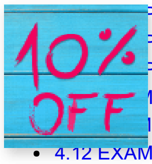
Book: PIC Microcontrollers - Programming in C

Chapter 4: Examples

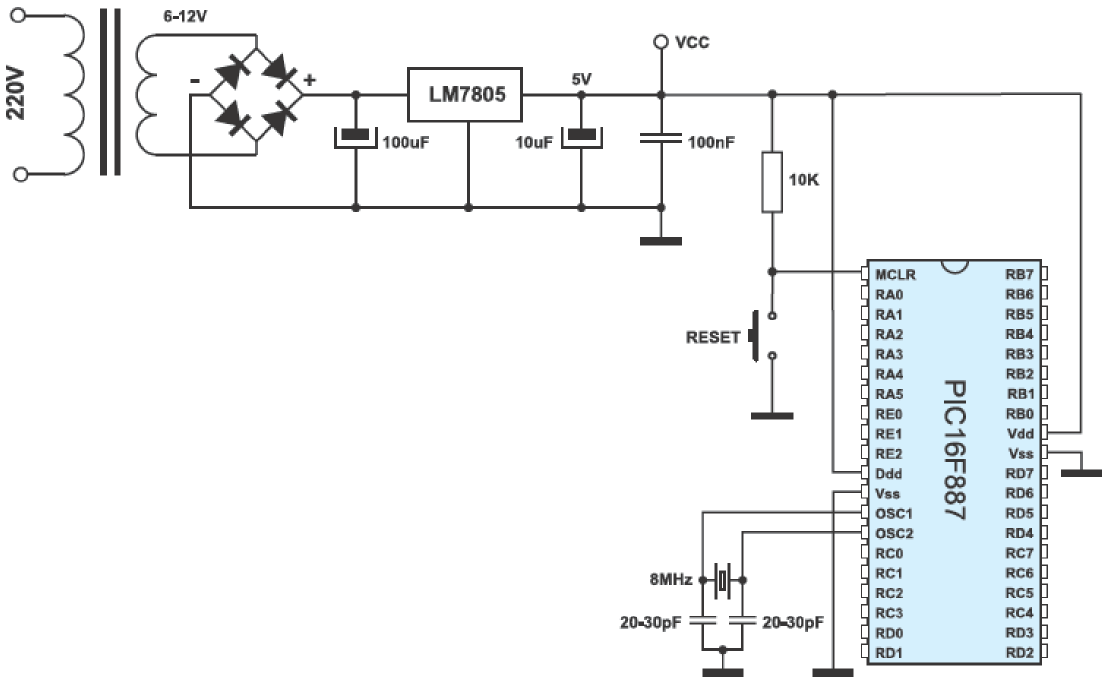
The purpose of this chapter is to provide basic information that one needs to know in order to be able to use microcontrollers successfully in practice. This chapter, therefore, doesn't contain any super interesting program or device schematic with amazing solutions. Instead, the following examples are better proof that program writing is neither a privilege nor a talent issue, but the ability of simply putting puzzle pieces together using directives. Rest assured that design and development of devices mainly consists of the 'test-correct-repeat' work. 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.


220V
6-12V
LM7805
5V
VCC
100uF
10uF
100nF
10K
RESET
8MHz
20-30pF
20-30pF
PIC16F887
MCLR RA0 RA1 RA2 RA3 RA4 RA5 RE0 RE1 RE2 Ddd Vss OSC1 OSC2 RC0 RC1 RC2 RC3 RD0 RD1
RB7 RB6 RB5 RB4 RB3 RB2 RB1 RB0 Vdd Vss RD7 RD6 RD5 RD4 RC7 RC6 RC5 RC4 RD3 RD2

As seen in figure above, it is about simple circuits, but it does not have to be always like that. If the target device is used for controlling expensive machines or life-support devices, everything gets increasingly complicated! However, this solution is sufficient for the time being...

