PIC Microcontrollers - Programming in BASIC

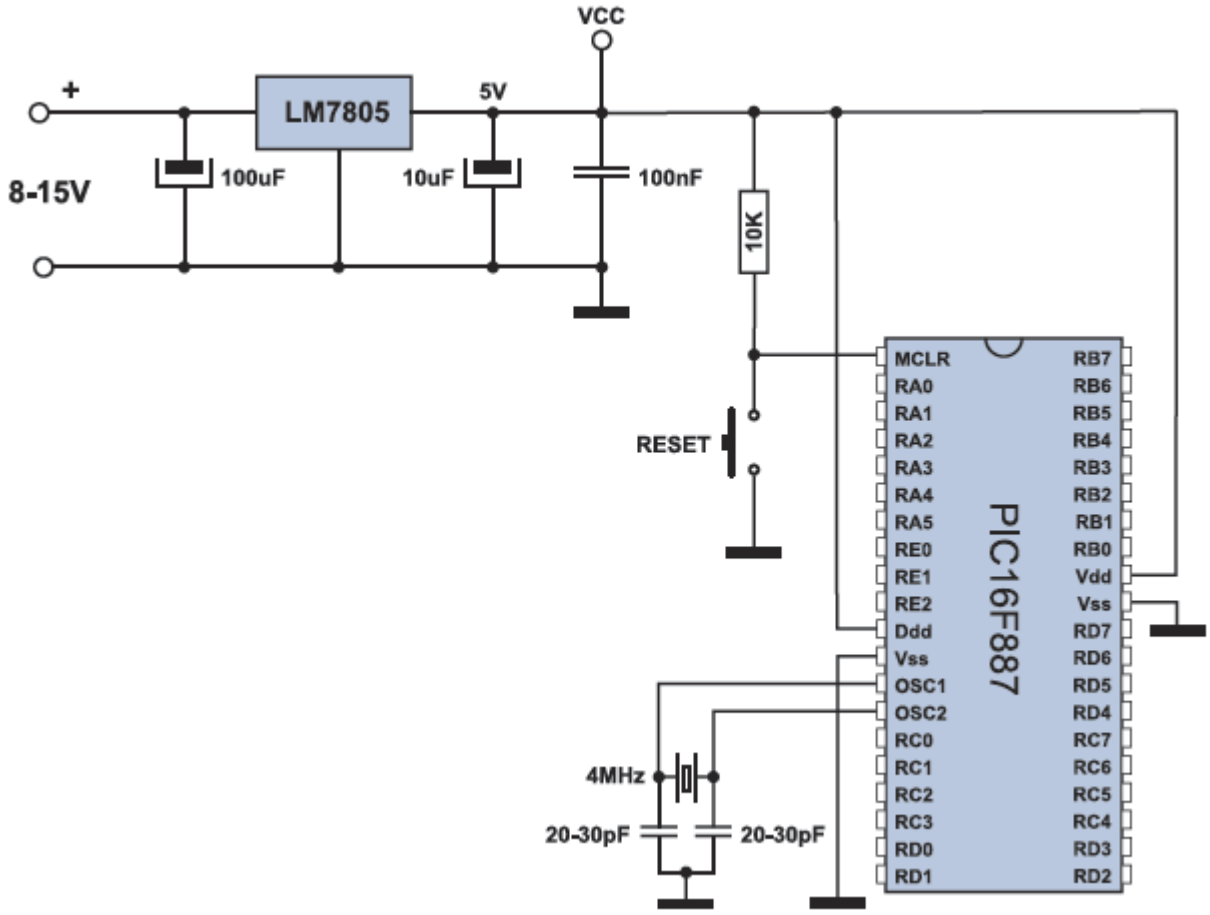
Chapter 4: Examples

The purpose of this chapter is to provide basic information that you need to know in order to be able to use microcontrollers successfully in practice. This chapter, therefore, doesn't contain any super interesting program or device schematic with amazing solutions. Instead, the following examples are better proof that program writing is neither a privilege nor a talent issue, but the ability of simple putting puzzle pieces together using directives. Design and development of devices mainly boil down to the 'test-correct-repeat' method. Of course, the more you are in it, the more complicated it gets since the puzzle pieces are put together by both children and first-class architects...

4.1 BASIC CONNECTING

In order to enable the microcontroller to operate properly it is necessary to provide:

- Power supply;
- Reset signal; and
- Clock signal.



As can be seen in figure above, it's about simple circuits. But it's not always the case. If the target device is used to control expensive machines or life-support devices, everything gets increasingly complicated. Anyway, this simple solution will do for the time being...

POWER SUPPLY

Even though the PIC16F887 can operate at different power supply voltages, why test 'Murphy's law'?! A 5V DC power supply voltage is the most suitable. The circuit, shown on the previous page, uses a cheap integrated three-terminal positive regulator LM7805 and provides a highquality voltage stability and quite enough current to enable the microcontroller and peripheral modules to operate normally (enough means 1A).

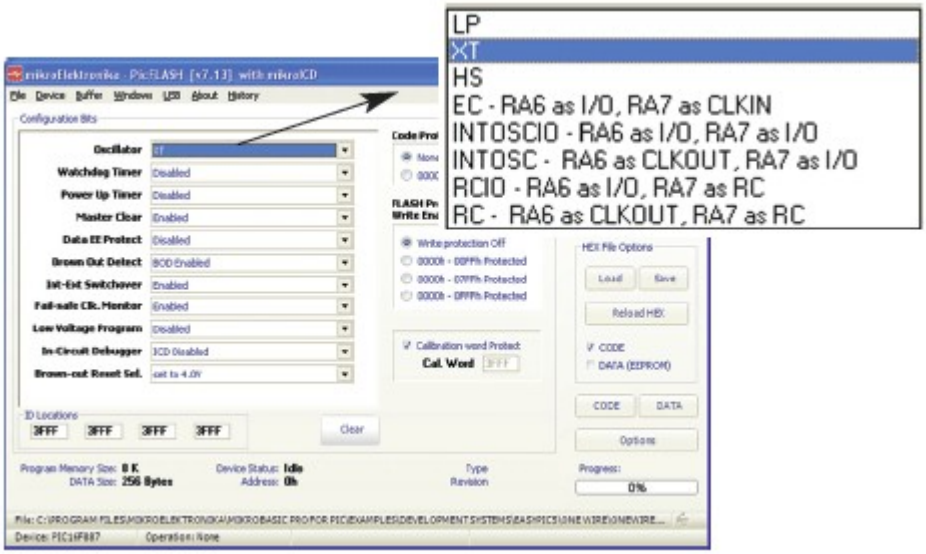
RESET SIGNAL

In order for the microcontroller to operate properly, a logic one (VCC) must be applied on the reset pin. A push button connecting the MCLR reset pin to GND is not necessary, but is almost always provided as it enables the microcontroller to recover fast if something goes wrong. By pressing this button, the MCLR pin is supplied with 0V, the microcontroller reset occurs and the program execution starts from the beginning. A 10K resistor is used to prevent shortening the 5V DC rail to earth from occuring when the RESET button is pressed.

CLOCK SIGNAL

Even though the microcontroller has a built-in oscillator, it cannot operate without external components which make its operation stable and determine its operating frequency. Depending on components in use and their operating frequencies, the oscillator can be run in four different modes:

- LP - Low Power Crystal;
- XT - Crystal / Resonator;
- HS - High speed Crystal / Resonator; and
- RC - Resistor / Capacitor.

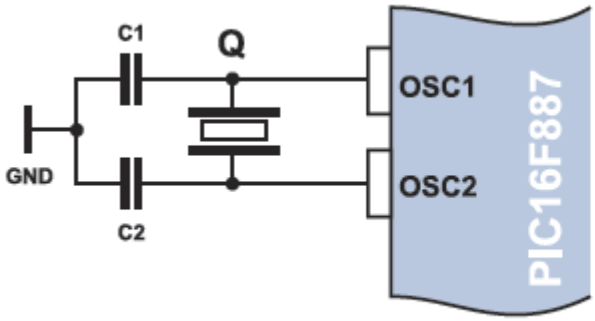


What's the point of using these modes? Owing to the fact that it is almost impossible to design an oscillator to operate stably over a wide frequency range, the microcontroller must be familiar with the type of quartz crystal connected so that it can adjust the operation of its clock oscillator to it. This is why all the programs used for programming microcontrollers contain an option for oscillator mode selection. See figure on the left.

Quartz Crystal

When the quartz crystal is used for frequency stabilization, the built-in oscillator operates at a precise frequency which is not affected by changes in temperature and power supply voltage. This frequency is usually labeled on the quartz crystal casing.

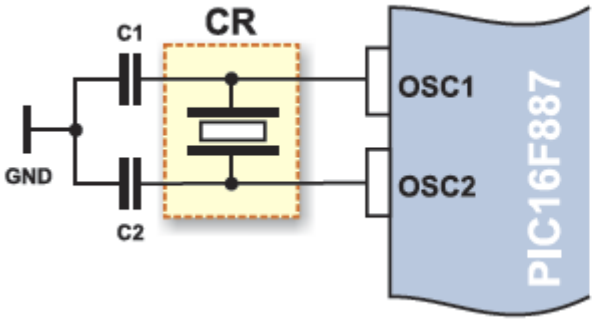
In addition to the crystal, capacitors C1 and C2 must also be connected as per schematic above. Their capacitance is not of great importance. Therefore, the values provided in the table next to the schematic should be considered as a recommendation, not as a strict rule.



Mode	Frequency	C1, C2
LP	32 KHz	33pF
	200 KHz	15pF
XT	200 KHz	47-68 pF
	1 MHz	15 pF
	4 MHz	15 pF
HS	4 MHz	15 pF
	8 MHz	15-33 pF
	20 MHz	15-33 pF

Ceramic Resonator

A ceramic resonator is cheaper, but very similar to quartz by its function and operating mode. This is why these two schematics, illustrating their connection to the microcontroller, are identical. However, compared to the quartz crystal, the capacitance of capacitors C1 and C2 is slightly different due to different electric features. Refer to table below.

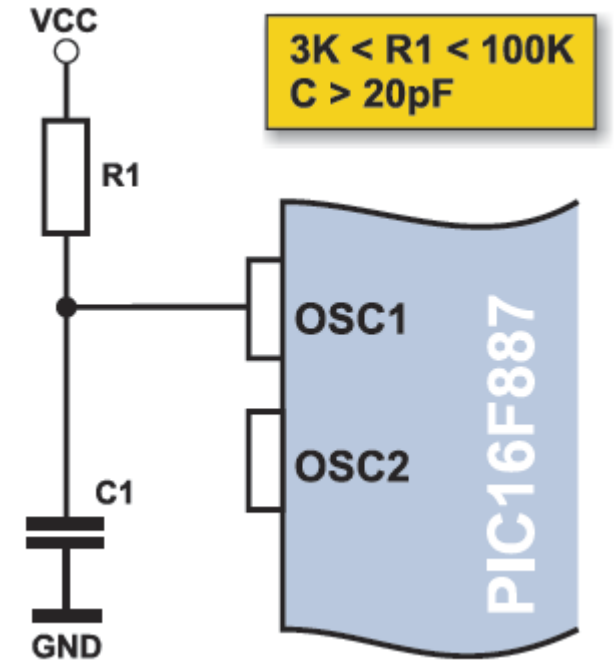


Mode	Frequency	C1, C2
XT	455 KHz	68-100 pF
	2 MHz	15-68 pF
	4 MHz	15-68 pF
HS	8 MHz	10-68 pF
	16 MHz	10-22 pF

Ceramic resonators are usually connected to oscillators when it is not necessary to provide extremely precise frequency.

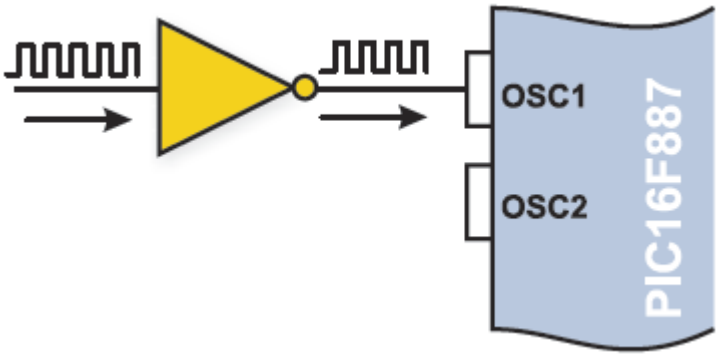
RC Oscillator

If the operating frequency doesn’t matter then there is no need to use additional expensive components for its stabilization. Instead, a simple RC network, as shown in figure below, should be used. In this case only the input of the microcontroller’s clock oscillator is used, which means that the clock signal with the Fosc/4 frequency will appear on the OSC2 pin. This frequency is the same as the operating frequency of the microcontroller, i.e. represents the speed of instruction execution.



External Oscillator

If it is required to synchronize the operation of several microcontrollers or if for some reason it is not possible to use any of the previous configurations, a clock signal may be generated by an external oscillator. Refer to figure below.



4.2 ADDITIONAL COMPONENTS

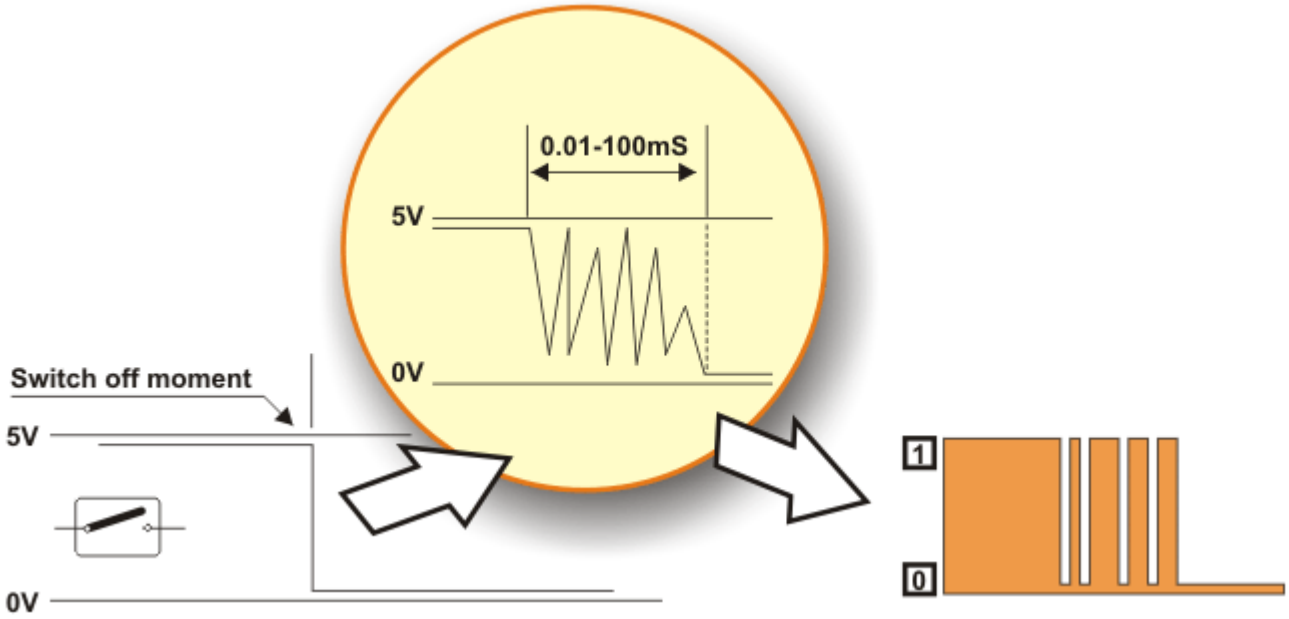
In spite of the fact that the microcontroller is a product of modern technology, it is of no use if not connected to additional components. Simply put, the appearance of voltage on the microcontroller pins means nothing if it is not used for performing certain operations such as turning something on/off, shifting, displaying etc.

This section covers some of the most commonly used additional components in practice such as resistors, transistors, LEDs, LED displays, LCD displays and RS-232 communication modules.

SWITCHES AND PUSH-BUTTONS

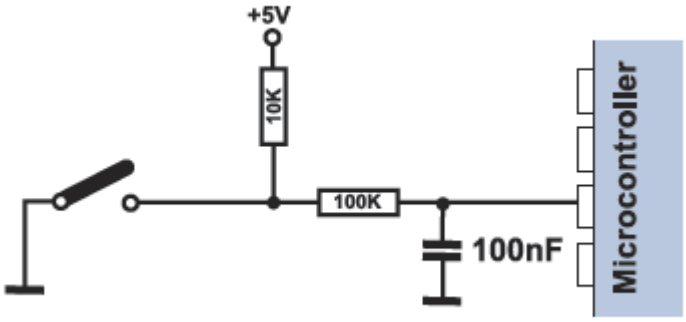
Switches and push-buttons are likely the simplest components which provide the simplest way of bringing voltage on a microcontroller input pins. Of course, it is not as simple as that in practice... What makes it complicated is a contact bounce.

The contact bounce is a common problem with mechanical switches. When the contacts collide together, their momentum and elasticity act together to cause a bounce. The result is a rapidly pulsed electrical current instead of a clean transition from zero to full current. It mostly occurs due to vibrations, slight rough spots and dirt between contacts. The bounce happens too fast so that it is not possible to notice it when these components are normally used.

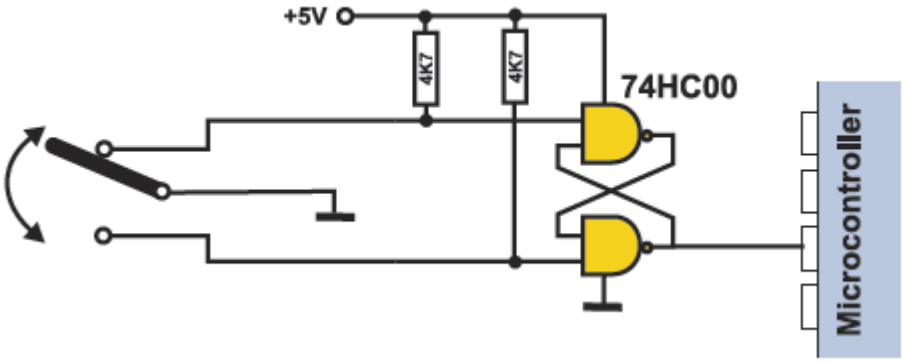


However, it causes problems in some analog and logic circuits that respond fast enough to misinterpret on/off pulses as a data stream. In other words, the whole process doesn’t last long (a few micro- or milliseconds), but long enough to be registered by the microcontroller. When only a push-button is used as a counter signal source, errors occur in almost 100% of cases.

One of possible solutions to this issue is to connect a simple RC circuit to suppress quick voltage changes. Since the bounce period is not defined, the values of components cannot be precisely determined. In most cases it is recommended to use the same values as shown in figure below.



If full stability is required then radical measures should be taken. The output of the logic circuit, as shown in figure below (RS flip-flop), will change its logic state after detecting the first pulse triggered by a contact bounce. This solution is more expensive (SPDT switch), but definitely safer.



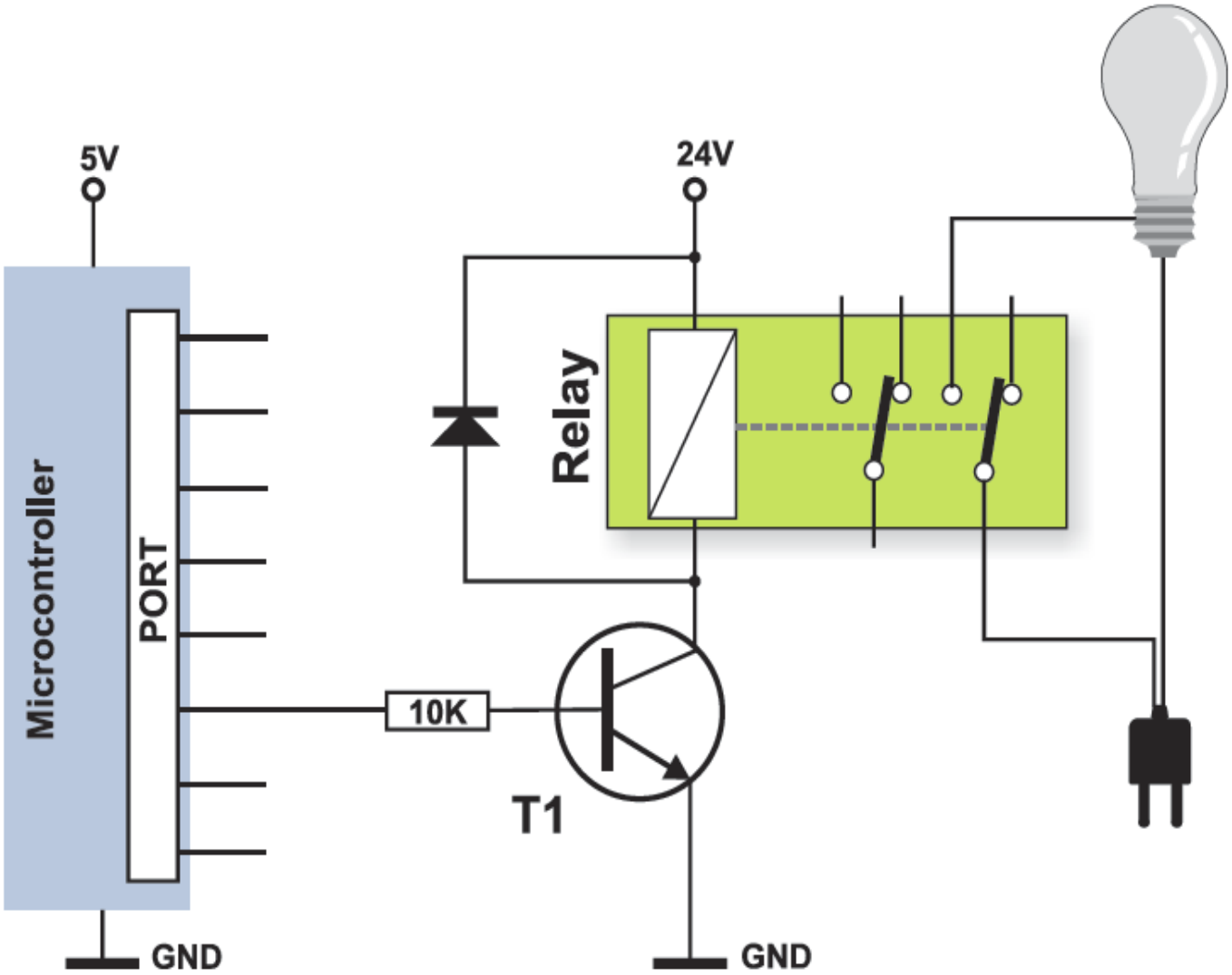
In addition to these hardware solutions, there is also a simple software solution. When the program tests the logic state of an input pin and detects a change, the check should be done one more time after a certain delay. The verification of change means that a switch/push button has changed its position. The advantages of such solution are clear: it is free of charge and can be applied to the poorer quality contacts.

RELAYS

A relay is an electrical switch that opens and closes under the control of another electrical circuit. It is therefore connected to output pins of the microcontroller and used to turn on/off power devices such as motors, transformers, heaters, bulbs, etc. These devices are almost always placed away from the on-board sensitive components. There are various types of relays and all operate in the same way. When current flows

through the coil, the relay is operated by an electromagnet to open or close one or more sets of contacts. Similar to optocouplers, there is no galvanic connection (electrical contact) between relays input and output. Relays usually require both high voltage and high current to start operation, but there are also miniature ones that can be activated by low current directly supplied from a microcontroller pin.

