serves for the serial communication between PC and the microcontroller. An additional option exists which enables the change of all the transfer parameters such as the port on which the microcontroller is attached, the transfer rate or the transfer format.



Option for examining the serial connection with the microcontroller