POWER SUPPLY

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

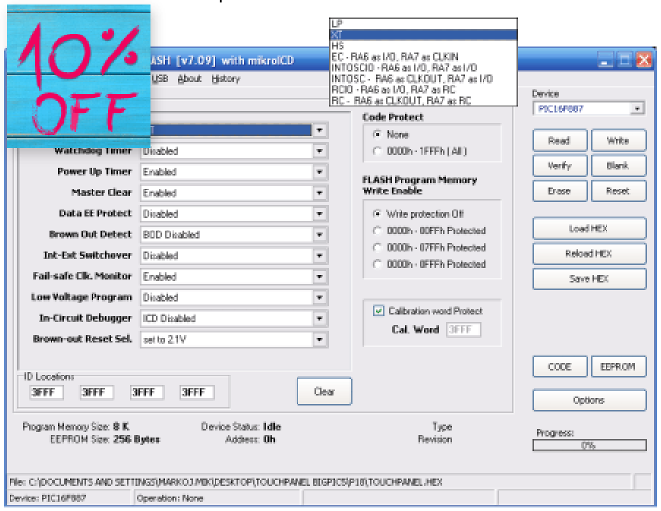
RESET SIGNAL

In order that the microcontroller can operate properly, a logic one (VCC) must be applied on the reset pin. The push button connecting the reset pin MCLR to GND is not necessary. However, it is almost always provided because it enables the microcontroller to return safely to normal operating conditions if something goes wrong. By pushing this button, 0V is brought to the pin, the microcontroller is reset and the program execution starts from the beginning. A10K resistor is used to allow 0V to be applied to the MCLR pin, via the push button, without shorting the 5VDCrail to earth.

CLOCK SIGNAL

Even though the microcontroller has a built-in oscillator, it cannot operate without external components which stabilize its operation and determine its frequency (operating speed of the microcontroller). Depending on elements in use as well as their 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.



LP
XT
HS
EC - RA6 as I/O, RA7 as CLKIN
INTOSCIO - RA6 as I/O, RA7 as I/O
INTOSC - RA6 as CLKOUT, RA7 as I/O
RCIO - RA6 as I/O, RA7 as RC
RC - RA6 as CLKOUT, RA7 as RC

Device: PIC16F887

Watchdog Timer: Disabled
Power Up Timer: Enabled
Master Clear: Enabled
Data EE Protect: Disabled
Brown Out Detect: BOD Disabled
Int-Ext Switchover: Disabled
Fail-safe Clk. Monitor: Enabled
Low Voltage Program: Disabled
In-Circuit Debugger: ICD Disabled
Brown-out Reset Sel.: set to 2.1V

Code Protect
- [✓ None]
- 0000h - 1FFFh (All)

FLASH Program Memory Write Enable
- [✓ Write protection Off]
- 0000h - 00FFh Protected
- 0000h - 07FFh Protected
- 0000h - 0FFFh Protected

[✓ Calibration word Protect]
Cal. Word: 3FFF

ID Locations: 3FFF 3FFF 3FFF 3FFF

Program Memory Size: 8 K
EEPROM Size: 256 Bytes
Device Status: Idle
Address: 0h
Type
Revision
Progress: 0%

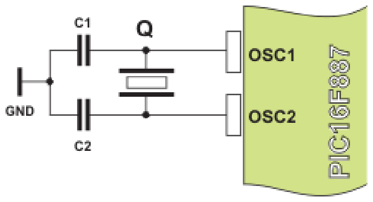
File: C:\DOCUMENTS AND SETTINGS\MARKO.J.MIK\DESKTOP\TOUCHPANEL BIGPIC5\P18\TOUCHPANEL.HEX
Device: PIC16F887
Operation: None

Why are these modes so important? Owing to the fact that it is almost impossible to make a stable oscillator which operates over a wide frequency range, the microcontroller must know which crystal is connected so that it can adjust the operation of its internal electronics to it. This is why all programs used for chip loading contain an option for oscillator mode selection. See figure on the left.

Quartz Crystal

When the quartz crystal is used for frequency stabilization, a 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 crysal casing.