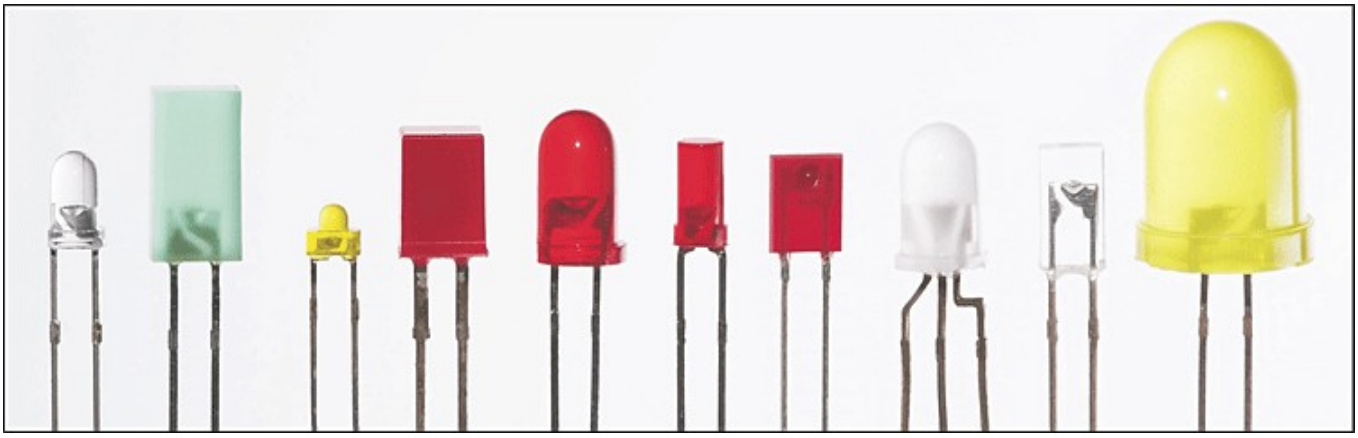
Figure below shows the most common way of connecting a relay to other mains powered devices.



In order to prevent the high voltage self-induction, caused by a sudden stop of the current flow through the coil, an inverted polarized diode is connected in parallel to the coil. The purpose of this diode is to ‘cut off’ the voltage peak.

LED DIODES

You probably know all you should know about LEDs, but we should also think of the younger generations... Let’s see, how to destroy a LED?! Well...Easily.



Quick Burning

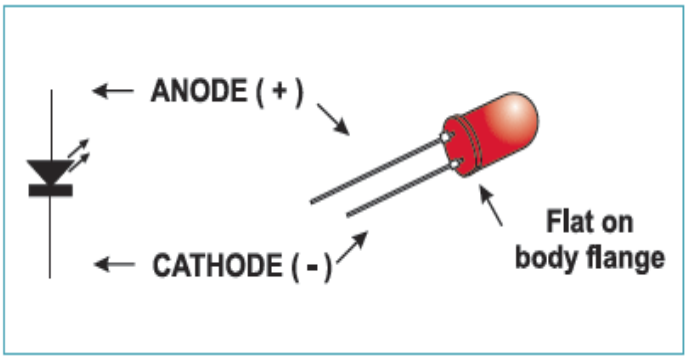
Like any other diode, LEDs have two ends - an anode and a cathode. Connect a diode properly to the power supply voltage and it will emit light happily. Turn the diode upside down and apply the same power supply voltage (even for a moment). It will probably not emit light - EVER AGAIN!

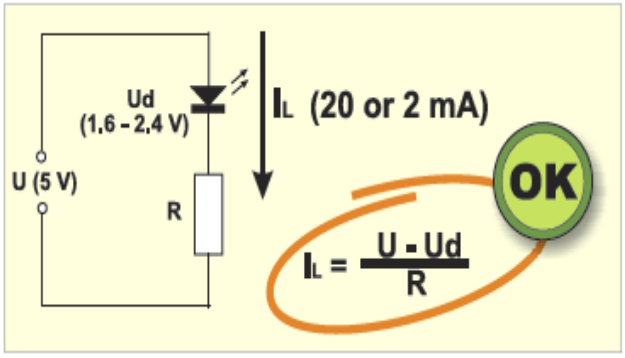
Slow Burning

There is a nominal (consider it maximum) current specified for every LED which must not be exceeded. If it happens, the diode will emit more intensive light, but just for a short period of time.

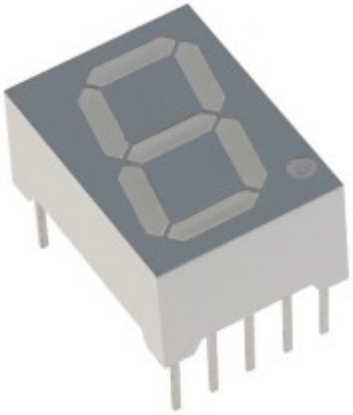
Something to Remember

Similarly, all you need to do is to remove a current limiting resistor shown below. Depending on the power supply voltage, the effects might be spectacular.





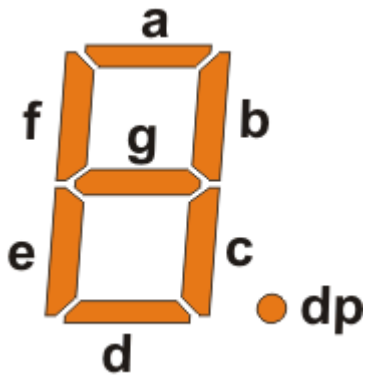
LED DISPLAY



Basically, an LED display is nothing more than several LEDs molded in the same plastic case. There are many types of displays and some of them are composed of several dozens built-in diodes which can display different symbols. Nevertheless, the most commonly used display is a 7-segment display. It is composed of 8 LEDs. Seven segments of a digit are arranged as a rectangle to display symbols, whereas the additional segment is used to display a decimal point. In order to simplify connection, anodes or cathodes of all diodes are connected to one single pin so that there are common anode displays and common cathode displays, respectively. Segments are marked with letters from a to g, plus dp, as shown in figure below. When connecting an LED display, each diode is treated separately, which means that each one must have its own current limiting resistor.

Here are a few things that you should pay attention to when buying LED displays:

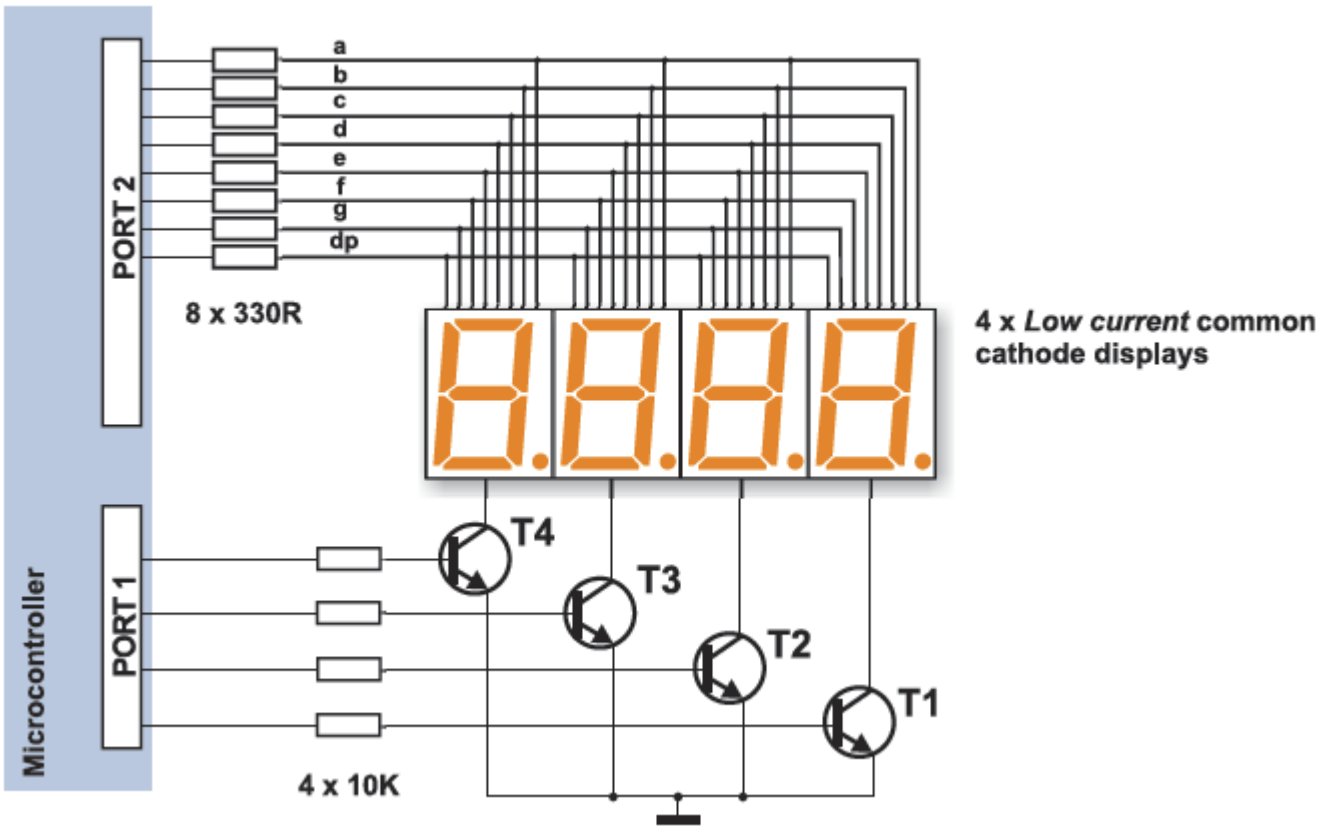
- As mentioned above, depending on whether anodes or cathodes are connected to the common pin, there are common anode displays and common cathode displays. There is no difference between them at all in their appearance so you are advised to double check which one is to be used prior to installing it.
- Maximum current that each microcontroller pin can receive or give is limited. Therefore, if several displays are connected to the microcontroller then so called Low current LEDs limited to only 2mA should be used.
- Display segments are usually marked with letters from a to g, but there is no fast rule indicating display pins they are connected to. For this reason, it is very important to check connection prior to start writing a program or designing a device.



LED displays connected to the microcontroller usually occupy a large number of valuable I/O pins, which can be troublesome especially when it is necessary to display multi digit numbers. It is even more complicated if, for example, it is necessary to display two 6-digit numbers. A simple calculation shows that

96 output pins are needed in this case. The solution to this issue is called multiplexing.

Here is how an optical illusion based on the same operating principle as a film camera is made. Only one digit is active at a time, but they change their on/off conditions so quickly creting the impression that all digits of a number are simultaneously active.

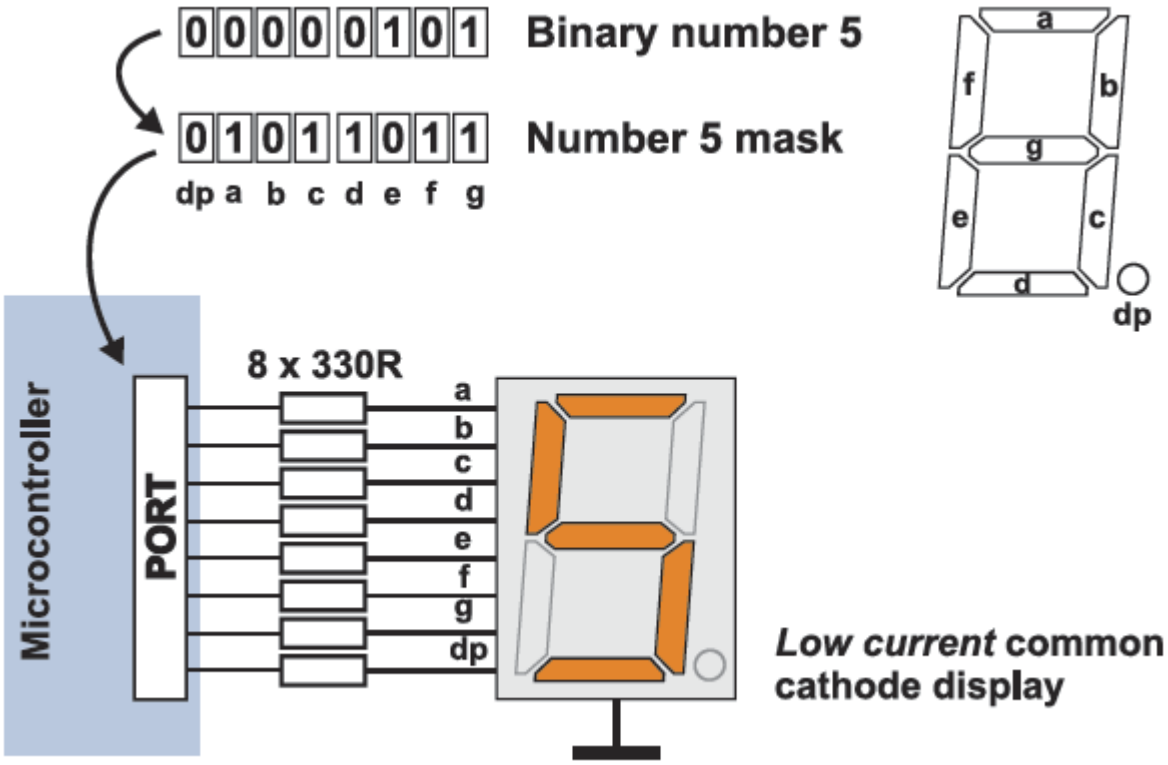


Let’s take a look at the figure above. First a byte representing units is loaded to the microcontroller port PORT2 and transistor T1 is activated at the same time. After a while, the transistor T1 is turned off, a byte representing tens is loaded to PORT2 and the transistor T2 is activated. This procedure is repeated cyclically at high speed for all digits and corresponding transistors.

A disappointing fact which indicates that the microcontroller is just a kind of miniature computer designed to understand only the language of zeros and ones is fully expressed when displaying digits. Namely, the microcontroller does not know what units, tens or hundreds are, nor what ten digits we are used to look like. For this reason, each number to be displayed must undergo the following procedure:

First of all, a multi digit number must be split into units, tens etc. in a specialized subroutine. Then each of these digits must be stored in a specific byte. Digits get recognizable appearance for humans by performing a simple procedure called ‘masking’. In other words, a binary number is replaced with a different combination of bits. For example, digit 8 (0000 1000) is replaced with the binary number 0111 1111 in order to activate all LEDs displaying this digit. The only diode remaining inactive here is reserved for the decimal point.

If the microcontroller port is connected to a display so as that bit 0 activates segment ‘a’, bit 1 activates segment ‘b’, bit 2 segment ‘c’ etc., then table below shows appropriate binary mask for each digit.



Digits to display	Display Segments
	dp a b c d e f g
0	0 1 1 1 1 1 1 0
1	0 0 1 1 0 0 0 0
2	0 1 1 0 1 1 0 1
3	0 1 1 1 1 0 0 1
4	0 0 1 1 0 0 1 1
5	0 1 0 1 1 0 1 1
6	0 1 0 1 1 1 1 1
7	0 1 1 1 0 0 0 0
8	0 1 1 1 1 1 1 1
9	0 1 1 1 1 0 1 1

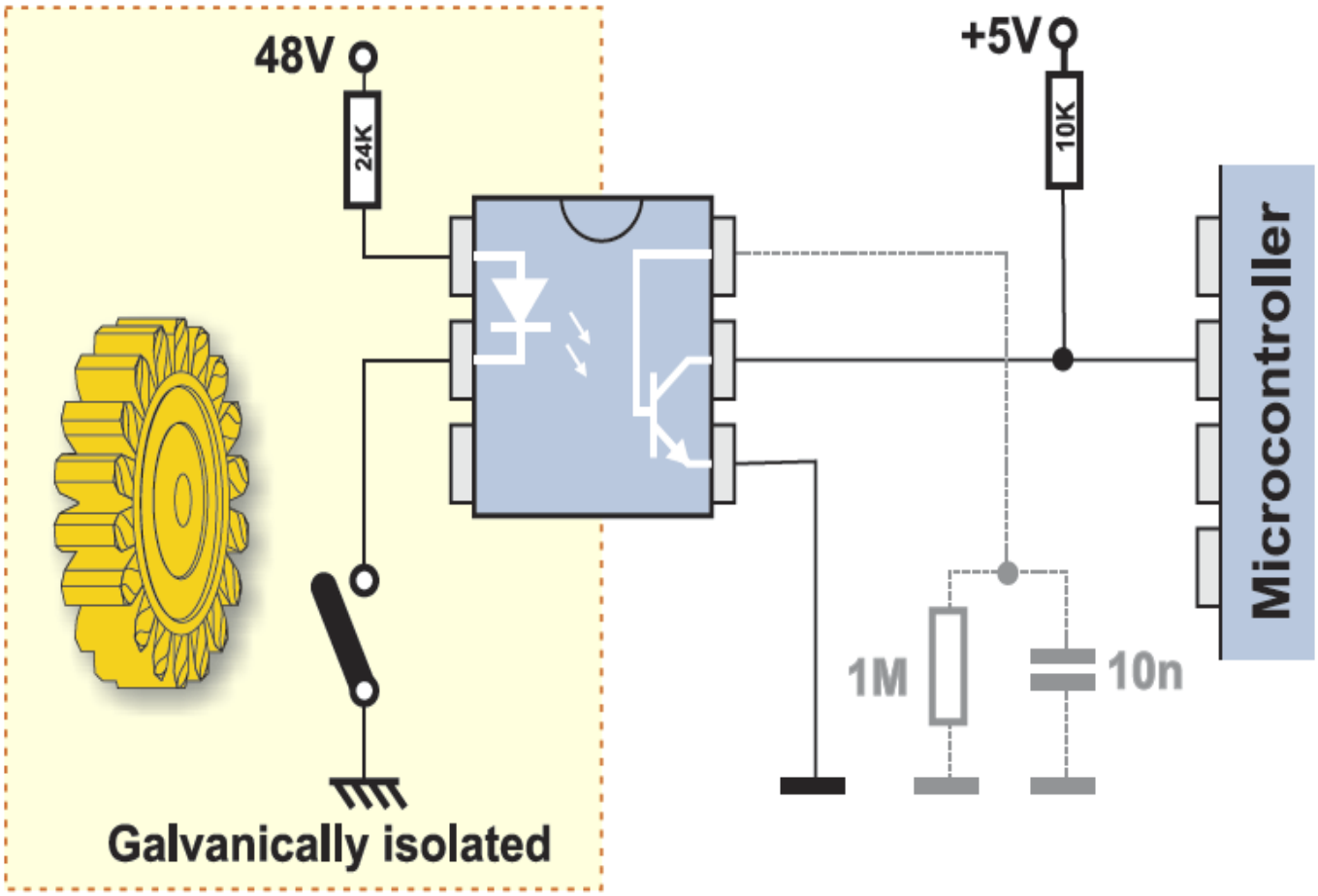
In addition to digits (0-9), there are some letters of alphabet - A, C, E, J, F, U, H, L, b, c, d, o, r, t- that can also be displayed by masking.

In case that common anode displays are used, all ones contained in the table above should be replaced with

zeros and vice versa. Additionally, PNP transistors should be used as drivers.

OPTOCOUPLER

An optocoupler is a component commonly used to galvanically separate microcontroller from any potentially dangerous current or voltage in its surrounding. Optocoupler usually have one, two or four light sources (LEDs) on its input while on its output, opposite to diodes, there is the same number of light-sensitive components (phototransistors, photo-thyristors or phototriacs). The point is that an optocoupler uses a short optical transmission path to transfer a signal between diodes and photo-sensitive components, while keeping them electrically isolated. Such isolation makes sense only if these components are separately powered. This way, the microcontroller and its additional modules are fully protected from high voltage and noises which usually cause them to be damaged or to operate in an unpredictable way. The most frequently used optocouplers are those with phototransistors on their output. When it comes to the optocouplers with internal base-to-pin 6 connection (there are also optocouplers without it), the base can be left unconnected.

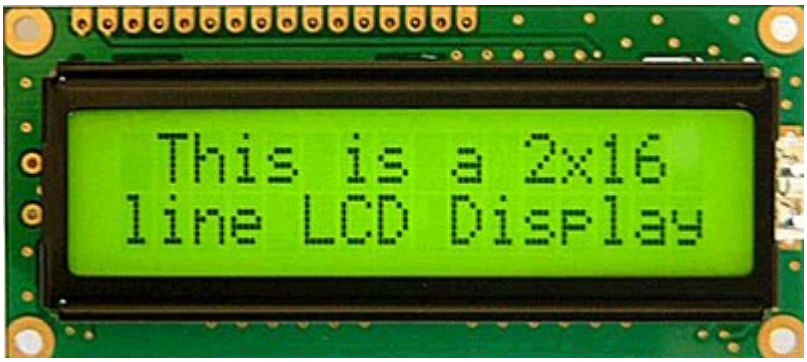


The R/C network outlined by a dotted line in figure above is an optional connection which lessens the effects of noise by eliminating very short pulses.

LCD DISPLAY

An LCD display is specifically manufactured for the use with microcontrollers, which means that it cannot be activated by standard IC circuits. It is used to display different messages on a miniature liquid crystal display. The LCD display described here is for its low price and great capabilities most frequently used in practice. It

is based on the HD44780 controller (Hitachi) and displays messages in two lines with 16 characters each. Different symbols such as letters of alphabet, Greek letters, punctuation marks, mathematical symbols etc. can be displayed on it. It is also possible to display symbols created by the user. Other useful features include automatic message shift (left and right), cursor appearance, LED backlight etc.



LCD Display Pins

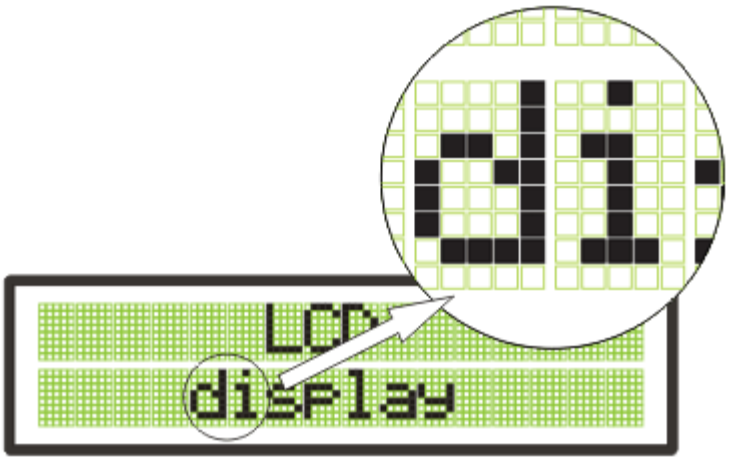
Along one side of the small printed board of the LCD display there are pins which enable it to be connected to the microcontroller. There are in total 14 pins marked with numbers (16, if there is a backlight available). Their functions are described in table bellow:

Function	Pin Number	Name	Logic State	Description
Ground	1	Vss	-	0V
Power supply	2	Vdd	-	+5V
Contrast	3	Vee	-	0 - Vdd
Control of operating	4	RS	0 1	D0 – D7 considered as commands D0 – D7 considered as data
	5	R/W	0 1	Write data (from controller to LCD) Read data (from LCD to controller)
	6	E	0 1 From 1 to 0	Access to LCD disabled Normal operating Data/commands being transferred to LCD
Data / commands	7	D0	0/1	Bit 0 LSB
	8	D1	0/1	Bit 1
	9	D2	0/1	Bit 2
	10	D3	0/1	Bit 3
	11	D4	0/1	Bit 4

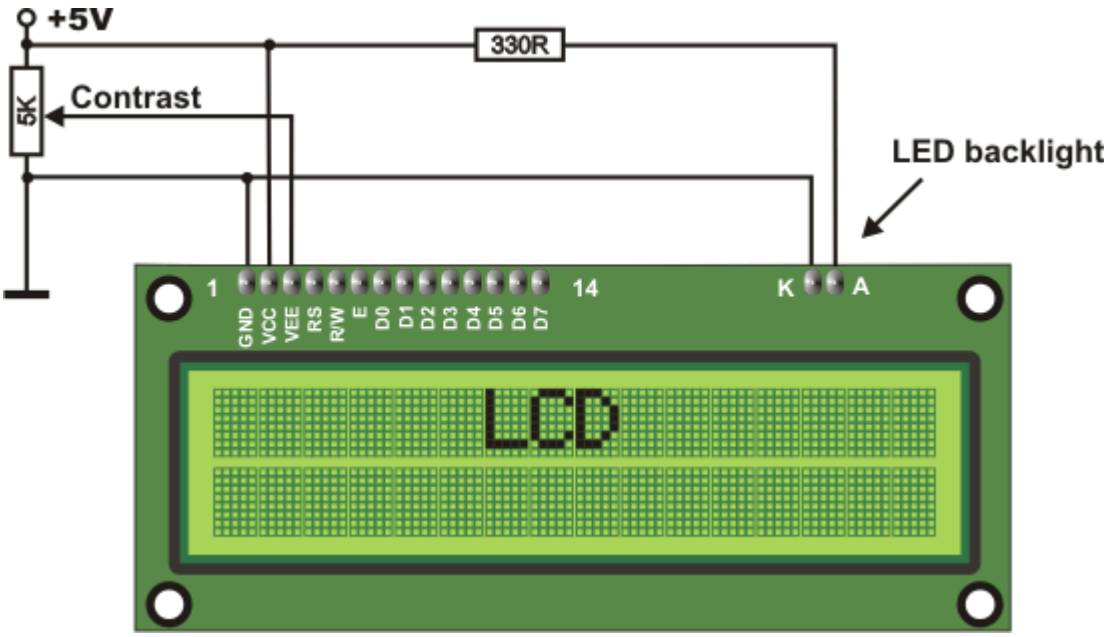
12	D5	0/1	Bit 5
13	D6	0/1	Bit 6
14	D7	0/1	Bit 7 MSB

LCD Screen

An LCD screen is a thin, flat panel used for displaying different contents. It consists of two lines each containing up to 16 characters of 5x8 or 5x11 pixels. The operation of a 5x8 LCD display will be described here as it is more frequently used.



Display contrast depends on the power supply voltage and whether messages are displayed in one or two lines. For this reason, a varying voltage (0-Vdd) is applied to the pin marked as Vee by using a trimmer potentiometer. Some LCD displays have a built-in backlight (blue or green LEDs). When the backlight is used, a current limiting resistor should be serially connected to one of the pins for backlight power supply (similar to LEDs).



If there are no characters displayed or if they are dimmed when the display is switched on, the first thing that

should be done is to check the potentiometer for contrast adjustment. Is it properly adjusted? The same applies if the operating mode of display has been changed (write in one or two lines).

LCD Memory

LCD display contains three memory blocks:

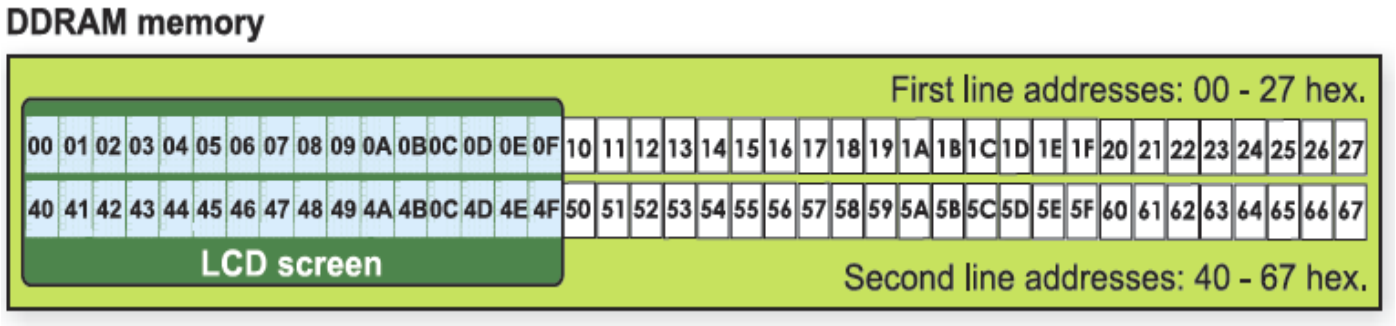
- DDRAM Display Data RAM;
- CGRAM Character Generator RAM; and
- CGROM Character Generator ROM.

DDRAM Memory

The DDRAM memory is used for storing characters to be displayed. It is capable of storing up to 80 characters. Some memory locations are directly related to the character fields on the screen.

The principle of DDRAM memory operation is quite simple: it is sufficient to configure a display to increment addresses automatically (shift right) and set the starting address for the message to be displayed (for example 00 hex).

After that, all characters sent through lines D0-D7 will be displayed on the screen as a message we are used to - from left to right. In this case, displaying starts from the first character field in the first line because the starting address is 00 hex. No matter how many characters are sent, only the first sixteen will be visible on the screen, while the rest of them will be saved and displayed afterwards using the shift command. Practically, LCD display is like a window shifting in left-right direction over memory locations containing different characters. This is in fact how the effect of the message shifting over the screen has been created.

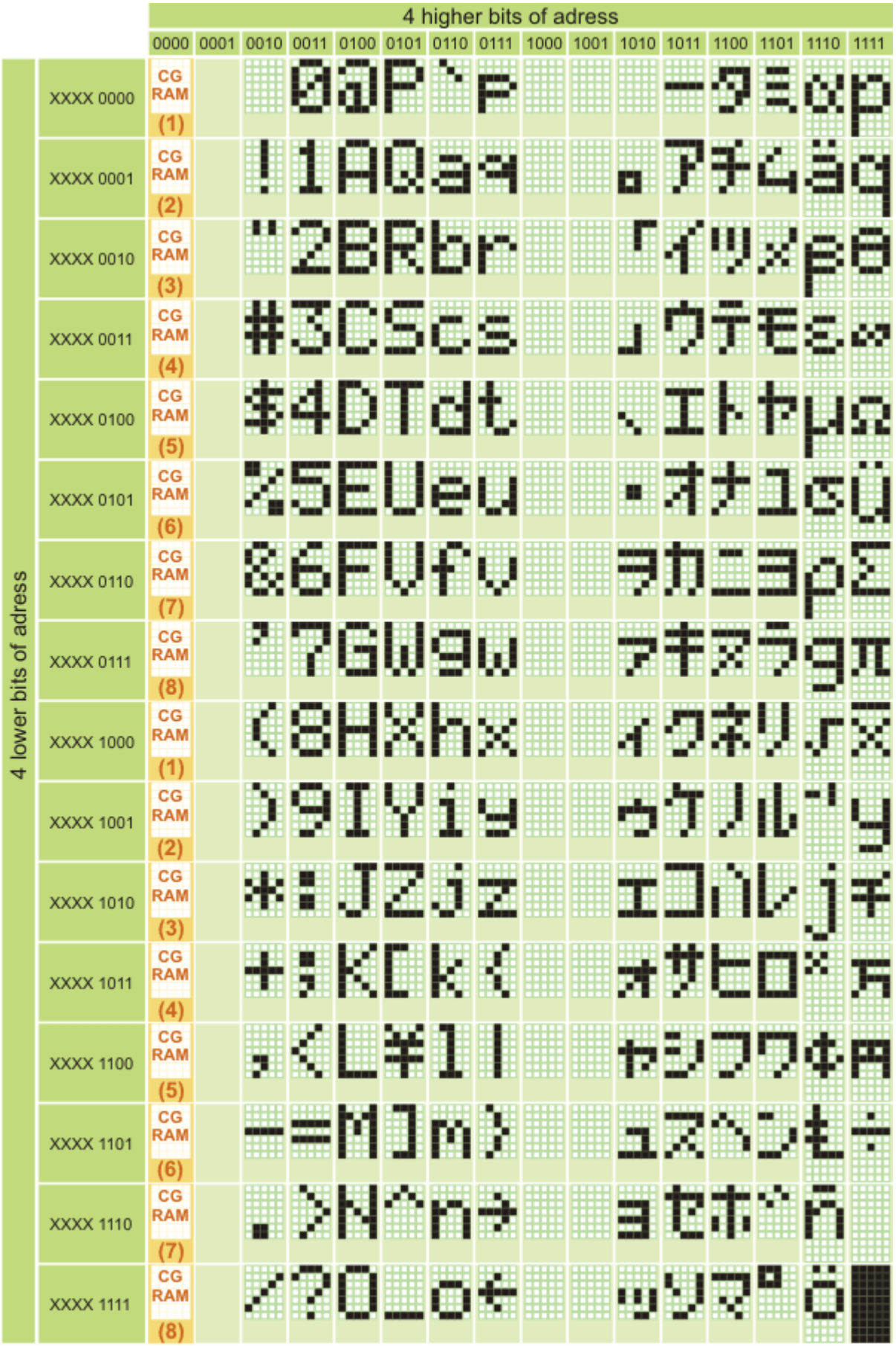


If the cursor is enabled, it is always positioned at the currently addressed character field. In other words, as soon as the appropriate character appears at the cursor position, the coursor automatically moves to the next addressed field.

As its name suggests, DDRAM memory is a kind of RAM, which means that data can be written to and read from it, while its content is irretrievably lost when the power goes off.

CGROM Memory

CGROM memory contains a standard character map with all characters that can be displayed on the screen. Each character is assigned one memory location:



Addresses of CGROM memory locations match standard ASCII characters. Let’s see what it acctually means. If the program being executed by the microcontroller encounters a command ‘send character P to port’, binary value 0101 0000 will appear on the port. This value is ASCII equivalent of character P. As a

result, the symbol matching the 0101 0000 CGROM memory location, i.e. letter P, will be displayed on the screen. The same applies to all letters of alphabet (capitals and small), but not to numbers.

If you look carefully at the map on the previous page, you will notice that addresses of all digits are shifted forward by 48 relative to their values (digit 0 address is 48, digit 1 address is 49, digit 2 address is 50 etc.). Therefore, in order to display digits correctly it is necessary to add decimal number 48 to each of them prior to sending it to an LCD.

ASCII	Hex	Symbol
0	0	NUL
1	1	SOH
2	2	STX
3	3	ETX
4	4	EOT
5	5	ENQ
6	6	ACK
7	7	BEL
8	8	BS
9	9	TAB
10	A	LF
11	B	VT
12	C	FF
13	D	CR
14	E	SO
15	F	SI

ASCII	Hex	Symbol
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

ASCII	Hex	Symbol
32	20	(space)
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

ASCII	Hex	Symbol
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII	Hex	Symbol
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

ASCII	Hex	Symbol
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

ASCII	Hex	Symbol
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

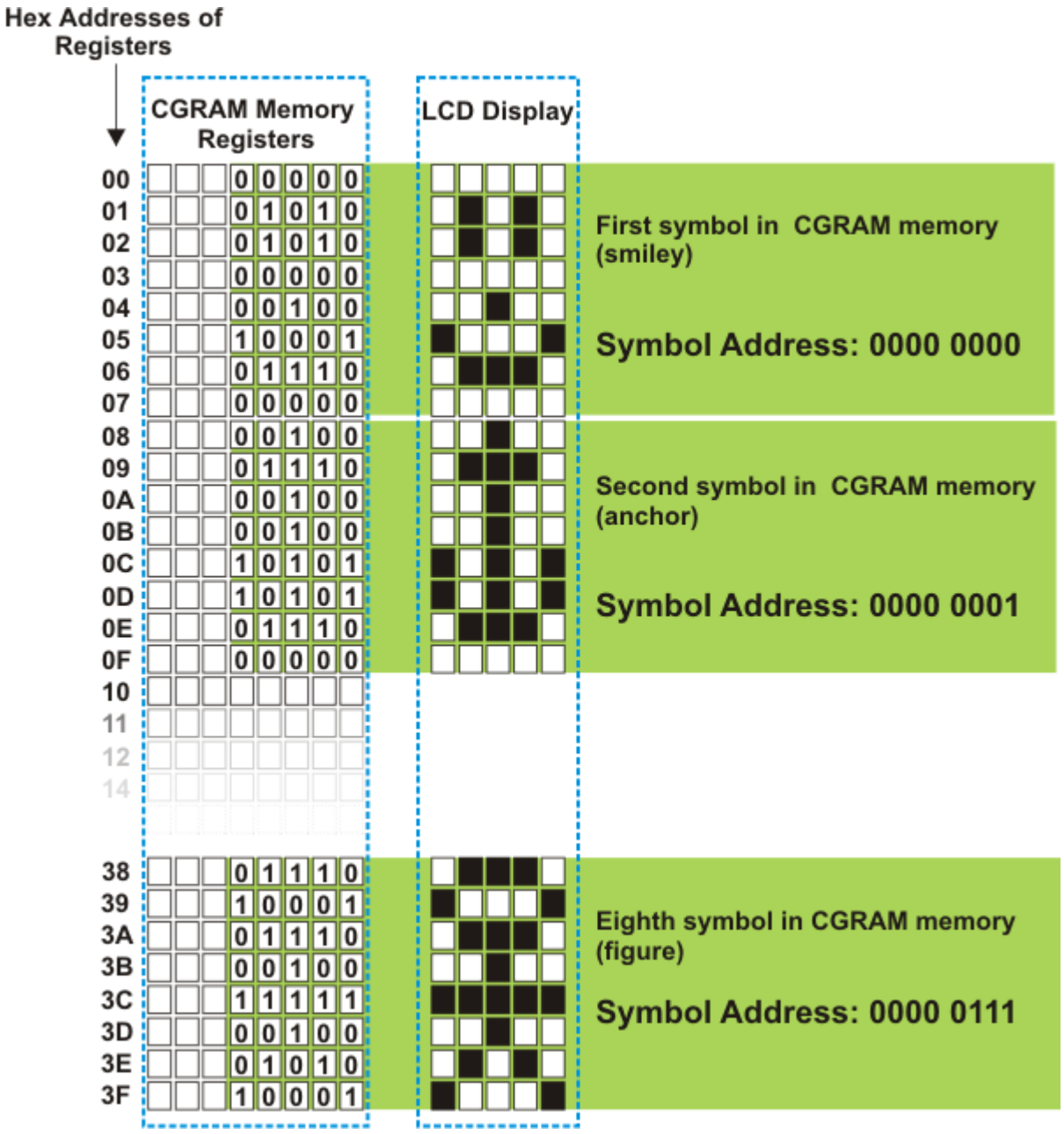
ASCII	Hex	Symbol
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	

What is ASCII? From their inception till today, computers can recognize only numbers, but not letters. It means that all data a PC swaps with a peripheral device is converted into binary format even though the same is recognized by humans as letters (keyboard is an excellent example). In other words, every character matches a unique combination of zeroes and ones. ASCII is character encoding based on the English alphabet. ASCII code specifies a correspondence between standard character symbols and their numeric equivalents.

CGRAM MEMORY

In addition to standard characters, the LCD display can also display user-defined symbols in size of 5x8 pixels. It is enabled by a special type of RAM called CGRAM (64 bytes).

Memory registers are 8 bits wide, but only 5 lower bits are used. Logic one (1) in every register represents a dimmed field, while 8 locations grouped together represent one character. Refer to figure below:



Symbols are usually defined at the beginnig of the program by simple writing zeros and ones to registers of CGRAM memory so as to form desired shapes. In order to display them, it is sufficient to specify appropriate memory address. Pay attention to the first column of the CGROM map of characters. It doesn't contain RAM memory addresses, but symbols being discussed here. In this example, ‘display 0’ means - display ‘smiley’, ‘display 1’ means - display ‘anchor symbol’ etc.

LCD Basic Commands

All data sent to an LCD through the pins D0-D7 will be interpreted either as a command or a data, which depends on the logic state of the RS pin:

- **RS** = **1** - Bits D0 - D7 are addresses of the characters to be displayed. A built-in LCD processor addresses one character from the character map and displays it. The DDRAM address specifies location on the screen on which the character is to be displayed. This address is predefined or the address of the previously sent character is automatically incremented.
- **RS** = **0** - Bits D0 - D7 are commands used for setting the operating mode of display.

Here is a list of commands related to the operation of LCD:

Command	RS	RW	D7	D6	D5	D4	D3	D2	D1	D0	Execution Time
Clear display	0	0	0	0	0	0	0	0	0	1	1.64mS
Cursor home	0	0	0	0	0	0	0	0	1	x	1.64mS
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	40uS
Display on/off control	0	0	0	0	0	0	1	D	U	B	40uS
Cursor/Display Shift	0	0	0	0	0	1	D/C	R/L	x	x	40uS
Function set	0	0	0	0	1	DL	N	F	x	x	40uS
Set CGRAM address	0	0	0	1	CGRAM address						40uS
Set DDRAM address	0	0	1	DDRAM address							40uS
Read "BUSY" flag (BF)	0	1	BF	DDRAM address							-
Write to CGRAM or DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	40uS
Read from CGRAM or DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	40uS

```
I/D 1 = Increment (by 1)          R/L 1 = Shift right
    0 = Decrement (by 1)              0 = Shift left

S 1 = Display shift on            DL 1 = 8-bit interface
  0 = Display shift off              0 = 4-bit interface

D 1 = Display on                  N 1 = Display in two lines
  0 = Display off                   0 = Display in one line

U 1 = Cursor on                   F 1 = Character format 5x10 dots
  0 = Cursor off                    0 = Character format 5x7 dots
B 1 = Cursor blink on             D/C 1 = Display shift
  0 = Cursor blink off                0 = Cursor shift
```

WHAT IS A BUSY FLAG?

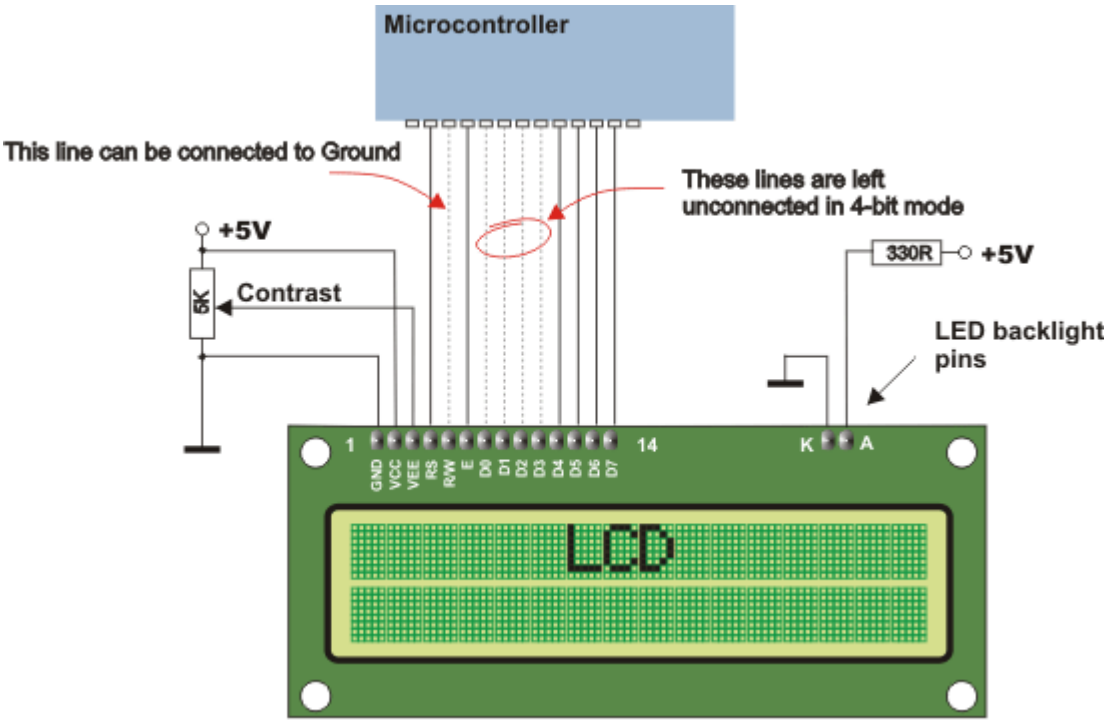
Compared to the microcontroller, LCD is an extremely slow component. For this reason, it was necessary to provide a signal which would, upon each command execution, indicate that the display is ready to receive next piece of data. This signal is called the busy flag and can be read from the D7 line. LCD display is ready to receive new data when the voltage on this line is 0V (BF=0).

LCD Connecting

Depending on how many lines are used for connecting an LCD to the microcontroller, there are 8-bit and 4-

bit operating modes of LCD. The appropriate mode is selected at the beginning of the operation in the process called ‘initialization’. The 8-bit LCD mode uses pins D0-D7 to transfer data as explained on the previous page.

The main purpose of the 4-bit LCD mode is to save valuable I/O pins of the microcontroller. Only 4 higher bits (D4-D7) are used for communication here, while others may be left unconnected. Each piece of data is sent to the LCD in two steps - four higher bits are sent first (normally via lines D4-D7), then four lower bits. As a result of the process of initialization, the LCD is able to link and interpret received bits correctly.



In addition, data is rarely read from the LCD. In most cases the microcontroller sends data to the LCD, which means that it is possible to save an extra I/O pin by simple connecting the R/W pin to Ground. Such saving has its price, of course. Even though the process of displaying data will be normally performed, it will not be possible to read the busy flag as it is not possible to read the display either. Good news is that there is a simple solution to this issue. After sending a character or a command to the LCD it is necessary to give it enough time to get ready for another receive. Owing to the fact that it takes approximately 1.64mS for one command to be executed, it will be sufficient to wait about 2mS.

LCD Initialization

As soon as the power supply goes on, the LCD is automatically cleared. The whole process lasts approximately 15mS. After that, the display is ready for operation and its operating mode is set by default. It means that:

1. Display is cleared.
2. Mode
   DL = 1 - Communication through 8-bit interface
   N = 0 - Data is displayed in one line
   F = 0 - Character font format is 5 x 8 pixel
3. Display/Cursor on/off
   D = 0 - Display off
   U = 0 - Cursor off

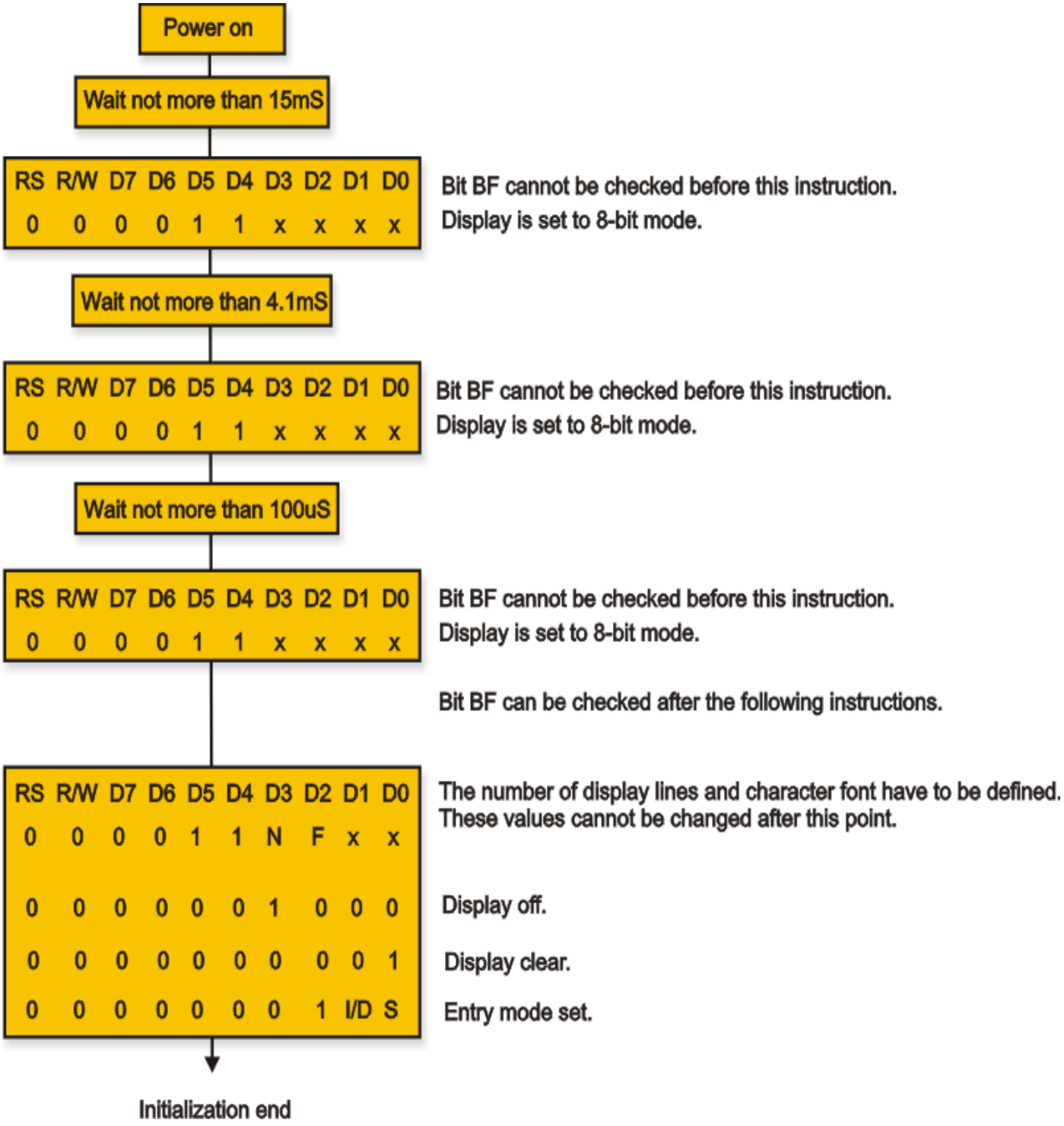
B = 0 - Cursor blink off

4. Character entry
   ID = 1 Display addresses are automatically incremented by 1
   S = 0 Display shift off

In most cases auto-reset normally occurs. Mostly, but not always. If for some reason the power supply voltage doesn't reach the maximum within 10mS, the display will start to perform completely unpredictably. If the power supply is not able to meet this condition or if it is necessary to provide safe operation, the process of initialization is required. It causes a new reset condition, thus enabling the display to operate normally.