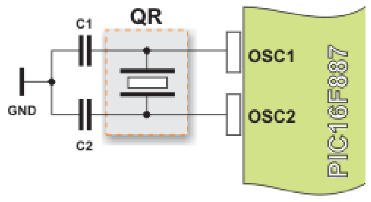
Apart from the crystal, capacitors C1 and C2 must also be connected as per schematic below. Their capacitance is not of great importance. Therefore, the values provided in the table below should be considered as a recommendation, not as a strict rule.



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

Ceramic resonator is cheaper, but very similar to quartz by its function and the way of operation. This is why schematics illustrating their connection to the microcontroller are identical. However, the capacitor value is slightly different due to different electric features. Refer to the table below.

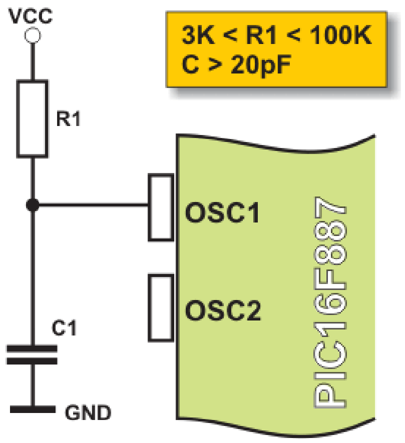


Mode	Frequency	C1, C2
XT	455 KHz	68-100 pF
	2 MHz	15-68 pF
	4 MHz	15-68 pF
HS	8 MHz	10-68 pF
	16 MHz	10-22 pF

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

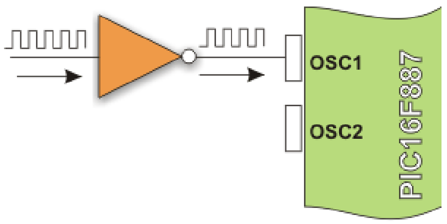
RC Oscillator

If the operating frequency is not of importance then there is no need to use additional expensive components for stabilization. Instead, a simple RC network, as shown in figure below, is sufficient. Since only the input of the local oscillator is used here, the clock signal with the Fosc/4 frequency will appear on the OSC2 pin. This frequency also represents the operating frequency of the microcontroller, i.e. 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 schematics, a clock signal may be generated by an external oscillator. Refer to figure below.



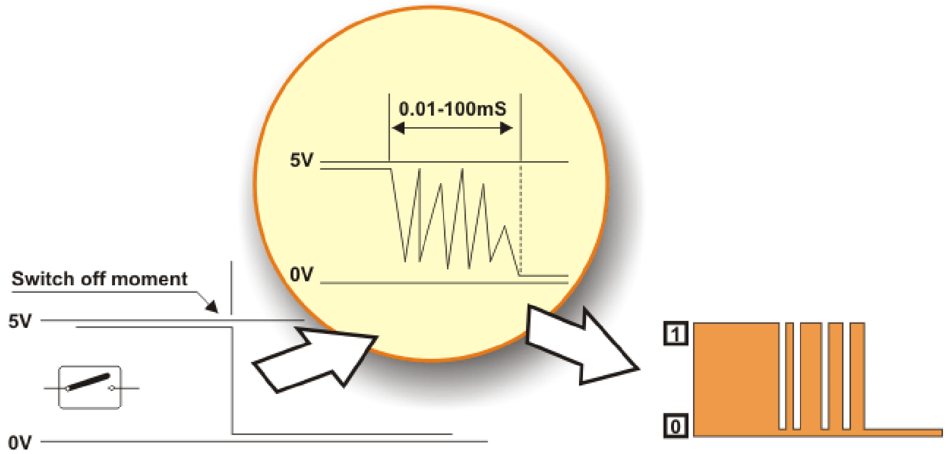
Regardless of the fact that the microcontroller is a product of modern technology, it is of no use if not connected to additional components. Simply put, the appearance of voltage on the microcontroller pins means nothing if not used for performing certain operations such as to turn something on/off,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

4.2 ADDITIONAL COMPONENTS

This section covers the most commonly used additional components in practice such as resistors, transistors, LED diodes, LED displays, LCD displays and RS232 communication circuits.