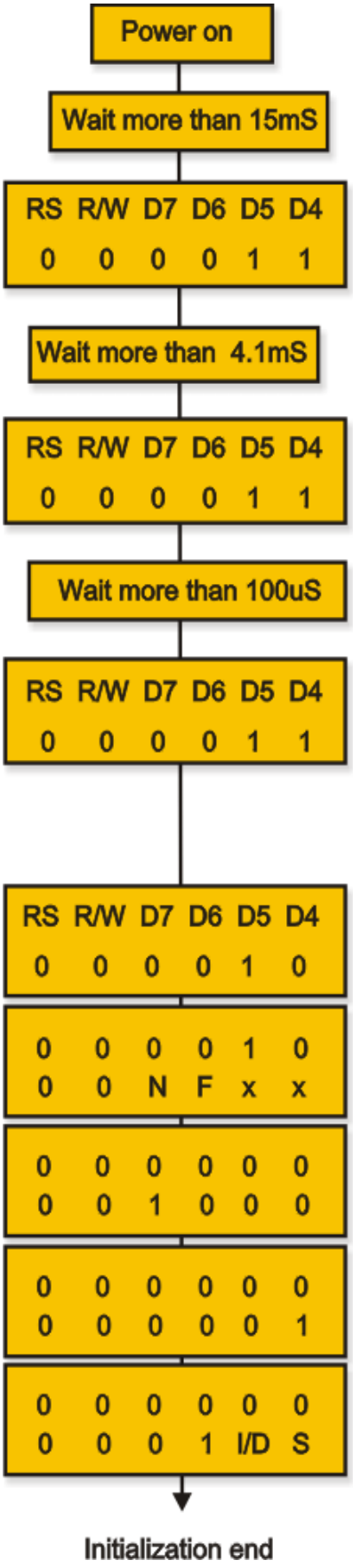
There are two initialization algorithms. Which one is to be performed depends on whether the connection to the microcontroller is established through 4- or 8-bit interface. The process following initialization is the same for both algorithms. You just have to specify a few basic commands and after that, you will be able to send massages to LCD display.

Figure below illustrates the 8-bit initialization of LCD:



It is not a mistake! In this algorithm, the same value is three times successively sent and displayed to the LCD display.

The procedure in the 4-bit initialization is as follows:



Bit BF cannot be checked before this instruction.
Display is set to 8-bit mode.

Bit BF cannot be checked before this instruction.
Display is set to 8-bit mode.

Bit BF cannot be checked before this instruction.
Display is set to 8-bit mode.

Bit BF can be checked after the following instructions.

Start operation in 4-bit mode.
After this point 4 higher bits are written first,
4 lower afterwards.

The number of displays lines and character font have to be defined.
These values cannot be changed after this point.

Display off.

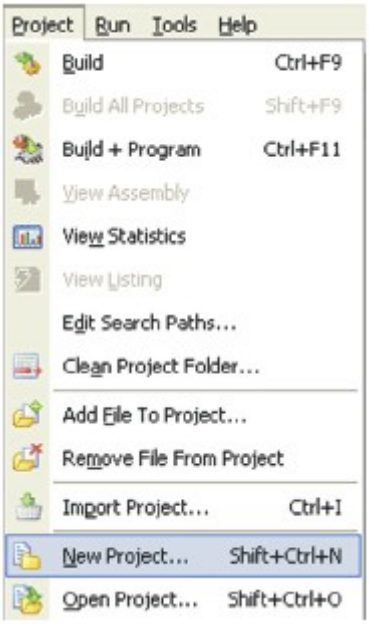
Display clear.

Set entry mode.

Let's do it in mikroBasic...

```
'In mikroBasic for PIC, it is sufficient to write only one function to perform the
'whole process of LCD initialization. Prior to calling this function it is necessary
'to declare bits LCD_D4-LCD_D7, LCD_RS and LCD_EN.
...
sub procedure Lcd_Init ' Initialize LCD
...
```

PRACTICAL EXAMPLES



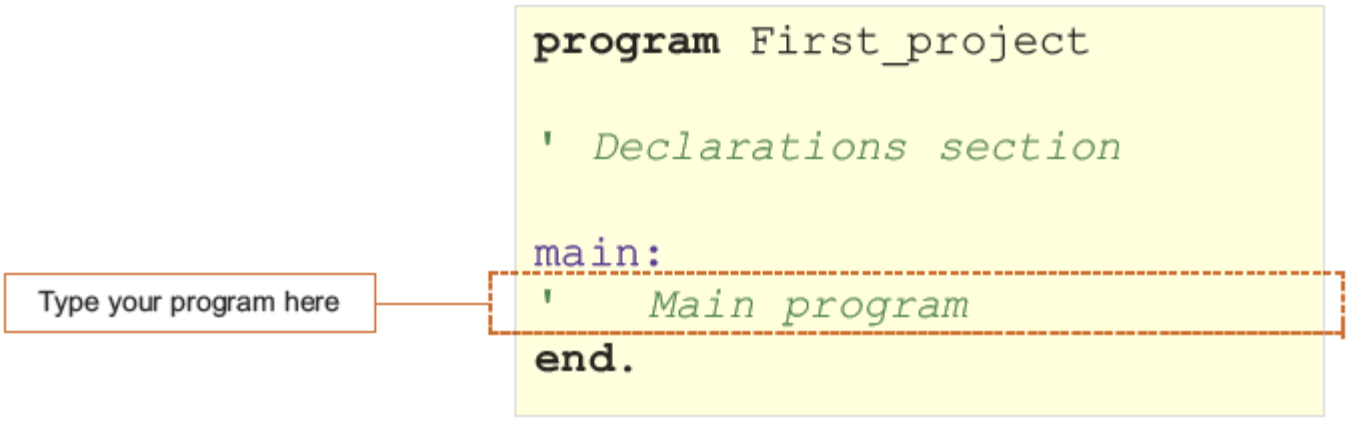
The process of creating a new project is very simple. Select the New Project option from the Project menu, as shown in Figure on the right.

A window called New Project Wizard, which will guide you through the process of creating a new project, appears. The introductory window of this application contains a list of actions to be performed when creating a new project. Click Next.

The process of creating a new project can be broken up into five steps:

1. Selecting the microcontroller to write a program for. In this case it is PIC16F887.
2. Selecting the device clock. In this case, it is 8 MHz clock.
3. Selecting the name and location of the project. In this case, the project name is First _Project and it will be saved in the C:\My projects folder. The compiler automatically appends the .mbppi extension to the project name and a source file having the same name (First_Project .mbas) will be created within it.
4. In case the project consists of several source files, it is necessary to specify them all and include into the project by clicking the Add button. In this example, there are no additional source files within the project.
5. Finally, it is necessary to approve all selected options by clicking Finish.

After creating the project, a new blank window to write a program in will appear. See Figure below.



When the program is written, it is necessary to compile it into a .hex code, by selecting one of the build options from the Project menu:

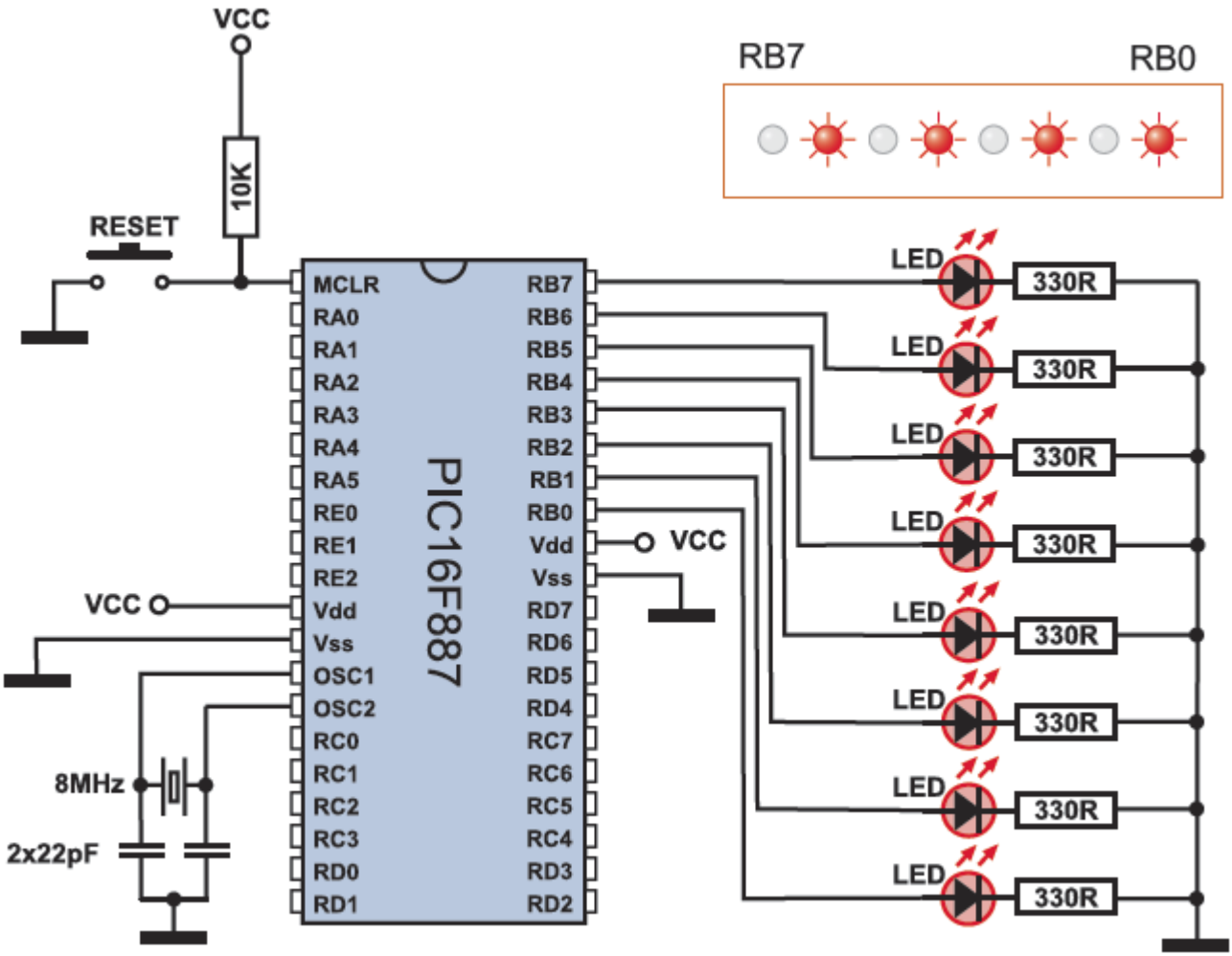
- To create a .hex file, select Build (Ctrl+F9) from the Project menu or click the Build icon from the Project toolbar.
- The Build All Projects (Shift+F9) option builds all files within the project, libraries (if there is a source code for them) and def files for the chip in use.
- The Build + Program (Ctrl+F11) option is special as it enables the mikroBasic PRO for PIC compiler to automatically load the program into the microcontroller after compilation. The process of programming is performed by using the PICflash programmer.

All the errors detected during compilation will be shown in the Messages window. If no errors are encountered, the mikroBasic PRO for PIC compiler will generate output files.

4.3 EXAMPLE 1

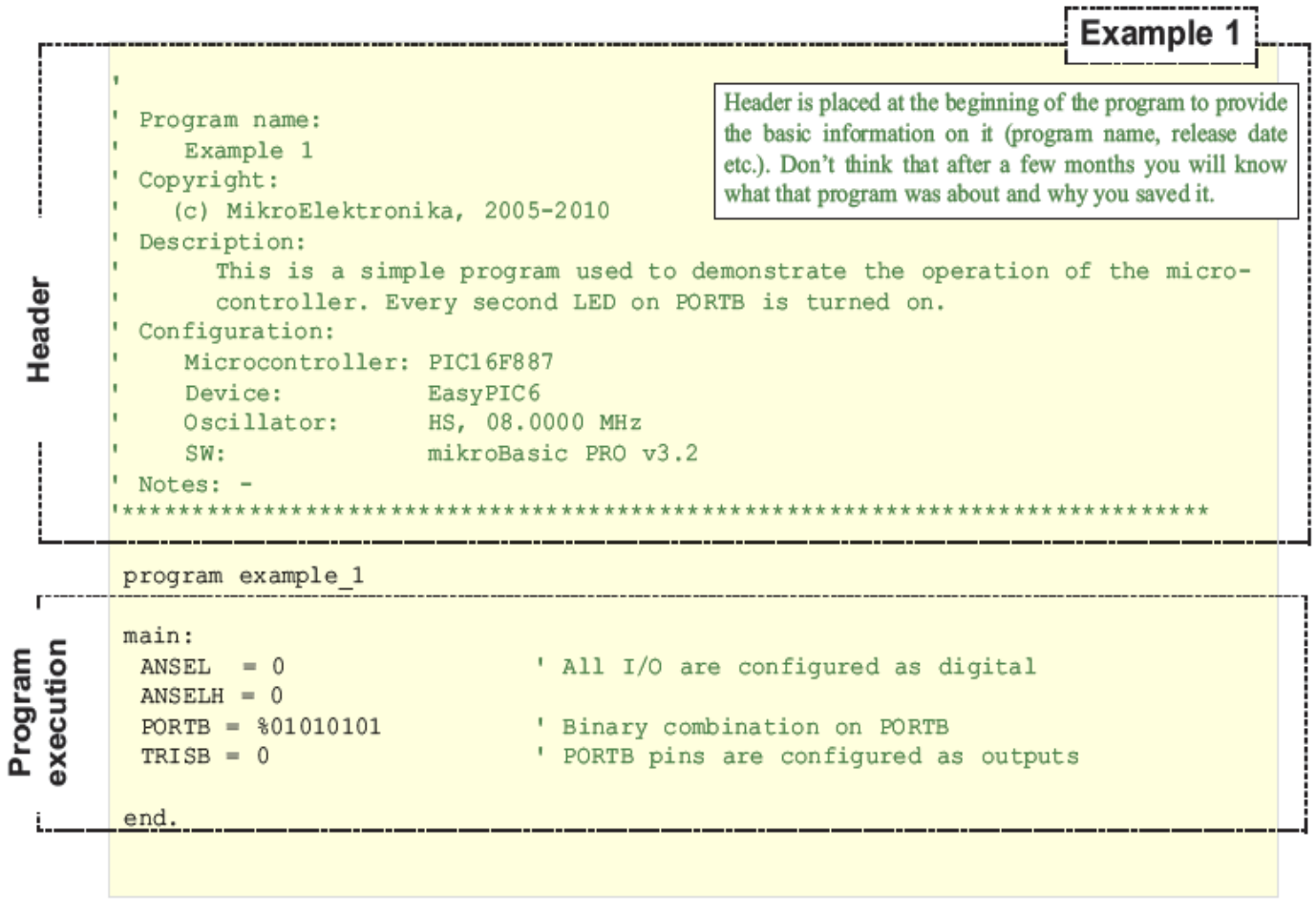
Write header, configure I/O pins and use delay function

Here is a simple program whose purpose just to turn on a few LEDs on PORTB. Use this example to study what a real program looks like. Figure below shows appropriate connection schematic, while the related program is on the next page.



When the power supply goes on, every second LED on PORTB will emit light, thus indicating that the microcontroller is properly connected and operates normally.

This example shows how a correctly written header looks like. Header is the same for all the programs described herein so it will be skipped in the following examples. Anyway, it is considered to be at the beginning of every program marked as Header.



To make this example more interesting, we will enable LEDs connected to PORTB to blink. There are several ways to do it:

1. As soon as the microcontroller is turned on, all LEDs will emit light for a second. The Delay function is in charge of it in the program. You just need to set delay expressed in milliseconds.
2. After one second, the program enters the for loop and remains there as long as variable k is less than 20. The variable is incremented by 1 after each iteration. Within the for loop, duty cycle of pulses is 5:1 (500mS:100mS) and any change of logic state on output pins causes all LEDs to blink.
3. When the program exits the for loop, the PORTB logic state changes (0xb 01010101) and the program enters the endless while loop and remains there as long as 1=1(endless loop). In this loop the PORTB logic state is inverted each 200mS.

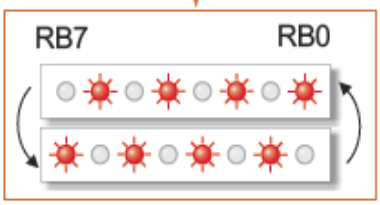
Example 1a

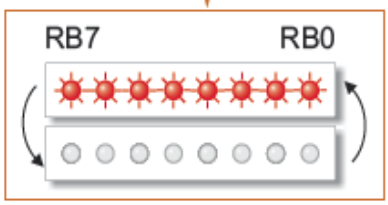
```
' Header ************************************************
program example_1a     ' Program name
dim k as byte          ' Variable k is of byte type

main:                  ' Start of program
 ANSEL  = 0            ' All I/O pins are configured as digital
 ANSELH = 0
 PORTB = 0xFF          ' Reset PORTB
 TRISB = 0             ' PORTB pins are configured as outputs

 Delay_ms(1000)        ' 1s delay

 PORTB = 0             ' Initial state of port PORTB

for k=1 to 20          ' Remain in the loop as long as 1<k<20, k is incremented
                       ' by 1 after each iteration

  if PORTB = 0 then            ' If PORTB=0,
    PORTB = 0xFF               ' change its state into 0xFF
    Delay_ms(50)               ' and provide 50mS delay
  else                         ' If PORTB is not 0
    PORTB = 0                  ' write 0 to PORTB
    Delay_ms(500)              ' and provide 500mS delay
  end if

next k                         ' End of for loop

 PORTB = %01010101             ' New binary combination on PORTB

 while 1                       ' endless loop
   PORTB =  not PORTB          ' Invert PORTB logic state
   Delay_ms(200)               ' 200mS delay
 wend
end.                           ' End of program
```

for loop

if statement

while loop





4.4 EXAMPLE 2

Use assembly instructions and internal oscillator LFINTOSC...

This example is actually a sequel to the previous one. It deals with a bit more complicated problem... The idea is to make LEDs on PORTB blink slowly. It can be done by setting delay parameter to be large in the Delay function. But there is also another, more efficient way to do it. You remember that this microcontroller has a built-in oscillator LFINTOSC which operates at the frequency of 31kHz? Now, it’s time to give it a try.

The program starts with the do-until loop and remains herein for 20 cycles. After each iteration, 100mS delay is provided, which is reflected as a relatively fast PORTB LEDs blinking. When the program exits this loop, the microcontroller starts using the LFINTOSC oscillator as a clock signal source. LEDs blink much slower now, even though the program executes the same do-while loop with 10 times shorter delay.

To demonstrate one potentionally dangerous situation here, control bits are activated by assembly instructions. Simply put, when entering or exiting an assembly sequence in the program, the compiler doesn't save data on the currently active RAM bank, which means that in this program section, bank selection depends on SFR registers in use. When switching back to the program section written in Basic, the control bits RP0 and RP1 must return the state they had before entering the assembly sequence. In this case, the saveBank auxiliary variable is used to save the state of these two bits.

```
'Header ****************************************
program example_2     ' Program name
dim k as byte         ' Variable k is of byte type
dim saveBank as byte  ' Variable saveBank is of byte type
main:                 ' Start of program
k = 0                 ' Initial value of variable k
ANSEL = 0             ' All I/O pins are configured as digital
ANSELH = 0
PORTB = 0             ' All PORTB pins are set to 0
TRISB = 0             ' PORTB pins are configured as outputs
do
PORTB = not PORTB     ' Invert PORTB logic state
Delay_ms(100)         ' 100mS delay
k = k+1               ' Increment k by 1
loop until k=20       ' Remain in loop while k<20
k=0                   ' Reset variable k
saveBank = STATUS and %01100000 ' Save the state of bits RP0 and RP1
                      ' (bits 5 and 6 of the STATUS register)
asm                   ' Start of assembly sequence
bsf STATUS,RP0        ' Select memory bank containing
bcf STATUS,RP1        ' the OSCCON register
bcf OSCCON,6          ' Select internal oscillator LFINTOSC
bcf OSCCON,5          ' with a frequency of 31KHz
bcf OSCCON,4
bsf OSCCON,0          ' Microcontroller uses internal oscillator
end asm               ' End of assembly sequence
STATUS = STATUS and %10011111 ' Bits RP0 and RP1 return their original state
STATUS = STATUS or saveBank
do
PORTB = not PORTB     ' Invert PORTB logic state
Delay_ms(10)          ' 10 mS delay
k = k+1               ' Increment k by 1
loop until k=20       ' Remain in loop while k<20
stay_here: goto stay_here ' Endless loop
end.
```

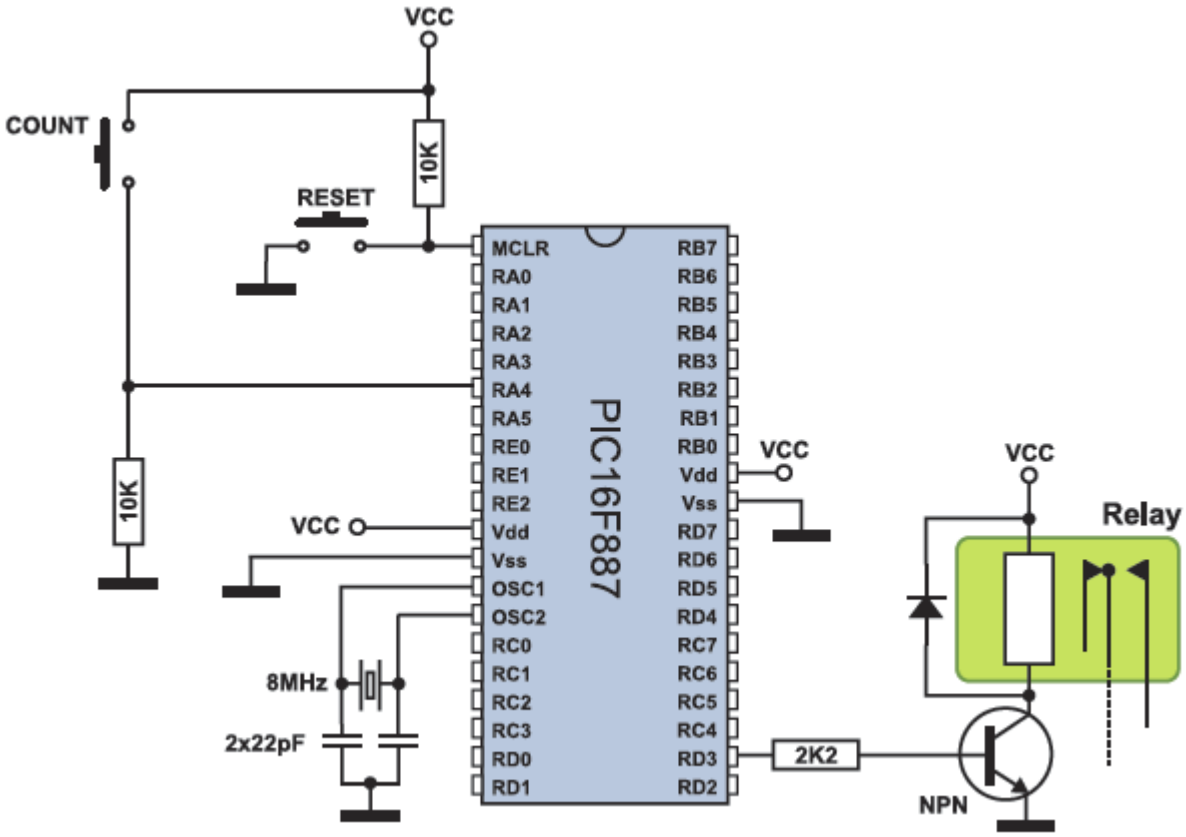
You have noticed that the clock signal source is changed on the fly. If you want to make sure of it, remove quartz crystal prior to switching the microcontroller on. The microcontroller will not start to operate because the Config Word loaded with the program requires the quartz crystal to be provided. If you remove this crystal later on during an operation, nothing will happen, that is to say it will not affect the operation of the microcontroller at all.

4.5 EXAMPLE 3

TMR0 as a counter, declare new variables, use symbols, use a relay ...

In the previous two examples the microcontroller executes the program without being affected by its surrounding. Practically, microcontroller-based devices operating in this manner are very rare (for example, a simple neon sign controller). Input pins are also used in this example. The schematic is given in figure below, while the program is on the next page. It's still very simple. Timer TMR0 is used as a counter. The counter input is connected to a push button in such a way that any button pressure causes the TMR0 timer to count one pulse. When the number of pulses matches the number stored in the TEST register, a logic one (5V) will appear on the PORTD.3 pin. This voltage is used to activate an electromechanical relay, and this bit is therefore called 'RELAY' in the program.

Here, the TEST register stores number 5. Of course, it can be any number obtained either by computing or defined as a constant. Besides, the microcontroller can run some other device instead of relay, while a sensor can be used instead of the push button. This example illustrates one of the most common applications of the microcontroller in the industry; when something is performed as many times as required, then something else should be turned on or off....




```
' Header*****************************************************
program example_3       ' Program name
symbol RELAY = PORTD.3  ' Pin PORTD.3 is named RELAY
dim TEST as byte        ' Variable TEST is of byte type
main:                   ' Start of program
TEST = 5                ' Constant TEST = 5
ANSEL = 0               ' All I/O pins are configured as digital
ANSELH = 0
PORTA = 0               ' Reset PORTA
TRISA = 0xFF            ' All portA pins are configured as inputs
```

```
PORTD = 0               ' Reset PORTD
TRISD = %11110111       ' Pin RD3 is configured as an output, while other pins are
                        ' configured as inputs
OPTION_REG.5 = 1        ' Counter TMR0 receives pulses through the RA4 pin
OPTION_REG.3 = 1        ' Prescaler rate is 1:1
TMR0 = 0                ' Reset timer/counter TMR0
while 1
if TMR0 = TEST then     ' Does the number in timer match constant TEST?
        RELAY = 1       ' Numbers match. Set the RD3 bit (output RELAY)
end if
wend                    ' Remain in endless loop
end.                    ' End of program
```

Only one symbol (RELAY) is used here. It is assigned the third pin of PORTD in declaration.

```
symbol RELAY = PORTD.3 ' Symbol RELAY = PORTD.3
```

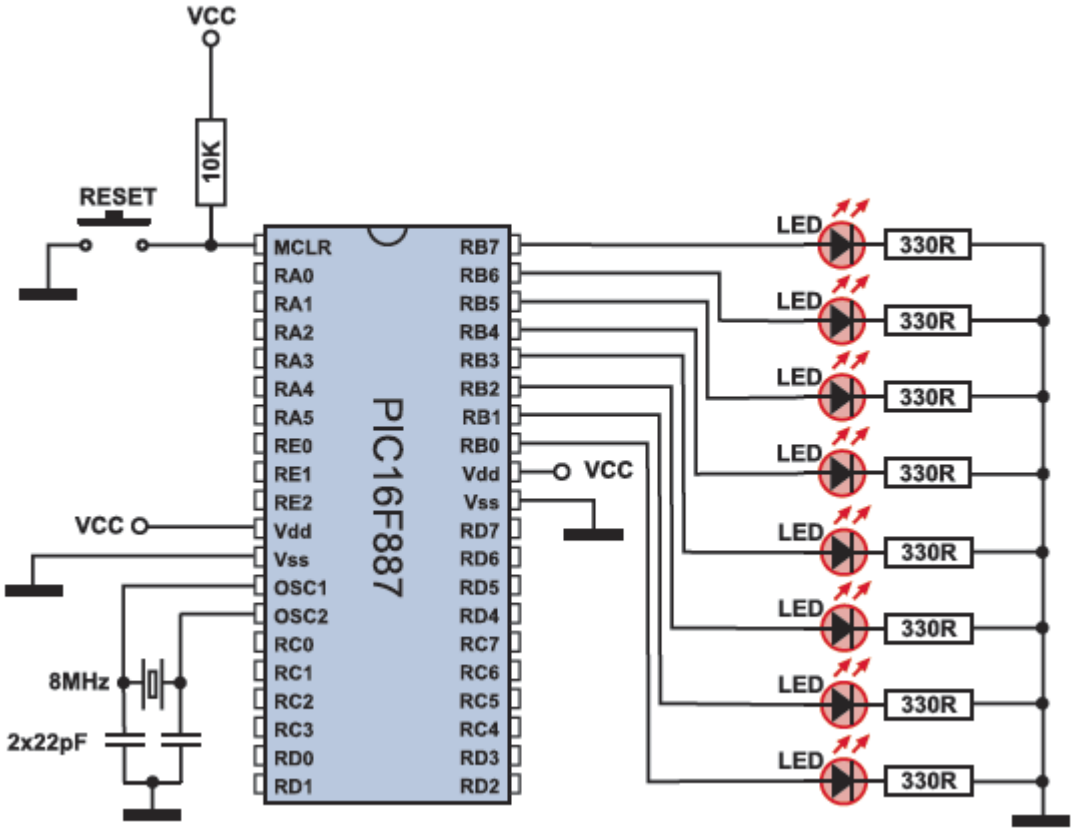
If several port D pins are connected to relays, the expression above could be written in this way as well:

4.6 EXAMPLE 4

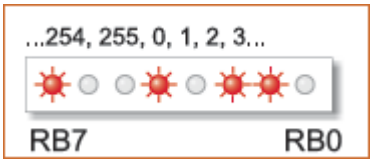
Use Timer0, Timer1 and Timer2. Use interrupts, declare new procedure...

If you have read previous examples, you have probably noticed a disadvantage of using time delays. In all those cases, the microcontroller is ‘captive’ and does nothing. It simply waits for some time to pass. Such waste of time is often an unacceptable luxury and some other method should be employed here.

Do you remember the story about timers? Interrupts? This example makes connection between them in a practical way. The schematic is still the same and challenge as well. It is necessary to provide a time delay long enough to notice changes on a port. Timer TMR0 with assigned prescaler is used for this purpose. An interrupt is generated on every timer register overflow and every interrupt routine increments the cnt variable by 1. When it reaches 50, the PORTB is incremented by 1. The whole procedure is performed ‘behind the scenes’, which enables the microcontroller to do something else.

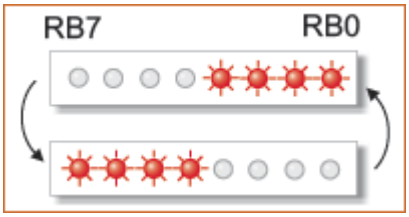



```
' Header*****************************************************
program example_4a ' Start of program
dim cnt as byte    ' Define variable cnt as byte
sub procedure interrupt ' This subprocedure determines what should
                        ' be done when an interrupt is generated
cnt = cnt + 1           ' Interrupt causes cnt to be incremented by 1
TMR0 = 96               ' Timer TMR0 is returned its initial value
INTCON = 0x20           ' Bit T0IE is set, bit T0IF is cleared
end sub                 ' End of interrupt routine
main:                   ' Start of program
OPTION_REG = 0x84       ' Prescaler is assigned to timer TMR0
ANSEL = 0               ' All I/O pins are configured as digital
ANSELH = 0
TRISB = 0               ' All PORTB pins are configured as outputs
PORTB = 0x0             ' Reset PORTB
TMR0 = 96               ' Timer T0 counts from 96 to 255
INTCON = 0xA0           ' Enable interrupt TMR0
cnt = 0                 ' Variable cnt is assigned a 0
while 1                 ' Endless loop
if cnt = 50 then        ' Increment PORTB after 50 interrupts
PORTB = PORTB + 1       ' Increment number on PORTB by 1
cnt = 0                 ' Reset variable cnt
end if
wend
end. ' End of program
```



An interrupt is generated on every timer register TMR0 overflow.

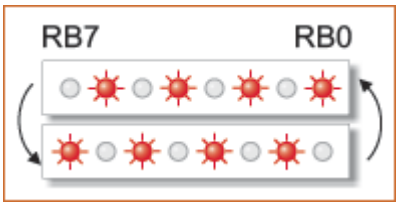
```
'Header*****************************************************
program example_4b       ' Program name
dim cnt as byte          ' Define variable cnt
sub procedure interrupt  ' Define interrupt subprocedure
cnt = cnt+1              ' Interrupt causes cnt to be incremented by 1
PIR1.TMR1IF = 0          ' Reset bit TMR1IF
TMR1H = 0x80             ' TMR1H and TMR1L timer registers are returned
TMR1L = 0x00             ' their initial values
end sub                  ' End of interrupt routine
main:                    ' Start of program
ANSEL = 0                ' All I/O pins are configured as digital
ANSELH = 0
PORTB = 0xF0             ' Initial value of PORTB bits
TRISB = 0                ' PORTB pins are configured as outputs
T1CON = 1                ' Set timer TMR1
PIR1.TMR1IF = 0          ' Reset bit TMR1IF
TMR1H = 0x80             ' Set initial value for timer TMR1
TMR1L = 0x00
PIE1.TMR1IE = 1          ' Enable interrupt on overflow
cnt = 0                  ' Reset variable cnt
INTCON = 0xC0            ' Enable interrupt (bits GIE and PEIE)
while 1                  ' Endless loop
if cnt = 76 then         ' Change PORTB state after 76 interrupts
PORTB = not PORTB        ' Number in PORTB is inverted
cnt = 0                  ' Reset variable cnt
end if
wend
end.                     ' End of program
```



In this case, an interrupt is enabled after the timer register TMR1 (TMR1H and TMR1L) overflow occurs. The combination of bits changing on PORTB is different from that in the previous example.

```
'Header*****************************************************
program example_4c       ' Program name
dim cnt as byte          ' Define variable cnt as byte
sub procedure Replace    ' Define procedure Replace
PORTB = not PORTB        ' Define new procedure 'Replace'
end sub                  ' Procedure inverts port state
sub procedure interrupt  ' Define interrupt subprocedure
if PIR1.TMR2IF then      ' If bit TMR2IF = 1,
cnt = cnt +1             ' Increment variable cnt by 1
PIR1.TMR2IF = 0          ' Reset bit and
TMR2 = 0                 ' reset register TMR2
end if
end sub                  ' End of interrupt routine
main:                    ' Start of program
cnt = 0                  ' Reset variable cnt
ANSEL = 0                ' All I/O pins are configured as digital
ANSELH = 0
PORTB = %10101010        ' Logic state on PORTB pins
TRISB = 0                ' All PORTB pins are configured as outputs
T2CON = 0xFF             ' Set timer T2
TMR2 = 0                 ' Initial value of timer register TMR2
PIE1.TMR2IE = 1          ' Enable interrupt
```

```
INTCON = 0xC0           ' Set bits GIE and PEIE
while 1                 ' Endless loop
  if cnt > 30 then      ' Change PORTB after more than 30 interrupts
    Replace             ' Function Replace inverts the PORTB state
    cnt = 0             ' Reset variable cnt
  end if
wend
end.                    ' End of program
```

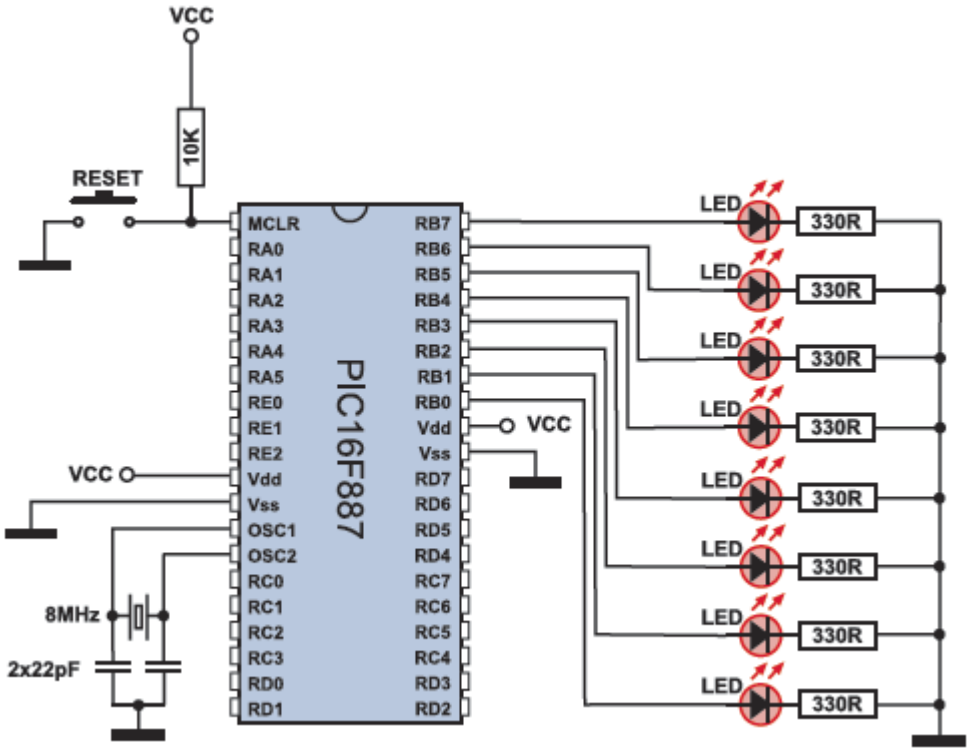


This time, an interrupt is generated after timer register TMR2 overflow occurs. The Replace procedure, which normally doesn't belong to mikroBasic, is used in this example to invert port pins state.

4.7 EXAMPLE 5

Use watch-dog timer

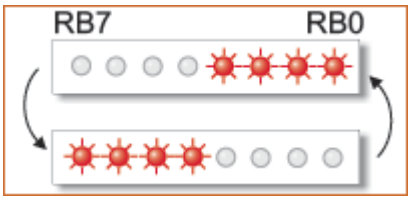
This example illustrates how the watch-dog timer should not be used. A command used to reset this timer is intentionally left out in the main program loop, thus enabling it to win the time battle and cause the microcontroller to be reset. As a result, the microcontroller will be reset every time, which is indicated by PORTB LEDs blinking.




```
'Header************************************************
program example_5 ' Program name
main:             ' Start of program
```

```
OPTION_REG = 0x0E ' Prescaler is assigned to timer WDT (1:64)
asm CLRWDT        ' Assembly command to reset WDT timer
end asm
PORTB = 0x0F      ' Initial value of the PORTB register
TRISB = 0         ' All PORTB pins are configured as outputs
Delay_ms(300)     ' 30mS delay
PORTB = 0xF0      ' Port PORTB value different from initial
while 1           ' Endless loop. Program remains here until WDT
wend              ' timer resets the microcontroller
end.              ' End of program
```

In order to make this example work properly, it is necessary to enable the watchdog timer by selecting the **Watchdog Timer - Enabled** option in **mE programmer**.



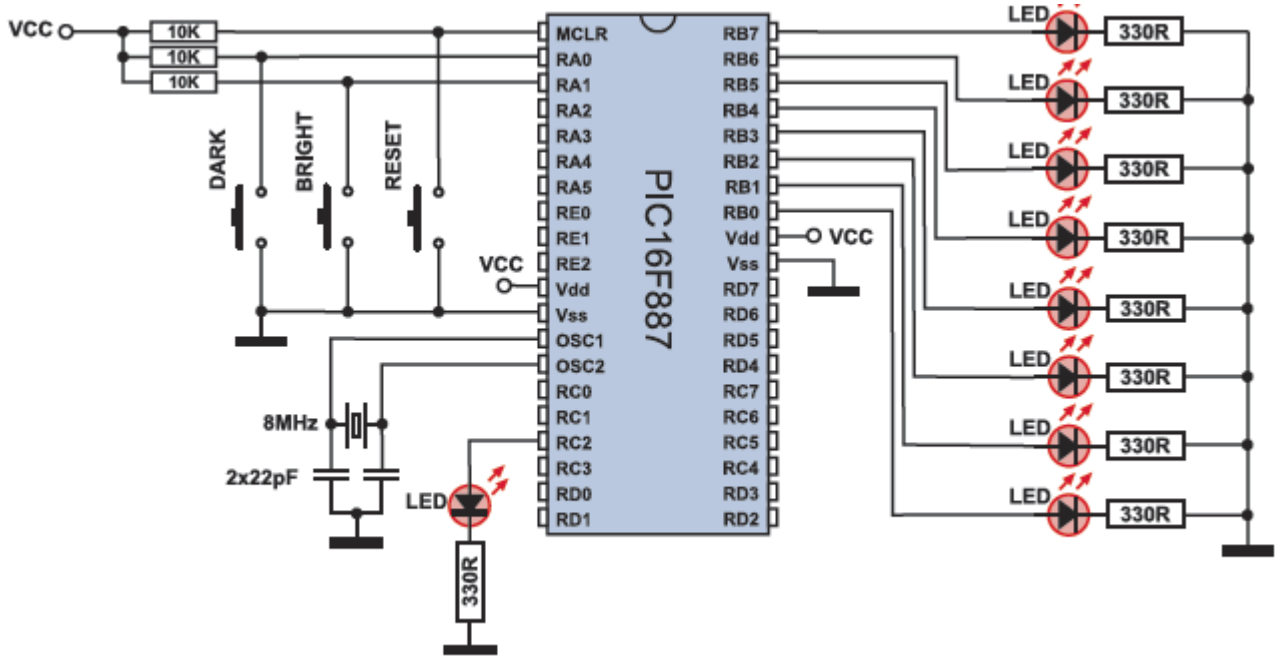
4.8 EXAMPLE 6

Module CCP1 as a PWM signal generator

This example illustrates the use of CCP1 module in PWM mode. To make it more interesting, the width of P1A output pulses (PORTC,2) may be changed using pushbuttons symbolically marked as ‘DARK’ and ‘BRIGHT’, while the set width is visible as a binary combination on PORTB. The operation of this module is under control of the procedures belonging to the specialized PWM Library. Three of them are used here:

1. **PWM1_init** has the prototype: `sub procedure PWM1_Init(const freq as longint)`
   Parameter freq sets the frequency of PWM signal expressed in herz. In this example it is 5 KHz.
2. **PWM1_Start** has the prototype: `sub procedure PWM1_Start()`
3. **PWM1_Set_Duty** has the prototype: `sub procedure PWM1_Set_Duty(dim duty_ratio as byte)`
   Parameter *duty_ratio* sets pulse duration in a pulse sequence.

The PWM library also contains the PWM_Stop procedure used to disable this mode. Its prototype is: `sub procedure PWM1_Stop()`




```
' Header ****************************************************
program example_6 ' Program name
dim current_duty, old_duty, oldstate as byte ' Define variables current_duty
                                             ' old_duty and oldstate
main:             ' Start of program
ANSEL = 0         ' All I/O pins are configured as digital
ANSELH = 0
PORTA = 255       ' PORTA initial state
TRISA = 255       ' All PORTA pins are configured as inputs
PORTB = 0         ' Initial state of PORTB
TRISB = 0         ' All PORTB pins are configured as outputs
PORTC = 0         ' PORTC initial state
TRISC = 0         ' All PORTC pins are configured as outputs
PWM1_Init(5000)   ' PWM module initialization (5 KHz)
current_duty = 16 ' Initial value of variable current_duty
old_duty = 0      ' Reset variable old_duty
PWM1_Start()      ' Start PWM1 module
while 1           ' Endless loop
  if oldstate and Button(PORTA, 0,1,1) then ' If the button connected to RA0 is pressed
    current_duty = current_duty + 1         ' increment variable current_duty
    if Button(PORTA, 0, 1, 1) then
      oldstate = 255
    end if
  end if

  if oldstate and Button(PORTA, 1,1,1) then ' If the button connected to RA1 is pressed
    current_duty = current_duty - 1         ' decrement value current_duty
    if Button(PORTA, 1, 1, 1) then
      oldstate = 255
    end if
  end if

  if old_duty <> current_duty then          ' If current_duty and old_duty are not
    PWM1_Set_Duty(current_duty)             ' equal set PWM to a new value,
    old_duty = current_duty                 ' save the new value
    PORTB = old_duty                        ' and show it on PORTB
  end if

  Delay_ms(200)                             ' 200mS delay
wend
```

```
end.                                    ' End of program
```

In order to make this example work properly, it is necessary to check the following libraries in the Library Manager prior to compiling:

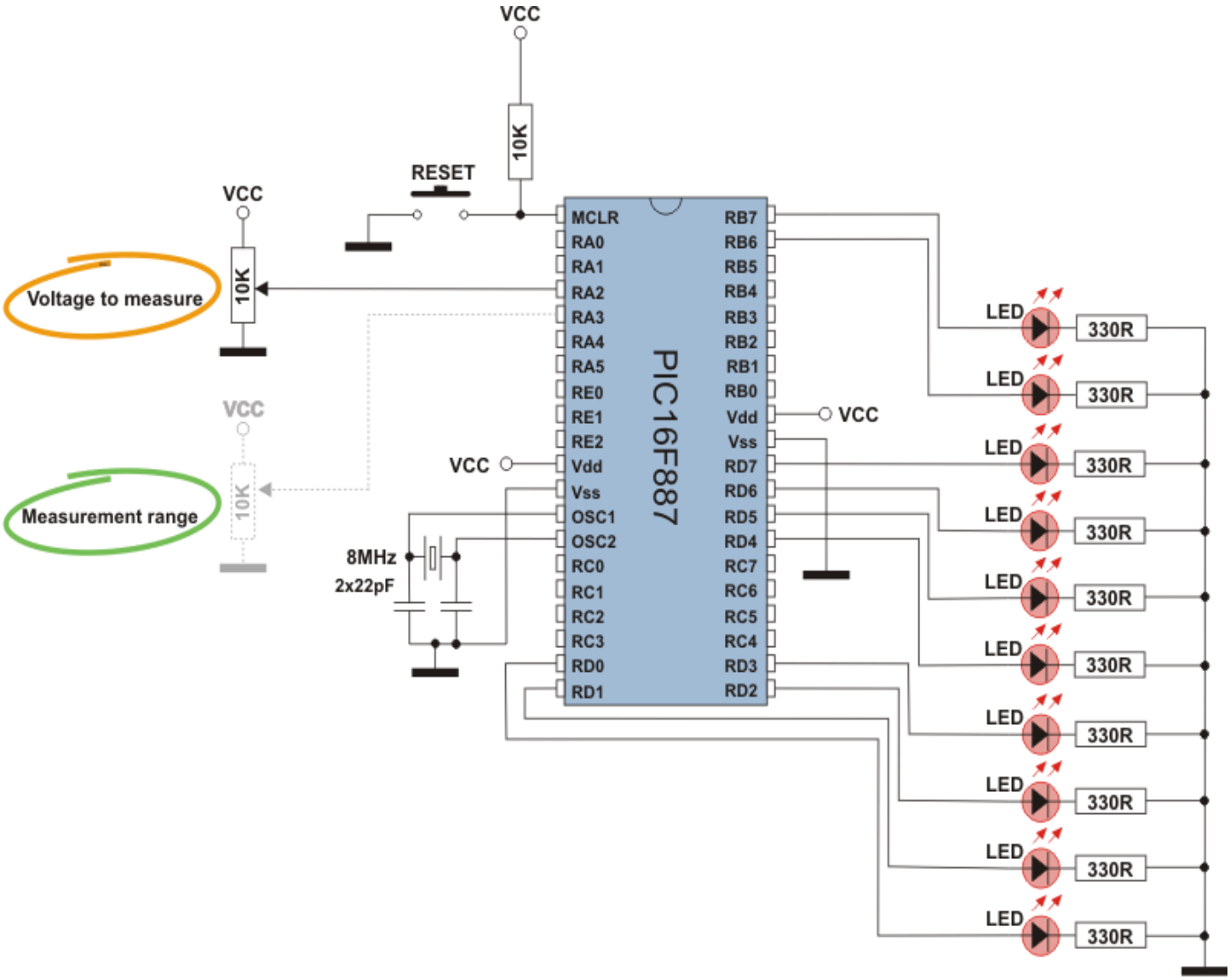
- PWM
- Button

4.9 EXAMPLE 7

Use A/D converter

An A/D converter provided on the PIC16F887 is used in this example. Is it necessary to say that everything is simple?! A variable analog signal is applied to the AN2 pin, while the 10- bit result of conversion is shown on ports PORTB and PORD (8 LSBs on PORTD and 2 MSBs on PORTB). GND is used as a negative voltage reference Vref-, while the VCC is used as a positive voltage reference.