SWITCHES AND PUSH-BUTTONS

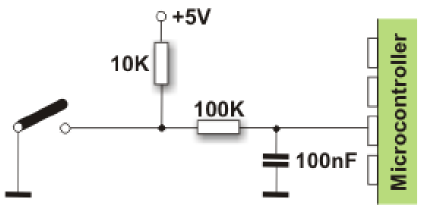
Switches and push-buttons are probably the simplest devices providing the simplest way of detecting the appearance of a voltage on a microcontroller input pin. Nevertheless, it is not as simple as it seems... The reason for it is a contact bounce.



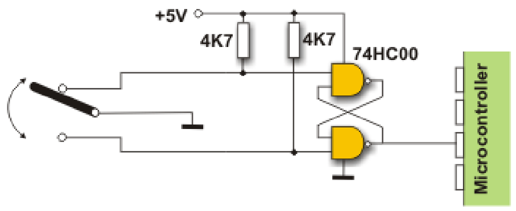
The contact bounce is a common problem with mechanical switches. When the contacts strike together, their momentum and elasticity act together to cause bounce. The result is a rapidly pulsed electrical current instead of a clean transition from zero to full current. It mostly occurs due to vibrations, slight rough spots and dirt between contacts. This effect is usually unnoticeable when using these components in everyday life because the bounce happens too fast to affect most equipment. However, it causes problems in some analog and logic circuits that respond fast enough to misinterpret on/off pulses as a data stream. Anyway, 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

0% of cases!

be easily solved by connecting a simple RC circuit to suppress quick voltage changes. Since the bounce period is not s of components are not precisely determined. In most cases it is recommended to use the values as shown in figure



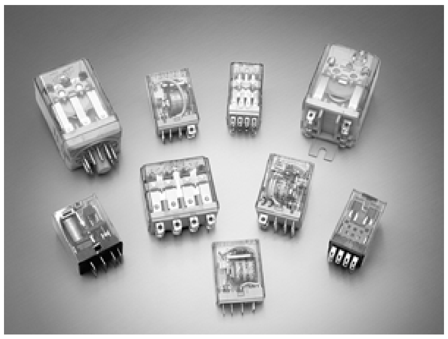
If complete stability is needed then radical measures should be taken. The output of the circuit, shown in figure below (RS flip-flop), will change its logic state only after detecting the first pulse triggered by a contact bounce. This solution is more expensive (SPDT switch), but the problem is definitely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



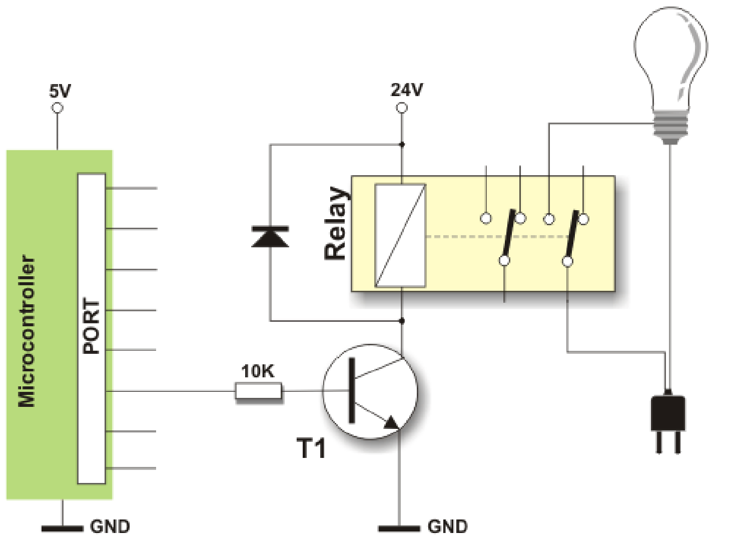
In addition to these hardware solutions, there is also a simple software solution. When the program tests the logic state of an input pin and detects a change, the check should be done one more time after a certain delay. If the program confirms the change, it means that a switch/push button has changed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

RELAY

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 high-power devices such as motors, transformers, heaters, bulbs, etc. These devices are almost always placed away from the board's sensitive components. There are various types of relays, but all of them 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 input and output circuits. Relays usually demand both higher voltage and higher current to start operation, but there are also miniature ones that can be activated by low current directly obtained from a microcontroller pin.