If you use varriable voltage as Vref+ (refer to dashed part of the schematic) you'll be able to ‘stretch and shrink’ the range of voltage measurement.



In other words, the A/D converter always generates a 10-bit binary number, which means that it detects 1024 voltage levels (210=1024) in total. The difference between two voltage levels is not always the same. The less the difference between Vref+ and Vref-, the less the difference between two of 1024 levels. As you can see, the A/D converter is able to detect slight changes in voltage.

```
'Header**************************************************
program example_7    ' Program name
dim temp_res as word ' Variable temp_res is of word type
main:                ' Start of program
ANSEL = 0x0C         ' Pin AN2 is configured as analog
TRISA = 0xFF         ' All PORTA pins are configured as inputs
ANSELH = 0           ' Other pins are configured as digital
TRISB = 0x3F         ' PORTB pins RB7 and RB6 are configured as
                     ' outputs
TRISD = 0            ' All PORTD pins are configured as outputs
ADCON1.B4 = 0        ' Positive voltage reference is VCC.
while 1              ' Endless loop
  temp_res = ADC_Read(2) ' Result of A/D conversion is copied to temp_res
  PORTD = temp_res       ' 8 LSBs are moved to PORTD
  PORTB = temp_res >> 2  ' 2 MSBs are moved to bits RB6 and RB7
wend
end. ' End of program
```

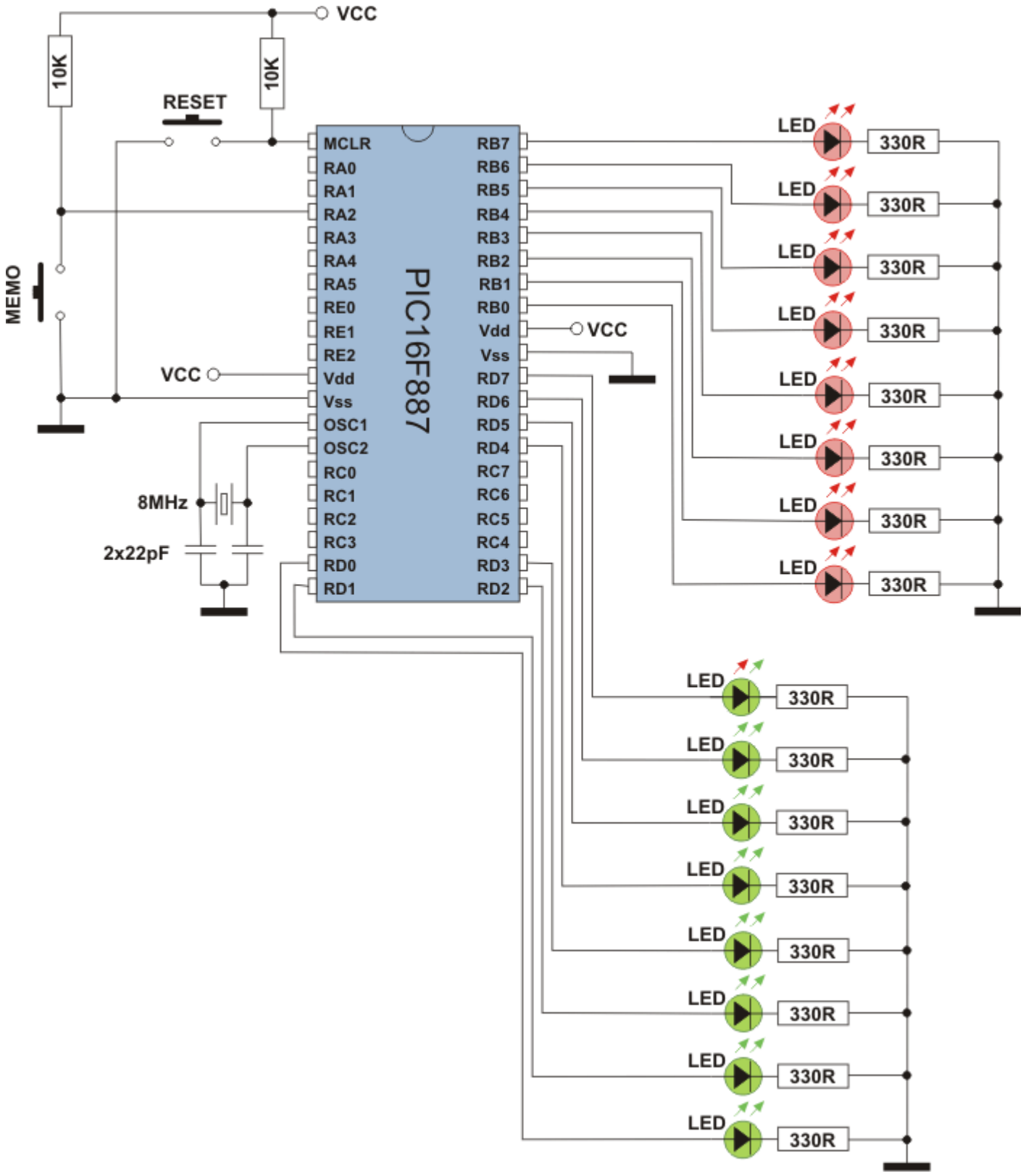
In order to make this example work properly, it is necessary to check the ADC library in the Library Manager prior to compiling:

- ADC

4.10 EXAMPLE 8

Use EEPROM Memory

This example illustrates write and read from built-in EEPROM memory. The program works as follows. The main loop first reads EEPROM memory location at address 5. The program then enters an endless loop in which PORTB is incremented and the state of PORTA.2 input is checked. At the moment of pressing the push button marked MEMO, a number stored in PORTB will be saved in EEPROM at address 5 and directly read from it and shown on PORTD in binary format.




```
'Header*****************************************************
program example_8       ' Program name
main:                   ' Start of program
ANSEL = 0               ' All I/O pins are configured as digital
ANSELH = 0
PORTB = 0               ' PORTB initial value
TRISB = 0               ' All PORTB pins are configured as outputs
PORTD = 0               ' PORTB initial value
TRISD = 0               ' All PORTD pins are configured as outputs
TRISA = 0xFF            ' All PORTA pins are configured as inputs
```

```
PORTD = EEPROM_Read(5) ' Read EEPROM memory at address 5
while 1                ' Endless loop
  PORTB = PORTB + 1      ' Increment PORTB by 1
  Delay_ms(100)          ' 100mS delay
  while not PORTA.B2     ' Remain in this loop as long as the button is pressed
    if not PORTA.B2 then
      EEPROM_Write(5,PORTB) ' If MEMO is pressed, save PORTB
      PORTD = EEPROM_Read(5) ' Read written data
    end if
  wend
wend
end. ' End of program
```

In order to make this example work properly, it is necessary to check the EEPROM library in the Library Manager prior to compiling:

- EEPROM

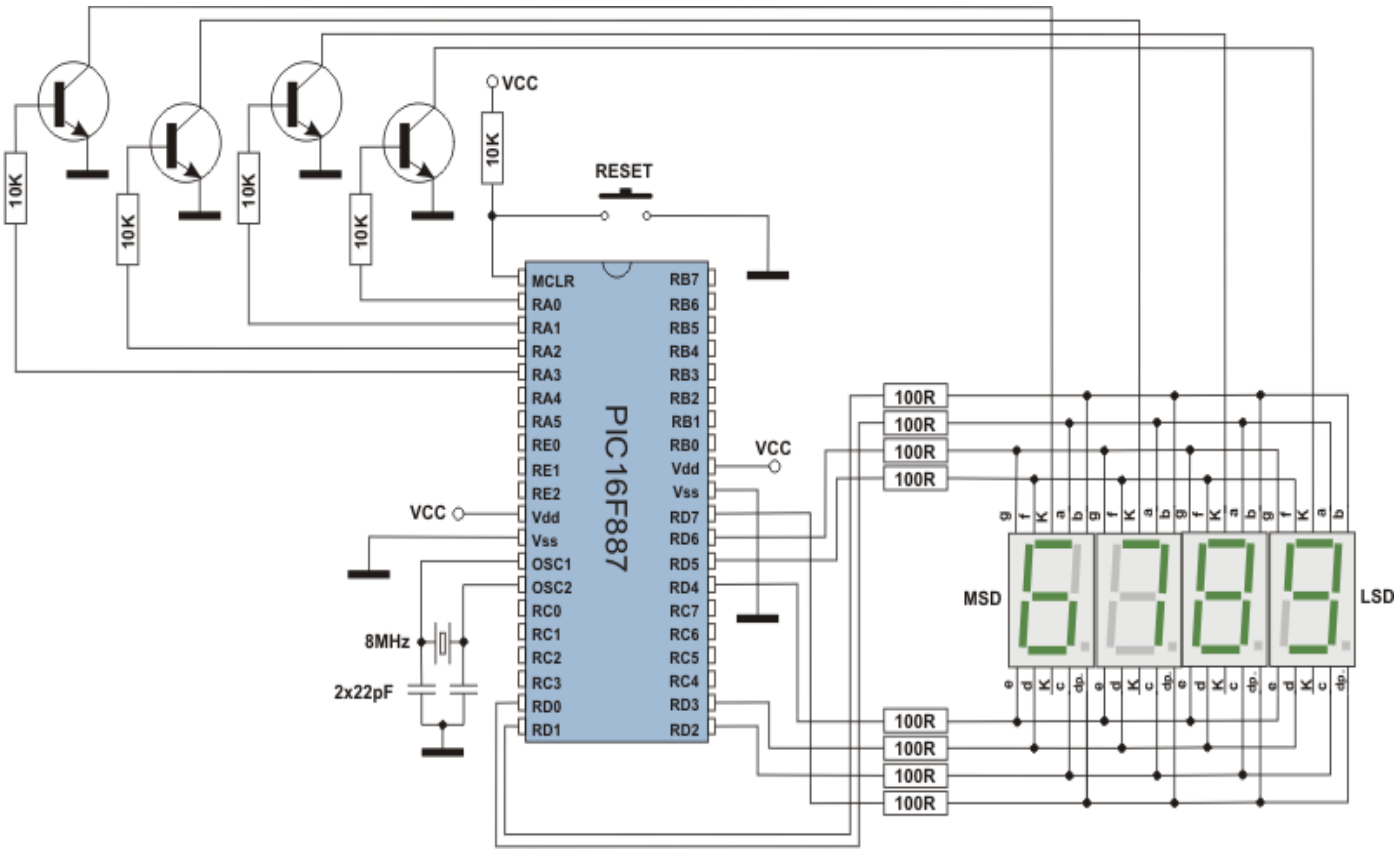
In order to check whether this program works properly, it is sufficient to press the MEMO push button and then turn off the microcontroller. After restarting it, the program will display on PORTD the value stored in EEPROM memory at addrsess 5. Remember that at the moment of writing, this value was displayed on PORTB.

4.11 EXAMPLE 9

Four-digit LED counter, multiplexing

The microcontroller operates as a four-digit counter here. Variable i is incremented (slow enough to be noticed) and its value is displayed on a four-digit LED display (9999-0). The objective is to convert a binary number into decimal and split it in four digits (thousands, hundreds, tens and ones). Since the LED display segments are connected in parallel, it is necessary to ensure that they change fast enough to make impression of simultaneous light emission (time-division multiplexing).

In this example, timer TMR0 is in charge of the time-multiplexing, while the mask function is used to convert a binary number into decimal.




```
'Header*****************************************************
program example_9                  ' Program name
dim shifter, portd_index as byte ' Variables shifter and portd_index are of byte type
digit, number as word              ' Variables digit and number are of word type
portd_array as word[4]             ' Array portd_array has 4 members of word type
sub function mask (dim num as Word) as Word ' Subroutine for masking
  select case num ' used to convert binary
    case 0 result = $3F            ' numbers into appropriate
    case 1 result = $06            ' combination of bits to be
    case 2 result = $5B            ' displayed on LED display
    case 3 result = $4F
    case 4 result = $66
    case 5 result = $6D
    case 6 result = $7D
    case 7 result = $07
    case 8 result = $7F
    case 9 result = $6F
  end select ' Case end
end sub ' End of subroutine
sub procedure interrupt ' Start of interrupt routine
  PORTA = 0              ' Turn off all 7-segment displays
  PORTD = portd_array [portd_index] ' Send appropriate value to PORTD
  PORTA = shifter        ' Turn on appropriate 7-segment display
  shifter = shifter << 1 ' Move shifter to the next digit

  if (shifter > 8) then
    shifter = 1
  end if

  Inc(portd_index)       ' Increment portd_index

  if (portd_index > 3) then
    portd_index = 0      ' Turn on 1st, turn off 4th 7segment display
  end if
```

```
  TMR0 = 0                ' Reset TIMER0 value
  T0IF_bit = 0            ' Clear Timer0 interrupt flag
end sub                   ' End of interrupt routine
main:             ' Start of program
ANSEL = 0         ' Configure analog pins as digital I/O
ANSELH = 0
OPTION_REG = $80 ' Timer0 settings (Timer0 work as timer with prescaler)
digit = 0         ' Initial value of variable digit
portd_index = 0   ' Turn on 1st LED display
shifter = 1       ' Initial value of variable shifter
TMR0 = 0          ' Clear Timer0
INTCON = $A0      ' Enable interrupt with GIE and T0IE bits
PORTA = 0         ' Clear PORTA
TRISA = 0         ' Set PORTA as output
PORTD = 0         ' Clear PORTD
TRISD = 0         ' Set PORTD as output
number = 6789     ' Some initial value on LED display
while TRUE        ' Endless loop
  digit = number / 1000         ' Extract thousands
  portd_array[3] = mask(digit)  ' and store it to PORTD array
  digit = (number / 100) mod 10 ' Extract hundreds
  portd_array[2] = mask(digit)  ' and store it to PORTD array
  digit = (number / 10) mod 10  ' Extract tens
  portd_array[1] = mask(digit)  ' and store it to PORTD array
  digit = number mod 10         ' Extract ones
  portd_array[0] = mask(digit)  ' and store it to PORTD array

  Delay_ms(1000)                ' One second delay
  Inc(number)                   ' Increment number
  if (number > 9999) then       ' Start to count from zero
    number = 0
  end if
wend
end. ' End of program
```

4.12 EXAMPLE 10

Use LCD display

This example illustrates the use of an alphanumeric LCD display. The function libraries make this program simpler.

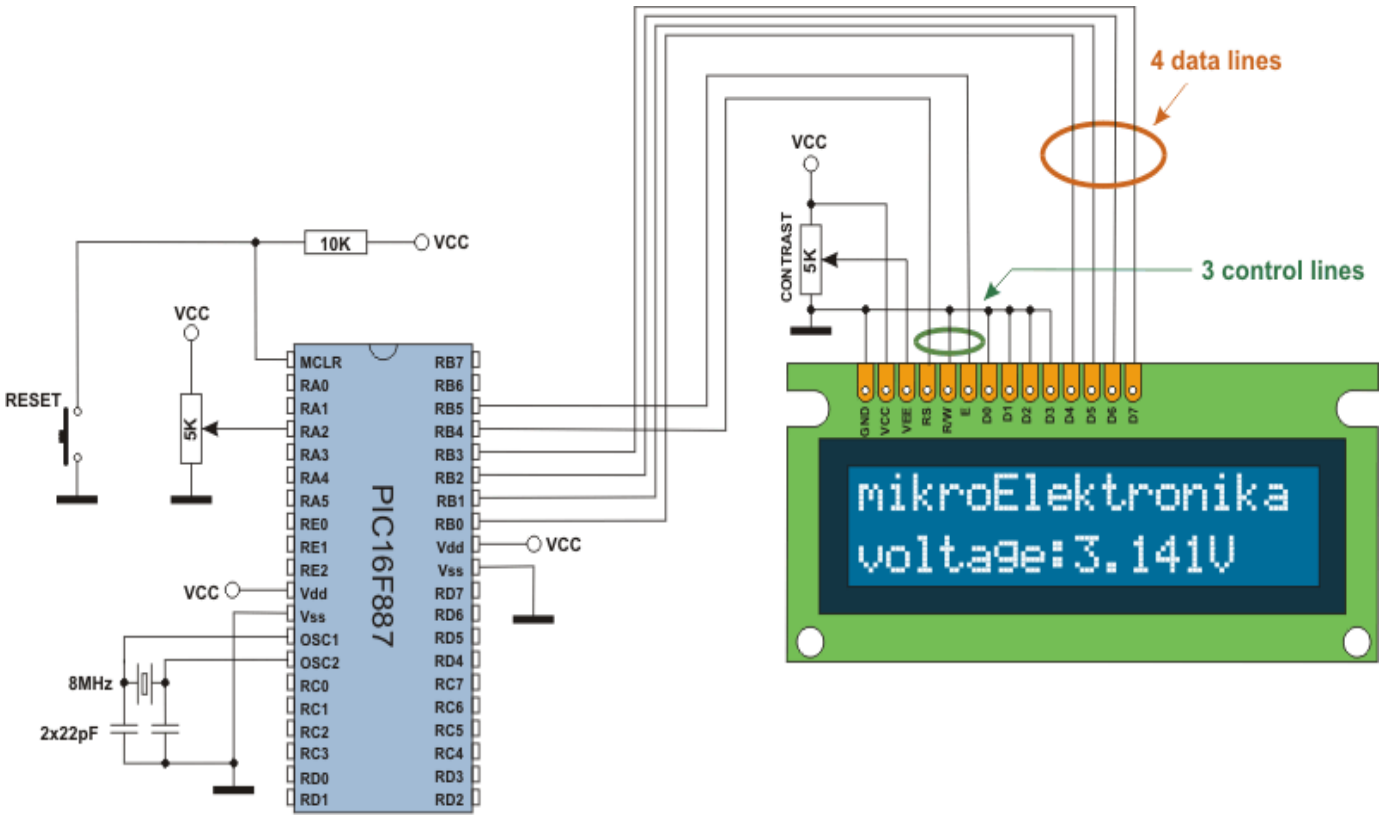
Two messages written in two lines appear on the display:

mikroElektronika
LCD example

Two seconds later, the second message is replaced with the voltage present on the A/D converter input (the RA2 pin). For example:

mikroElektronika
voltage:3.141V

Anyway, the current temperature or some other measured value can be displayed instead of voltage.



In order to make this example work properly, it is necessary to check the following libraries in the Library Manager prior to compiling:

- ADC
- LCD

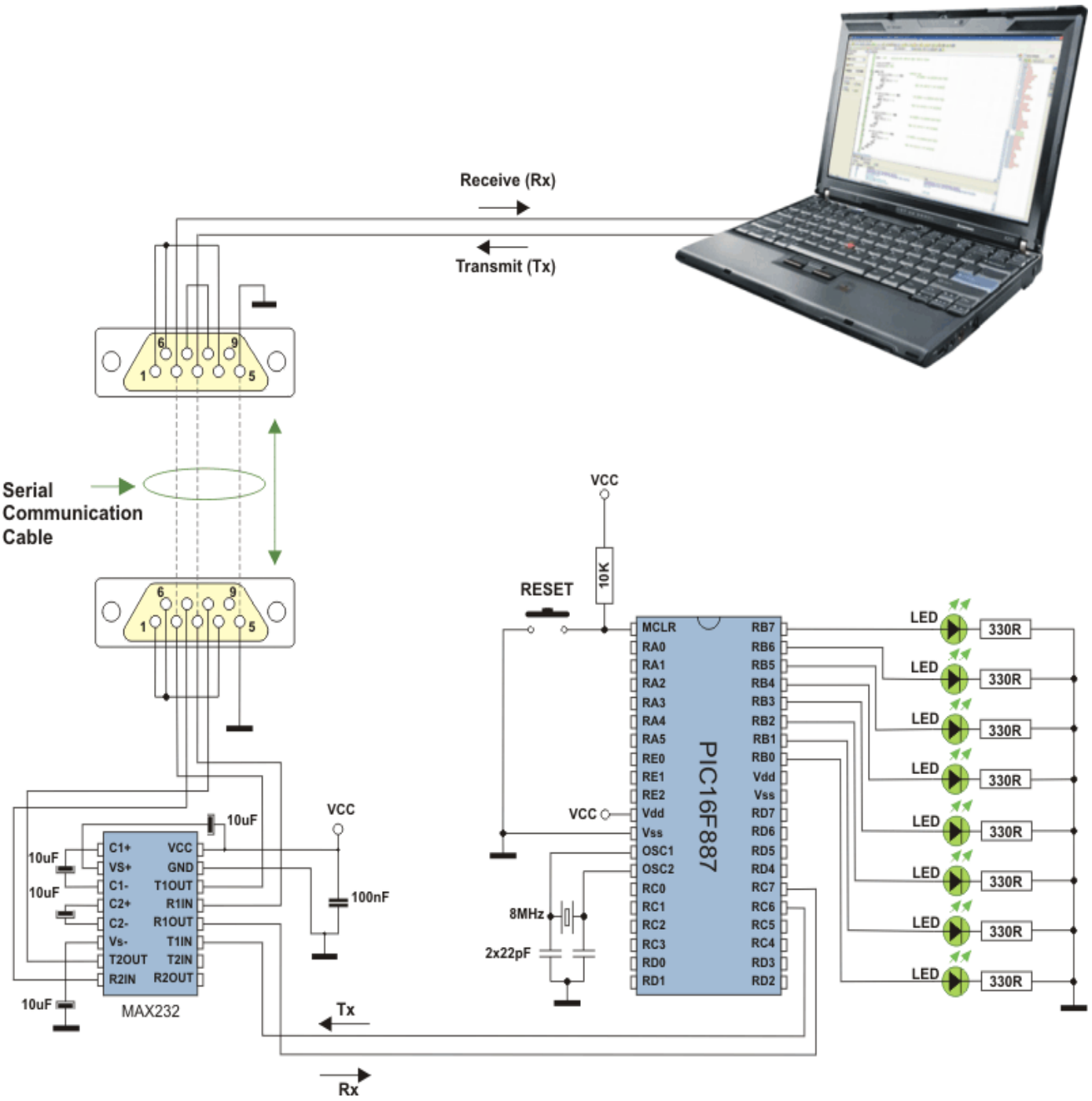
```
'Header*****************************************************
program example_10 ' Program name
dim LCD_RS as sbit at RB4_bit ' Lcd module connections
LCD_EN as sbit at RB5_bit
LCD_D4 as sbit at RB0_bit
LCD_D5 as sbit at RB1_bit
LCD_D6 as sbit at RB2_bit
LCD_D7 as sbit at RB3_bit
LCD_RS_Direction as sbit at TRISB4_bit
LCD_EN_Direction as sbit at TRISB5_bit
LCD_D4_Direction as sbit at TRISB0_bit
LCD_D5_Direction as sbit at TRISB1_bit
LCD_D6_Direction as sbit at TRISB2_bit
LCD_D7_Direction as sbit at TRISB3_bit ' End Lcd module connections
dim text as string [16]   ' Variable text is of string type
dim ch, adc_rd as word    ' Variables ch and adc_rd are of word type
dim tlong as longword     ' Variable tlong is of longword type
main:                     ' Start of program
TRISB = 0                 ' All port PORTB pins are configured as outputs
PORTB = 0xFF
INTCON = 0                ' All interrupts disabled
ANSEL = 0x04              ' Pin RA2 is configured as an analog input
TRISA = 0x04
ANSELH = 0                ' Rest of pins is configured as digital
Lcd_Init()                ' LCD display initialization
Lcd_Cmd(_LCD_CURSOR_OFF)  ' LCD command (cursor off)
Lcd_Cmd(_LCD_CLEAR)       ' LCD command (clear LCD)
text = "mikroElektronika" ' Define the first message
```

```
Lcd_Out(1,1,text)          ' Write the first message in the first line
text = "LCD example"       ' Define the second message
Lcd_Out(2,1,text)          ' Write the second message in the second line
ADCON1 = 0x80              ' A/D voltage reference is VCC
TRISA = 0xFF               ' All PORTA pins are configured as inputs
Delay_ms(2000)
text = "Voltage="          ' Define the third message
while 1                    ' Endless loop
  adc_rd = ADC_Read(2)     ' A/D conversion. Pin RA2 is an input.
  Lcd_Out(2,1,text)        ' Write result in the second line
  tlong = adc_rd * 5000    ' Convert the result in millivolts
  tlong = tlong / 1023     ' 0..1023 -> 0-5000mV
  ch = (tlong / 1000) mod 10 ' Extract volts (thousands of millivolts)
                             ' from result
  Lcd_Chr(2,9,48+ch)       ' Write result in ASCII format
  Lcd_Chr_CP(".")          ' Write the decimal pint
  ch = (tlong / 100) mod 10 ' Extract hundreds of millivolts
  Lcd_Chr_CP(48+ch)          ' Write result in ASCII format
  ch = (tlong / 10) mod 10  ' Extract tens of millivolts
  Lcd_Chr_CP(48+ch)          ' Write result in ASCII format
  ch = tlong mod 10          ' Extract digits for millivolts
  Lcd_Chr_CP(48+ch)          ' Write result in ASCII format
  Lcd_Chr_CP("V")            ' Write a mark for voltage "V"
  Delay_ms(1)                ' 1mS delay
wend
end. ' End of program
```

4.13 EXAMPLE 11

RS232 serial communication

This example illustrates the use of the microcontroller’s EUSART module. Connection between the microcontroller and a PC is established in compliance with the RS232 communication standard. The program works as follows. Every byte received via serial communication is displayed using LED diodes connected to PORTB and is automatically sent back to the sender thereupon. The easiest way to test the program operation is by using a standard Windows program called Hyper Terminal.




```
' Header*****************************************************
program example_11 ' Program name
dim i as byte      ' Variable is of byte type
main:              ' Start of program
UART1_Init(19200)  ' Initialize USART module
' (8 bit, 19200 baud rate, no parity bit...)
while 1            ' Endless loop
  if UART1_Data_Ready() then ' If data has been received
    i = UART1_Read()         ' read it
    UART1_Write(i)           ' and send it back
  end if
wend
end.               ' End of program
```

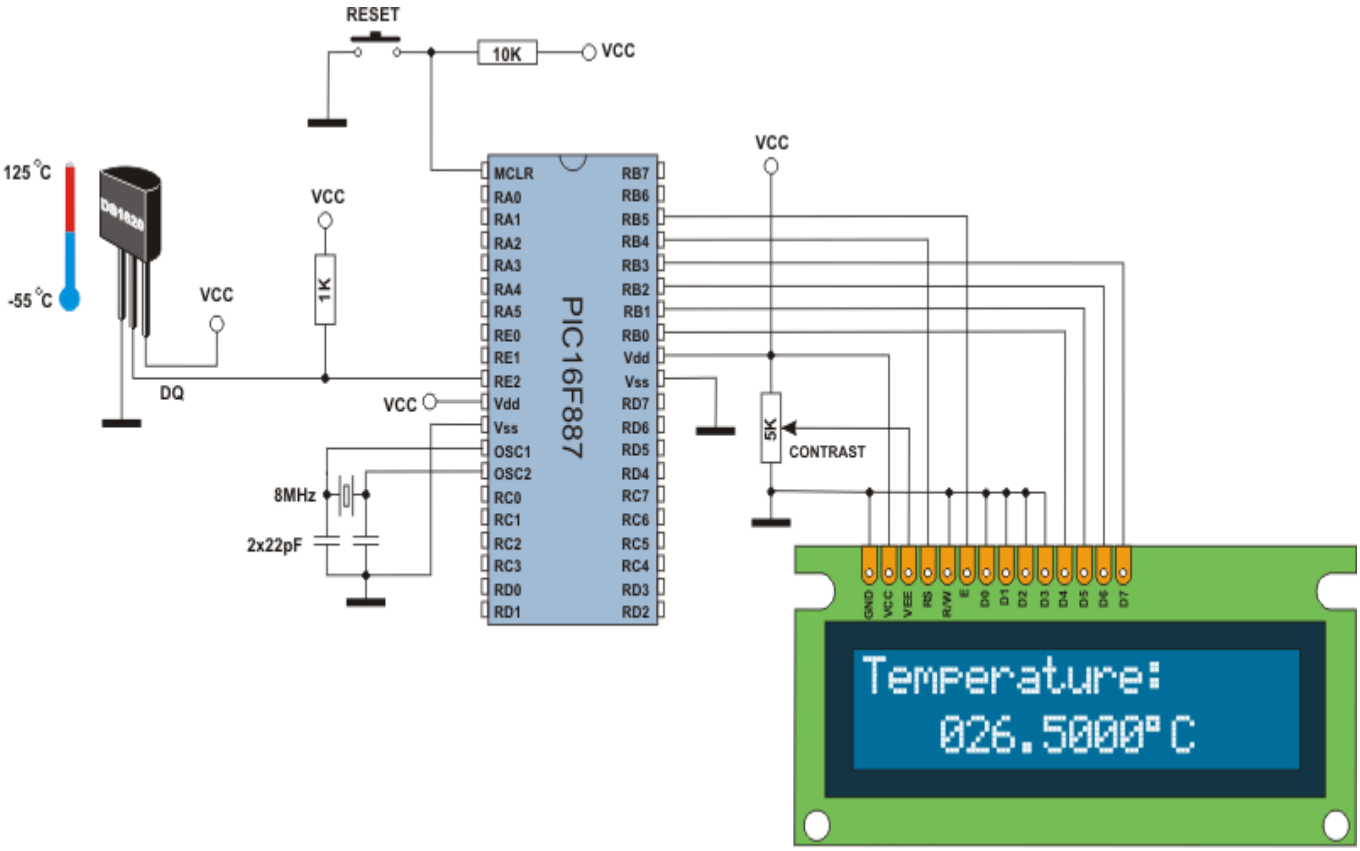
In order to make this example work properly, it is necessary to check the UART library in the Library Manager prior to compiling:

- UART

4.14 EXAMPLE 12

Measure temperature using DS1820 sensor. Use ‘1-wire’ protocol...

Temperature measurement is one of the most common operations performed by the microcontroller. A DS1820 temperature sensor is used here for measuring. It is capable of measuring temperature within the range of -55 °C to 125 °C with a 0.5 °C accuracy. To transfer data to the microcontroller, a special type of serial communication called 1-wire is used. Due to their simple and wide application, such sensors are run and controlled by functions stored in the One_Wire library.



This library contains three functions in total:

- **Ow_Reset** is used to reset sensor;
- **Ow_Read** is used for receiving data from sensor; and
- **Ow_Write** is used for sending commands to sensor.

Here you can see the advantage of using libraries with ready-to-use functions. You obviously don’t need to study documentation provided by the manufacturer so as to use this sensor properly. It is sufficient to copy appropriate functions to the program. If you want to know how any of these functions is declared, just right click on it and select the Help option.

```
' Header***************************************************
program example_12 ' Program name
dim LCD_RS as sbit at RB4_bit ' Lcd module connections
LCD_EN as sbit at RB5_bit
```

```
LCD_D4 as sbit at RB0_bit
LCD_D5 as sbit at RB1_bit
LCD_D6 as sbit at RB2_bit
LCD_D7 as sbit at RB3_bit
LCD_RS_Direction as sbit at TRISB4_bit
LCD_EN_Direction as sbit at TRISB5_bit
LCD_D4_Direction as sbit at TRISB0_bit
LCD_D5_Direction as sbit at TRISB1_bit
LCD_D6_Direction as sbit at TRISB2_bit
LCD_D7_Direction as sbit at TRISB3_bit ' End Lcd module connections
' Set TEMP_RESOLUTION to the corresponding resolution of the DS18x20 sensor in use:
' 18S20: 9 (default setting can be 9,10,11 or 12); 18B20: 12
const TEMP_RESOLUTION as byte = 9 ' Constant TEMP_RESOLUTION is of byte type
dim text as char[9]               ' Variable text is of char type
temp as word                      ' Variable temp is of word type
sub procedure Display_Temperature( dim temp2write as word )
  const RES_SHIFT = TEMP_RESOLUTION - 8
  dim temp_whole as byte          ' Variable temp_whole is of byte type
  temp_fraction as word           ' Variable temp_fraction is of word type
  text = "000.0000"
  if (temp2write and 0x8000) then ' Check if temperature is negative
    text[0] = "-"
    temp2write = not temp2write + 1
  end if
  temp_whole = word(temp2write >> RES_SHIFT) ' Extract temp_whole

  if ( temp_whole div 100 ) then ' Convert temp_whole to characters
    text[0] = temp_whole div 100 + 48
  else
    text[0] = "0"
  end if
  text[1] = (temp_whole div 10) mod 10 + 48         ' Extract tens
  text[2] = temp_whole mod 10 + 48                  ' Extract ones
  temp_fraction = word(temp2write << (4-RES_SHIFT)) ' Extract temp_fraction
  temp_fraction = temp_fraction and 0x000F          ' and convert it to
  temp_fraction = temp_fraction * 625               ' unsigned int
  text[4] = word(temp_fraction div 1000) + 48       ' Extract thousands
  text[5] = word((temp_fraction div 100) mod 10 + 48) ' Extract hundreds
  text[6] = word((temp_fraction div 10) mod 10 + 48)  ' Extract tens
  text[7] = word(temp_fraction mod 10) + 48         ' Extract ones
  Lcd_Out(2, 5, text)                               ' Print temperature on Lcd
end sub
main:     ' Start of program
ANSEL = 0 ' Configure analog pins as digital I/O
ANSELH = 0
text = "000.0000"
Lcd_Init()               ' Initialize Lcd
Lcd_Cmd(_LCD_CLEAR)      ' Clear Lcd
Lcd_Cmd(_LCD_CURSOR_OFF) ' Turn off cursor
Lcd_Out(1, 1, " Temperature: ")
Lcd_Chr(2,13,178) ' Print degree character, "C" for Centigrades
' Different LCD displays have different char code for degree
Lcd_Chr(2,14,"C") ' If you see greek letter ‘alpha’ type 178 instead of 223
while 1 ' Temperature is read in the main loop
  Ow_Reset(PORTE, 2)       ' Onewire reset signal
  Ow_Write(PORTE, 2, 0xCC) ' Issue command SKIP_ROM
  Ow_Write(PORTE, 2, 0x44) ' Issue command CONVERT_T
  Delay_us(120)
  Ow_Reset(PORTE, 2)
  Ow_Write(PORTE, 2, 0xCC) ' Issue command SKIP_ROM
  Ow_Write(PORTE, 2, 0xBE) ' Issue command READ_SCRATCHPAD
  temp = Ow_Read(PORTE, 2)
  temp = (Ow_Read(PORTE, 2) << 8) + temp
```

```
  Display_Temperature(temp) ' Format and display result on Lcd
  Delay_ms(520)             ' 520 mS delay
wend
end.                        ' End of program
```

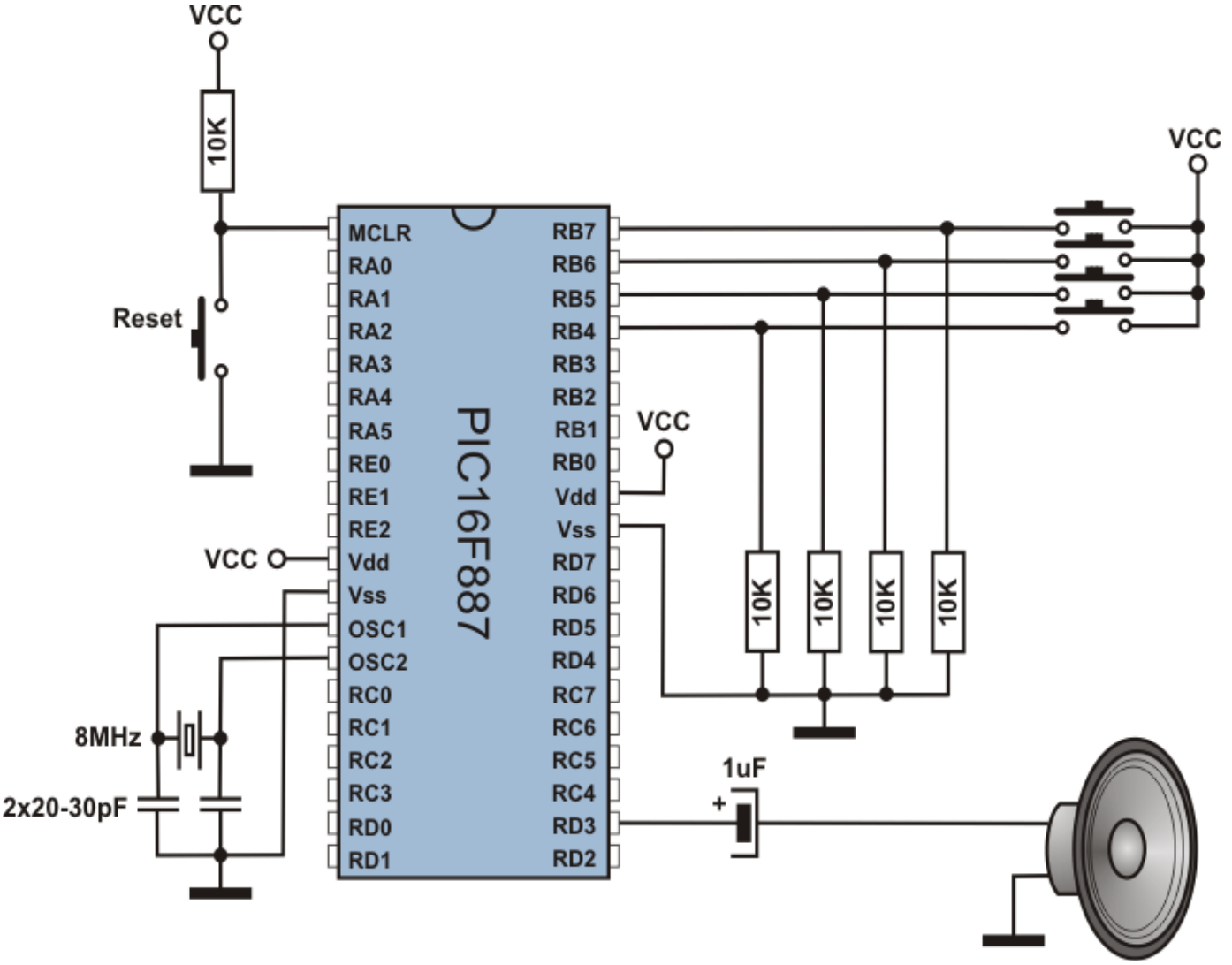
In order to make this example work properly, it is necessary to check the following libraries in the Library Manager prior to compiling:

- One_Wire
- LCD

4.15 EXAMPLE 13

Sound generation, sound library...

Audio signals are often used when it is necessary to call the user's attention to do something, to confirm that one of the push buttons is pressed, to warn that minimum or maximum values are reached etc. It can be just a 'beep' signal as well as longer or shorter melodies. This example demonstrates how to generate a sound using functions belonging to the Sound library.



In adition to these functions, the Button function is also used for testing push buttons.

```
'Header*****************************************************
program example_13 ' Program name
sub procedure Tone1()
  Sound_Play(659, 250) ' Frequency = 659Hz, duration = 250ms
end sub
sub procedure Tone2()
  Sound_Play(698, 250) ' Frequency = 698Hz, duration = 250ms
end sub
sub procedure Tone3()
  Sound_Play(784, 250) ' Frequency = 784Hz, duration = 250ms
end sub
sub procedure Melody() ' Play the melody "Yellow house"
  Tone1() Tone2() Tone3() Tone3()
  Tone1() Tone2() Tone3() Tone3()
  Tone1() Tone2() Tone3()
  Tone1() Tone2() Tone3() Tone3()
  Tone1() Tone2() Tone3()
  Tone3() Tone3() Tone2() Tone2() Tone1()
end sub
sub procedure ToneA() ' Tones used in Melody2 function
  Sound_Play( 880, 50)
end sub
sub procedure ToneC()
  Sound_Play(1046, 50)
end sub
sub procedure ToneE()
  Sound_Play(1318, 50)
end sub
sub procedure Melody2() ' Play Melody2
  dim counter as byte
  for counter = 9 to 1 step -1
    ToneA()
    ToneC()
    ToneE()
  next counter
end sub
main: ' Start of program
ANSEL = 0    ' Configure analog pins as digital I/O
ANSELH = 0
C1ON_bit = 0 ' Disable comparators
C2ON_bit = 0
TRISB = 0xF0 ' Configure RB7..RB4 as inputs and RB3 as output
Sound_Init(PORTD, 3)
Sound_Play(880, 5000)
while TRUE   ' Endless loop
  if (Button(PORTB,7,1,1)) then ' If PORTB.7 is pressed play Tone1
      Tone1()
    while (RB7_bit <> 0)
      nop ' Wait for the button to be released
    wend
  end if
  if (Button(PORTB,6,1,1)) then ' If PORTB.6 is pressed play Tone1
      Tone2()
    while (RB6_bit <> 0)
      nop ' Wait for the button to be released
    wend
  end if
  if (Button(PORTB,5,1,1)) then ' If PORTB.5 is pressed play Tone1
    Melody2()
    while (RB5_bit <> 0)
      nop ' Wait for the button to be released
    wend
  end if
```

```
  if (Button(PORTB,4,1,1)) then ' If PORTB.4 is pressed play Tone1
    Melody()
    while (RB4_bit <> 0)
      nop ' Wait for the button to be released
    wend
  end if
wend
end. ' End of program
```

In order to make this example work properly, it is necessary to check the following libraries in the Library Manager prior to compiling:

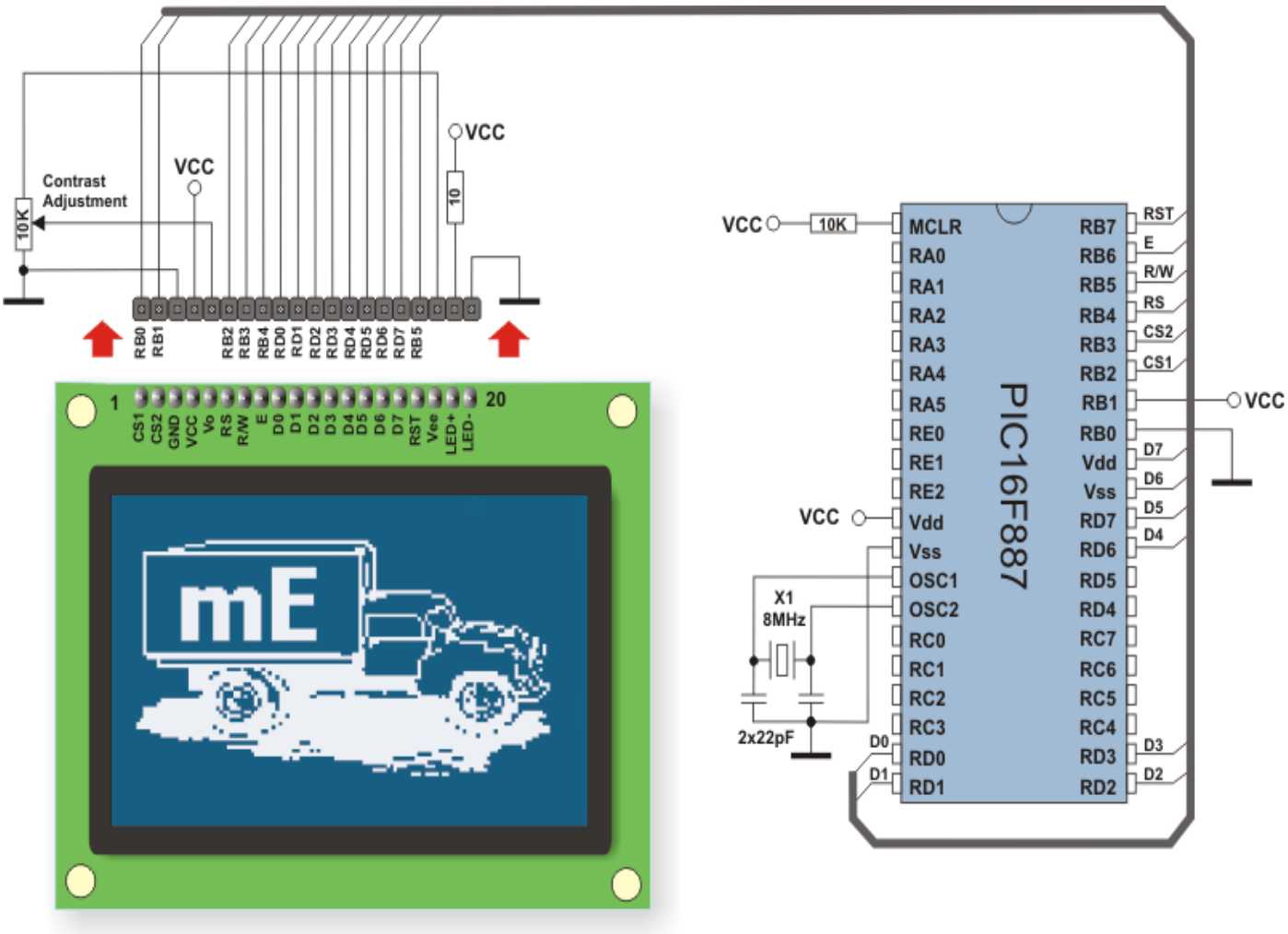
- Button
- Sound

4.16 EXAMPLE 14

Use graphic LCD display

A graphic LCD (GLCD) provides an advanced method for displaying visual messages. While the character LCD can display only alphanumeric characters, the GLCD can also display messages in the form of drawings and bitmaps. The most commonly used graphic LCD has a screen resolution of 128x64 pixel. The GLCD contrast can be adjusted by means of potentiometer P1.