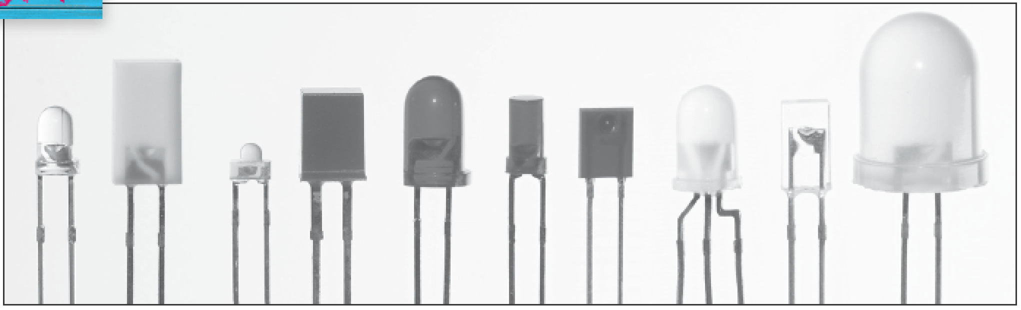


This figure below shows the most commonly used solution.



In order to prevent the appearance of 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.

w all you need to know about LED diodes, but you should also think of the younger generations... Let's see, how to Well...Easily.



Quick Burning

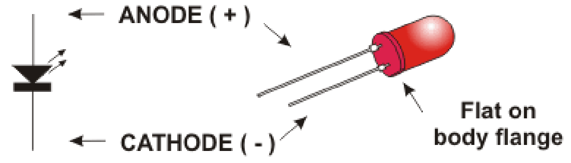
Like any other diode, LEDs have two ends- an anode and a cathode. Connect a diode properly to the power supply voltage and it will happily emit light. Turn the diode upside down and apply the same power supply voltage (even for a moment). It will not emit light - NEVER AGAIN!

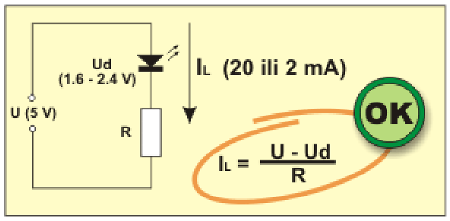
Slow Burning

There is a nominal, i.e. maximum current limitation 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

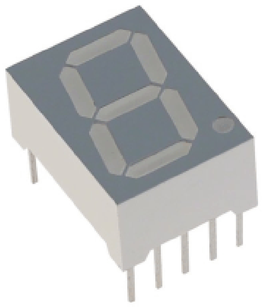
Similarly, all you need to do is to discard a current limiting resistor shown below. Depending on the power supply voltage, the effects might be spectacular!





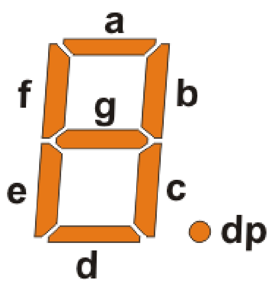
LED DISPLAY

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 the 7-segment display. It is composed of 8 LEDs. Seven segments of a digit are arranged as a rectangle for symbol displaying, whereas the additional segment is used for the purpose of displaying decimal point. In order to simplify connection, anodes or cathodes of all diodes are connected to the common pin so that there are common anode displays and common cathode displays, respectively. Segments are marked with the letters from a to g, plus dp, as shown in figure below. When connecting, each diode is treated separately, which means that each must have its own current limiting resistor.



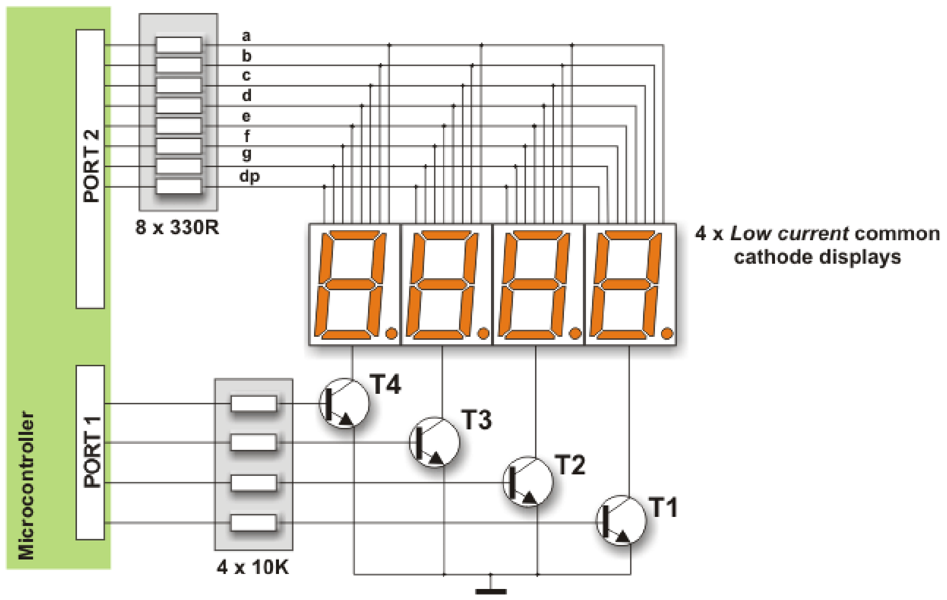
Here are a few important things that you should pay attention to when buying LED displays:

- As mentioned, depending on whether anodes or cathodes are connected to the common pin, there are common anode displays and common cathode displays. As for their appearance, there is no difference between these displays at all so it is recommended to check carefully prior to installing them which one is used.
- Each microcontroller pin has a maximum current limitation it can receive or give. Thus, if several displays are connected to the microcontroller it is recommended to use the so called *Low current* LEDs using only 2mA for the operation.
- Display segments are usually marked with the letters from a to g, but there is no fast rule indicating to which display pins they are connected. For this reason it is very important to check connecting prior to commencing writing a program or designing a device.



Displays connected to the microcontroller usually occupy a large number of valuable I/O pins, which can be a big problem especially when it is needed to display multi digit numbers. The problem is more than obvious if, for example, it is needed to display two 6-digit numbers (a simple calculation shows that 96 output pins are needed in this case). The solution to this problem is called MULTIPLEXING.

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

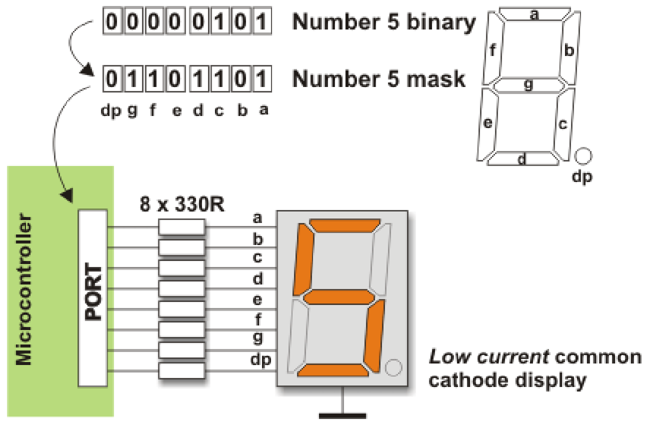


Here is an explanation on the figure above. First a byte representing units is applied on a microcontroller PORT2 and the transistor T1 is activated at the same time. After a while, the transistor T1 is turned off, a byte representing tens is applied on the PORT2 and the transistor T2 is activated. This process is being cyclically repeated 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 any digit. 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 go through the following procedure:

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

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



0 0 0 0 0 1 0 1 Number 5 binary
0 1 1 0 1 1 0 1 Number 5 mask
dp g f e d c b a
a b c d e f g dp
8 x 330R
Microcontroller
PORT
Low current common cathode display