Here, the GLCD displays a truck the bitmap of which is stored in the **`truck_bmp.mbas`** file.




```
'Header*****************************************************
program example_14 ' Program name
dim GLCD_DataPORT as byte at PORTD
dim GLCD_CS1 as sbit at RB0_bit ' Glcd module connections
GLCD_CS2 as sbit at RB1_bit
GLCD_RS as sbit at RB2_bit
GLCD_RW as sbit at RB3_bit
GLCD_EN as sbit at RB4_bit
GLCD_RST as sbit at RB5_bit
dim GLCD_CS1_Direction as sbit at TRISB0_bit
GLCD_CS2_Direction as sbit at TRISB1_bit
GLCD_RS_Direction as sbit at TRISB2_bit
GLCD_RW_Direction as sbit at TRISB3_bit
GLCD_EN_Direction as sbit at TRISB4_bit
GLCD_RST_Direction as sbit at TRISB5_bit ' End Glcd module connections
dim counter as byte
someText as char[18]
sub procedure Delay2S() ' 2 seconds delay sub function
  Delay_ms(2000)
end sub
main:       ' Start of program
ANSEL = 0   ' Configure analog pins as digital I/O
ANSELH = 0
Glcd_Init()     ' Initialize Glcd
Glcd_Fill(0x00) ' Clear Glcd
while TRUE                    ' Endless loop
  Glcd_Image(@truck_bmp)      ' Draw image
  Delay2S() delay2S()
  Glcd_Fill(0x00)             ' Clear Glcd
```

```
  Glcd_Box(62,40,124,63,1)    ' Draw box
  Glcd_Rectangle(5,5,84,35,1) ' Draw rectangle
  Glcd_Line(0, 0, 127, 63, 1) ' Draw line
  Delay2S()
  counter = 5
  while (counter <= 59)       ' Draw horizontal and vertical lines
    Delay_ms(250)
    Glcd_V_Line(2, 54, counter, 1)
    Glcd_H_Line(2, 120, counter, 1)
    Counter = counter + 5
  wend
  Delay2S()
  Glcd_Fill(0x00) ' Clear Glcd
  Glcd_Set_Font(@Character8x7, 8, 7, 32) ' Choose font "Character8x7"
  Glcd_Write_Text("mikroE", 1, 7, 2)     ' Write string
  for counter = 1 to 10 ' Draw circles
    Glcd_Circle(63,32, 3*counter, 1)
  next counter
  Delay2S()
  Glcd_Box(10,20, 70,63, 2)                  ' Draw box
  Delay2S()
  Glcd_Fill(0xFF)                            ' Fill Glcd
  Glcd_Set_Font(@Character8x7, 8, 7, 32)     ' Change font
  someText = "8x7 Font"
  Glcd_Write_Text(someText, 5, 0, 2)         ' Write string
  delay2S()
  Glcd_Set_Font(@System3x5, 3, 5, 32)        ' Change font
  someText = "3X5 CAPITALS ONLY"
  Glcd_Write_Text(someText, 60, 2, 2)        ' Write string
  delay2S()
  Glcd_Set_Font(@font5x7, 5, 7, 32)          ' Change font
  someText = "5x7 Font"
  Glcd_Write_Text(someText, 5, 4, 2)         ' Write string
  delay2S()
  Glcd_Set_Font(@FontSystem5x7_v2, 5, 7, 32) ' Change font
  someText = "5x7 Font (v2)"
  Glcd_Write_Text(someText, 5, 6, 2)         ' Write string
  delay2S()
wend
end. ' End of program
```

truck_bmp.mbas file:

```
module bitmap ' Module with bitmap code
const truck_bmp as byte[1024] =
(0,0,0,0,0,248,8,8,8,8,8,8,12,12,12,12,12,10,10,10,10,10,10,9,9,9,9,9,9,9,9,9,9,
9,9,9,9,9,9,9,9,9,137,137,137,137,137,137,137,137,137,137,137,137,9,9,9,9,9,
9,9,9,9,9,9,9,13,253,13,195,6,252,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,0,
0,0,0,0,0,0,0,0,0,0,240,240,240,240,240,224,224,240,240,240,240,240,224,192,192,22
4,240,240,240,240,240,224,192,0,0,0,255,255,255,255,255,195,195,195,195,195,195,
195,3,0,0,0,0,0,0,0,0,0,0,0,0,255,240,79,224,255,96,96,96,32,32,32,32,32,32,32,3
2,32,32,32,32,32,64,64,64,64,128,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,0,0,0,0,0,0,0,0,0,0,255,255,255,255,255,0,0
,0,0,255,255,255,255,255,0,0,0,0,255,255,255,255,255,0,0,0,255,255,255,255,255,1
29,129,129,129,129,129,129,128,0,0,0,0,0,0,0,0,0,0,0,0,0,255,1,248,8,8,8,8,8,8,8,8
,8,8,8,8,8,8,16,224,24,36,196,70,130,130,133,217,102,112,160,192,96,96,32,32,160
,160,224,224,192,64,64,128,128,192,64,128,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
63,96,96,96,224,96,96,96,96,96,96,99,99,99,99,99,96,96,96,96,99,99,99,99,99,96,9
6,96,96,99,99,99,99,99,96,96,96,99,99,99,99,99,99,99,99,99,99,99,99,99,96,96,96,
96,96,96,96,64,64,64,224,224,255,246,1,14,6,6,2,2,2,2,2,2,2,2,2,2,2,130,67,114,6
2,35,16,16,0,7,3,3,2,4,4,4,4,4,4,4,28,16,16,16,17,17,9,9,41,112,32,67,5,240,126,
```

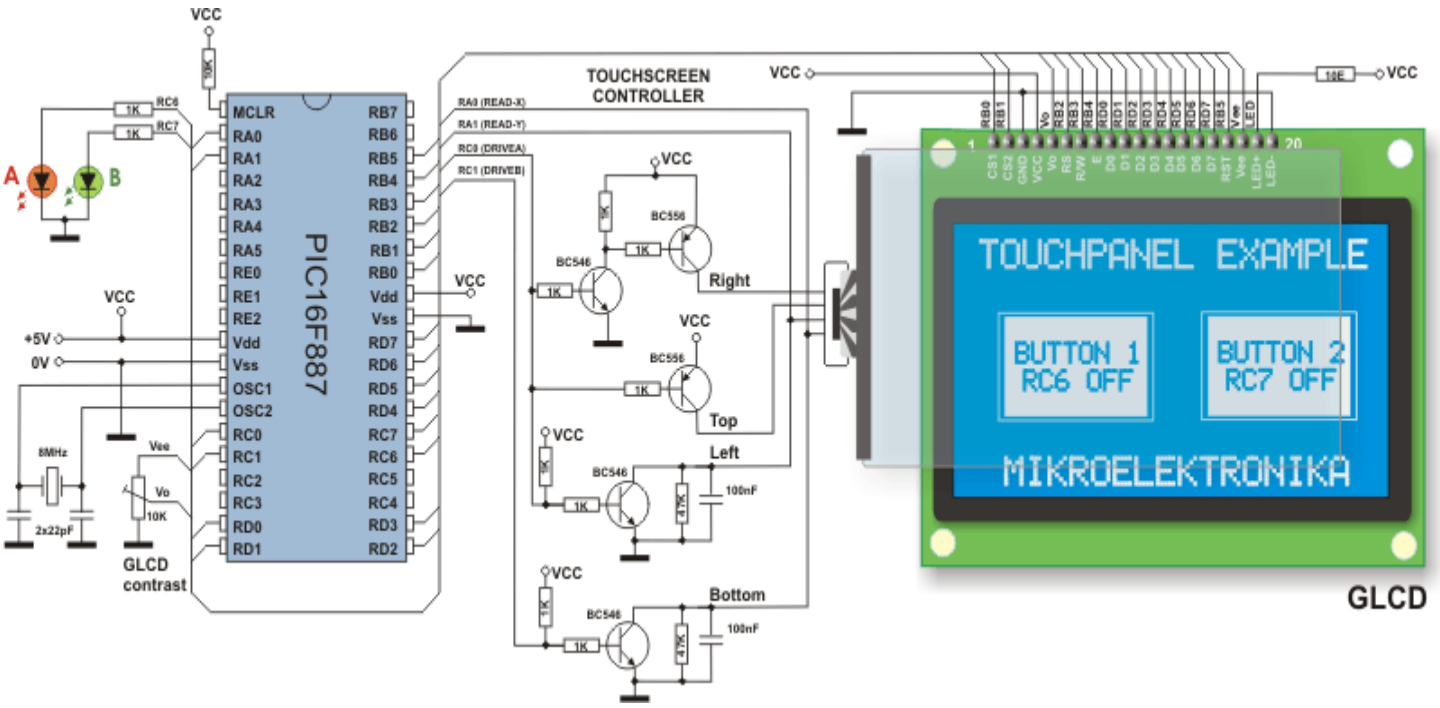
```
174,128,56,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,127,127,127,127,255
,255,247,251,123,191,95,93,125,189,189,63,93,89,177,115,243,229,207,27,63,119,25
5,207,191,255,255,255,255,255,255,255,255,127,127,127,127,127,127,127,127,255,25
5,255,127,127,125,120,120,120,120,120,248,120,120,120,120,120,120,248,248,232,14
3,0,0,0,0,0,0,0,0,128,240,248,120,188,220,92,252,28,28,60,92,92,60,120,248,248,9
6,192,143,168,216,136,49,68,72,50,160,96,0,0,0,0,0,0,0,0,128,192,248,248,248,2
48,252,254,254,254,254,254,254,254,254,254,254,254,255,255,255,255,255,246,239,2
08,246,174,173,169,128,209,208,224,247,249,255,255,252,220,240,127,255,223,255,2
55,255,255,255,255,254,254,255,255,255,255,255,255,255,254,255,255,255,255,255,2
55,255,254,254,254,254,254,254,254,254,254,254,254,254,254,254,254,255,255,255,2
55,255,254,255,190,255,255,253,240,239,221,223,254,168,136,170,196,208,228,230,2
48,127,126,156,223,226,242,242,242,242,242,177,32,0,0,0,0,0,0,0,0,0,1,1,1,1,3,
3,3,7,7,7,7,7,15,15,15,7,15,15,15,7,7,15,14,15,13,15,47,43,43,43,43,43,47,111,23
9,255,253,253,255,254,255,255,255,255,255,191,191,239,239,239,191,255,191,255,25
5,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,25
5,255,255,255,255,255,255,127,127,127,127,255,255,191,191,191,191,255,254,255,25
3,255,255,255,251,255,255,255,127,125,63,31,31,31,31,31,31,63,15,15,7,7,3,3,3,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,3,3,3,11,11,11
,11,7,3,14,6,6,6,2,18,19,19,3,23,21,21,17,1,19,19,3,6,6,14,15,15,7,15,15,15,11,2
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
implements
end. ' End of module
```

In order to make this example work properly, it is necessary to check the GLCD library in the Library Manager before compilation. Also, it is necessary to include the **Bitmap.mbas** document into the project.

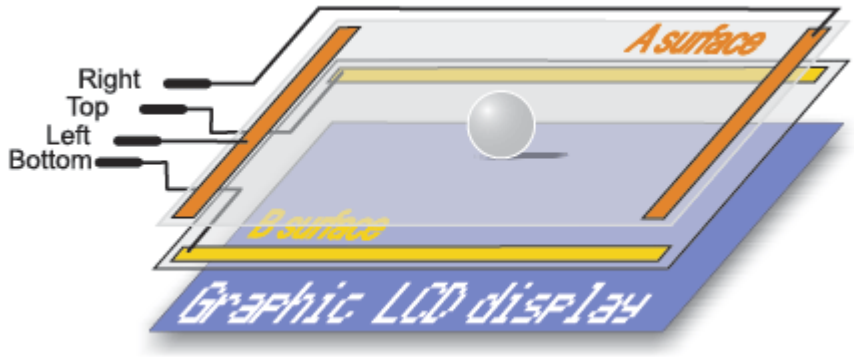
4.17 EXAMPLE 15

Use a touch panel

A touch panel is a thin, self-adhesive transparent panel placed over the screen of a graphic LCD. It is very sensitive to pressure so that even a soft touch causes some changes on the output signal. There are a few types of touch panel. The simplest one is a resistive touch panel.

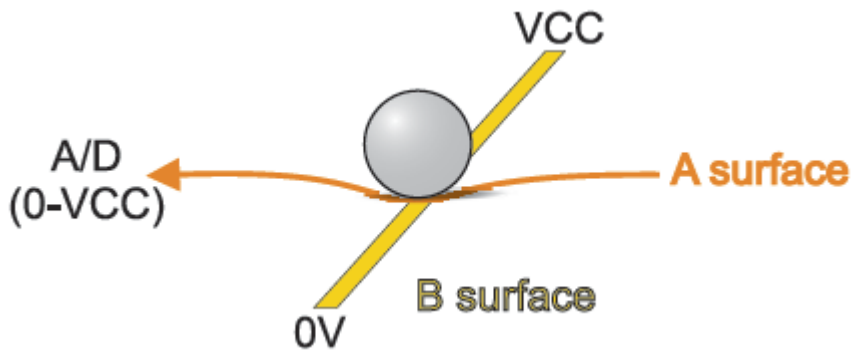


It consists of two transparent rigid foils, forming a ‘sandwich’ structure, that have resistive layers on their inner sides. The resistance of these layers usually does not exceed 1K. The opposite sides of these foils have contacts available for use via a flat cable.



The process of determining coordinates of the point in which the touch panel is pressed can be broken into two steps. The first one is the determination of the X coordinate and the second one is the determination of the Y coordinate of the point.

In order to determine the X coordinate, it is necessary to connect the left contact on the surface A to ground and the right contact to the power supply. This enables a voltage divider to be formed by pressing the touch panel. The value of the divider is read on the bottom contact of the surface B. Voltage can go within the range of 0V to the power supply (5V) and depends on the X coordinate. If the point is closer to the left contact of the surface A, the voltage will be closer to 0V.



In order to determine the Y coordinate, it is necessary to connect the bottom contact on the surface B to ground, and the upper contact to the power supply. In this case, the voltage is read on the left contact of the surface A.

In order to connect a touch panel to the microcontroller it is necessary to provide a circuit for touch panel control. By means of this circuit, the microcontroller connects appropriate contacts of the touch panel to the ground and the power supply (as described above) so as to determine coordinates X and Y. The bottom contact of the surface B and left contact of the surface A are connected to the microcontroller’s A/D converter. The X and Y coordinates are determined by measuring voltage on these contacts, respectively. The related program outlines a menu on the graphic LCD, turns the circuit for touch panel control on/off (driving touch panel) and reads results of A/D conversion which actually represent the X and Y coordinates of the point.

On the basis of these coordinates it is possible to decide what you want the microcontroller to do. In this example, the microcontroller turns on/off two digital pins connected to LEDs A and B.

Functions belonging to the Glcd, Glcd_Fonts and ADC librares are used in this example.

As the touch panel surface is slightly larger than the surface of the graphic LCD, it is necessary to perform the software calibration of the touch panel in order to provide greater accuracy when determining the coordinates

```
'Header*****************************************************
program example_15 ' Name of program
dim GLCD_DataPORT as byte at PORTD ' GLCD module connections
dim GLCD_CS1 as sbit at RB0_bit
GLCD_CS2 as sbit at RB1_bit
GLCD_RS as sbit at RB2_bit
GLCD_RW as sbit at RB3_bit
GLCD_EN as sbit at RB4_bit
GLCD_RST as sbit at RB5_bit
dim GLCD_CS1_Direction as sbit at TRISB0_bit
GLCD_CS2_Direction as sbit at TRISB1_bit
GLCD_RS_Direction as sbit at TRISB2_bit
GLCD_RW_Direction as sbit at TRISB3_bit
GLCD_EN_Direction as sbit at TRISB4_bit
GLCD_RST_Direction as sbit at TRISB5_bit ' End Glcd module connections
dim x_coord, y_coord,
x_coord128, y_coord64 as longint ' Scaled x-y position
sub function GetX() as word ' Reading X
  PORTC.0 = 1               ' DRIVEA = 1 (LEFT drive on, RIGHT drive on, TOP drive off)
  PORTC.1 = 0               ' DRIVEB = 0 (BOTTOM drive off)
  Delay_ms(5)
  result = ADC_Read(0)      ' READ-X (BOTTOM)
end sub
sub function GetY() as word ' Reading Y
  PORTC.0 = 0               ' DRIVEA = 0 (LEFT drive off, RIGHT drive off, TOP drive
on)
  PORTC.1 = 1               ' DRIVEB = 1 (BOTTOM drive on)
  Delay_ms(5)
  result = ADC_Read(1)      ' READ-X (LEFT)
end sub
main:        ' Start of program
PORTA = 0x00
TRISA = 0x03 ' RA0 i RA1 are analog inputs
ANSEL = 0x03
ANSELH = 0   ' Configure other analog pins as digital I/O
PORTC = 0
TRISC = 0    ' PORTC pins are configured as outputs
Glcd_Init()  ' Glcd_Init_EP5
Glcd_Set_Font(@font5x7, 5, 7, 32) ' Choose font size 5x7
Glcd_Fill(0) ' Clear GLCD
Glcd_Write_Text("TOUCHPANEL EXAMPLE",10,0,1)
Glcd_Write_Text("MIKROELEKTRONIKA",17,7,1)
Glcd_Rectangle(8,16,60,48,1) ' Outline two ‘buttons’ on GLCD:
Glcd_Rectangle(68,16,120,48,1)
Glcd_Box(10,18,58,46,1)
Glcd_Box(70,18,118,46,1)
Glcd_Write_Text("BUTTON1",14,3,0)
Glcd_Write_Text("RC6 OFF",14,4,0)
Glcd_Write_Text("BUTTON2",74,3,0)
Glcd_Write_Text("RC7 OFF",74,4,0)
while TRUE ' Read X-Y and convert it to 128x64 space
x_coord = GetX()
y_coord = GetY()
x_coord128 = (x_coord * 128) / 1024
y_coord64 = 64 -((y_coord *64) / 1024)
```

```
' If BUTTON1 is selected:
if ((x_coord128 >= 10) and (x_coord128 <= 58) and (y_coord64 >= 18) and (y_coord64 <=
46)) then
  if(PORTC.6 = 0) then
    PORTC.6 = 1
    Glcd_Write_Text("RC6 ON ",14,4,0)
  else
    PORTC.6 = 0
    Glcd_Write_Text("RC6 OFF",14,4,0)
  end if
end if
' If BUTTON2 is selected:
if ((x_coord128 >= 70) and (x_coord128 <= 118) and (y_coord64 >= 18) and (y_coord64 <=
46)) then
  if(PORTC.7 = 0) then
    PORTC.7 = 1
    Glcd_Write_Text("RC7 ON ",74,4,0)
  else
    PORTC.7 = 0
    Glcd_Write_Text("RC7 OFF",74,4,0)
  end if
end if
Delay_ms(100)
wend ' While true
end. ' End of program
```

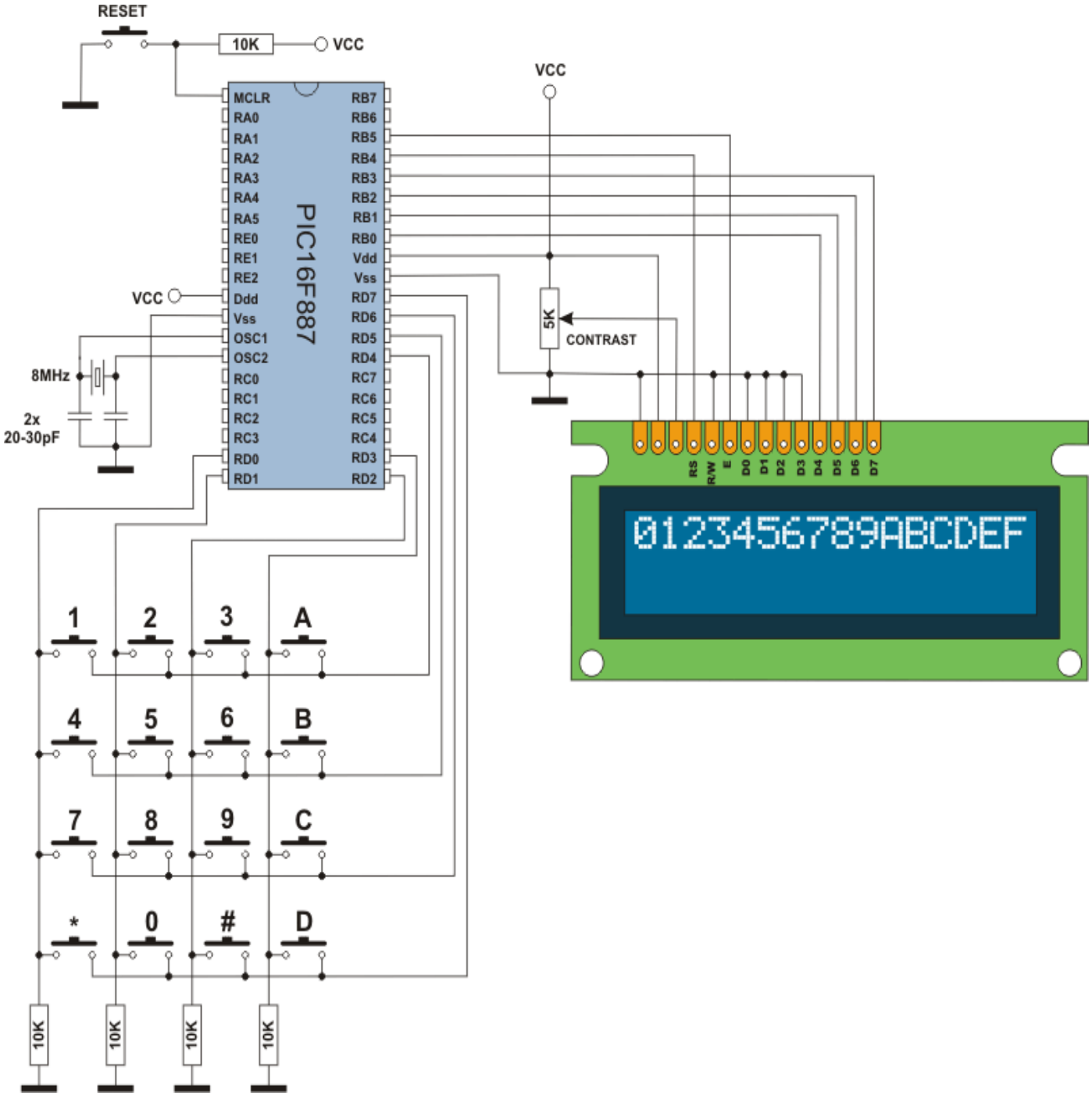
In order to make this example work properly, it is necessary to check the following libraries in the Library Manager prior to compiling:

- GLCD
- ADC
- C_Stdlib

4.18 EXAMPLE 16

Use a 4x4 keypad

A keypad is just a set of push-buttons connected in such a way to form 'rows' and 'columns', thus reducing a number of I/O pins necessary for their connection. A keypad with 16 push-buttons arranged in 4 rows and 4 columns is used in this example. The Keypad 4x4 library contains all the functions necessary for reading this keypad as well as for initializing the port it is connected to. In order to demonstrate the operation of the keypad 4x4, the message will be shown on an LCD display.




```
'Header*****************************************************
program example_16 ' Program name
dim kp, curX, curY as byte
dim keypadPORT as byte at PORTD ' This variable must be defined in all projects using
Keypad Lib.
                                ' It define the port used for keypad connection
dim LCD_RS as sbit at RB4_bit   ' Lcd module connections
LCD_EN as sbit at RB5_bit
LCD_D4 as sbit at RB0_bit
LCD_D5 as sbit at RB1_bit
LCD_D6 as sbit at RB2_bit
LCD_D7 as sbit at RB3_bit
LCD_RS_Direction as sbit at TRISB4_bit
LCD_EN_Direction as sbit at TRISB5_bit
LCD_D4_Direction as sbit at TRISB0_bit
LCD_D5_Direction as sbit at TRISB1_bit
LCD_D6_Direction as sbit at TRISB2_bit
```

```
LCD_D7_Direction as sbit at TRISB3_bit ' End Lcd module connections
main:                ' Start of program
curX=1               ' For keeping a record of the 2x16 LCD cursor position
curY=1
ANSEL = 0            ' Configure analog pins as digital I/O
ANSELH = 0
TRISB = 0
PORTB = 0xFF
Keypad_Init()        ' Initialize keypad on PORTC
Lcd_Init()           ' Initialize LCD on PORTB,
Lcd_Cmd(_LCD_CLEAR) ' Clear display
while true           ' Wait for some key to be pressed and released
  kp = 0
  while kp = 0
    kp = Keypad_Key_Click()
    Delay_ms(10)
  wend
  select case kp     ' Prepare value for output
    case 1 kp = "1"
    case 2 kp = "2"
    case 3 kp = "3"
    case 4 kp = "A"
    case 5 kp = "4"
    case 6 kp = "5"
    case 7 kp = "6"
    case 8 kp = "B"
    case 9 kp = "7"
    case 10 kp = "8"
    case 11 kp = "9"
    case 12 kp = "C"
    case 13 kp = "*"
    case 14 kp = "0"
    case 15 kp = "#"
    case 16 kp = "D"
  end select
  if (curY > 16) then ' Change cursor position
    if (curX = 1) then
      Lcd_Cmd(_LCD_SECOND_ROW)
      curX = 2
      curY = 1
    else
      Lcd_Cmd(_LCD_FIRST_ROW)
      curX = 1
      curY = 1
    end if
  end if

  Lcd_Chr_CP(kp) ' Display on LCD
  Inc(curY)
wend
end. ' End of program
```

In order to make this example work properly, it is necessary to check the following libraries in the Library Manager prior to compiling:

- Keypad4x4
- LCD