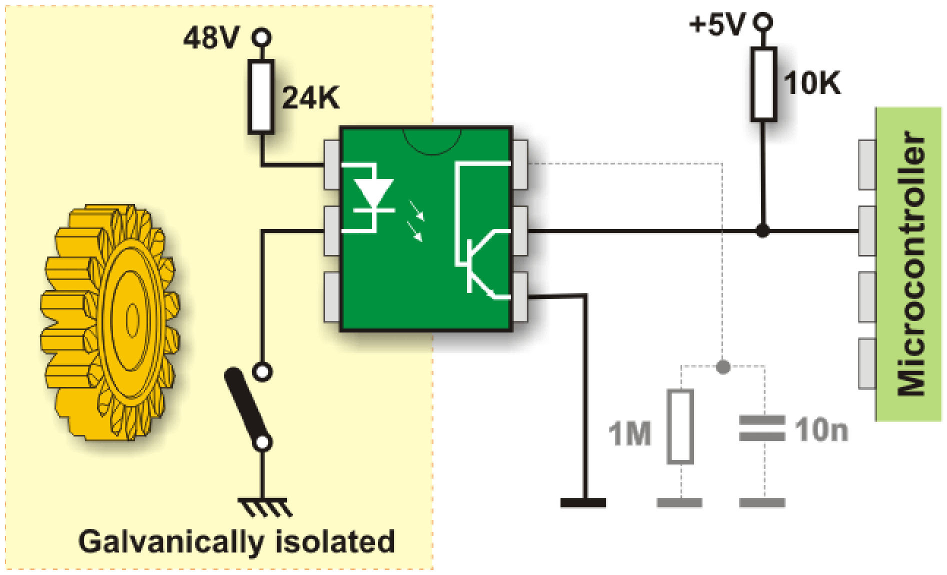
DIGITS TO	DISPLAY SEGMENTS							
	dp	a	b	c	d	e	f	g
[illegible]	0	1	1	1	1	1	1	0
[illegible]	0	0	1	1	0	0	0	0
2	0	1	1	0	1	1	0	1
3	0	1	1	1	1	0	0	1
4	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1
6	0	1	0	1	1	1	1	1
7	0	1	1	1	0	0	0	0
8	0	1	1	1	1	1	1	1
9	0	1	1	1	1	0	1	1

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

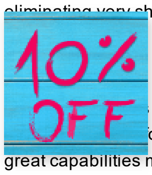
In the event that common anode displays are used, all ones contained in the previous table should be replaced by zeros and vice versa. Additionally, PNP transistors should be used as drivers.

OPTOCOUPLER

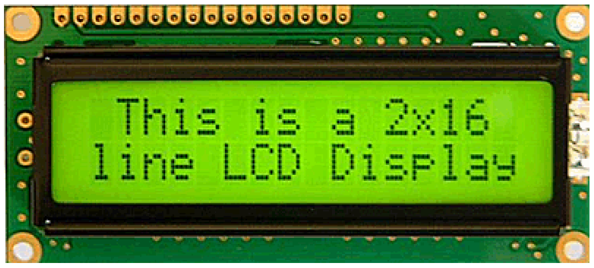
An optocoupler is a device commonly used to galvanically separate microcontroller electronics from any potentially dangerous current or voltage in its surroundings. Optocouplers usually have one, two or four light sources (LED diodes) on their input while on their output, opposite to diodes, there is the same number of elements sensitive to light (phototransistors, photo-thyristors or photo-triacs). The point is that an optocoupler uses a short optical transmission path to transfer a signal between the elements of circuit, while keeping them electrically isolated. This isolation makes sense only if diodes and photosensitive elements are separately powered. In this way, the microcontroller and expensive additional electronics are completely protected from high voltage and noises which are the most common cause of destroying, damaging or unstable operation of electronic devices in practice. The most frequently used optocouplers are those with phototransistors on their outputs. 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 represented by a broken line in the figure above denotes an optional connection which lessens the effects of noises by eliminating very short pulses.



specifically manufactured to be used with microcontrollers, which means that it cannot be activated by standard IC or displaying different messages on a miniature liquid crystal display. The model described here is for its low price and great capabilities most frequently used in practice. It is based on the HD44780 microcontroller (*Hitachi*) and can display messages in two lines with 16 characters each. It can display all the letters of alphabet, Greek letters, punctuation marks, mathematical symbols etc. It is also possible to display symbols made up by the user. Other useful features include automatic message shift (left and right), cursor appearance, LED backlight etc.



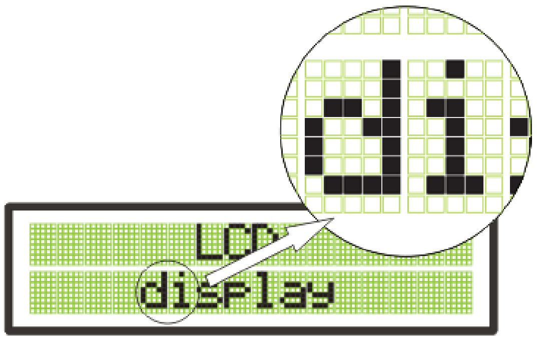
LCD Display Pins

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

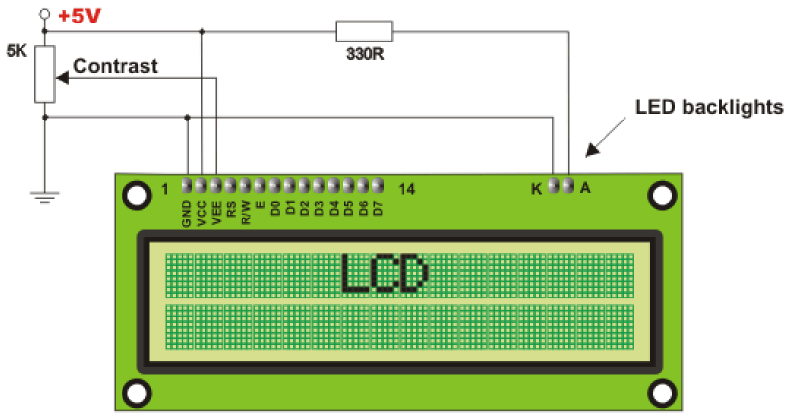
FUNCTION	PIN NUMBER	NAME	LOGIC STATE	DESCRIPTION
Ground	1	Vss	-	0V
Power supply	2	Vdd	-	+5V
Contrast	3	Vee	-	0 - Vdd
Control of operating	4	RS	0 1	D0 – D7 are interpreted as commands D0 – D7 are interpreted as data
	5	R/W	0 1	Write data (from controller to LCD) Read data (from LCD to controller)
	6	E	0 1 From 1 to 0	Access to LCD disabled Normal operating Data/commands are 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 can display two lines with 16 characters each. Every character consists of 5x8 or 5x11 dot matrix. This book covers a 5x8 character display which is most commonly used.



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



If there are no characters displayed or if all of them 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 mode of operation has been changed (writing in one or two lines).

LCD Memory

LCD display contains three memory blocks:

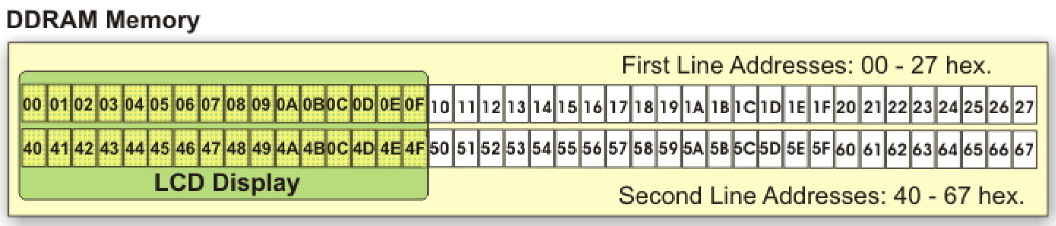
- DDRAM Display Data RAM;
- CGRAM Character Generator RAM; and
- CGROM Character Generator ROM.

DDRAM Memory

DDRAM memory is used for storing characters to be displayed. The size of this memory is capable of storing 80 characters. Some memory locations are directly connected to the characters on display.

Everything works quite simply: it is enough to configure the display to increment addresses automatically (shift right) and set the starting address for the message to be displayed (for example 00 hex).

Afterwards, all characters sent through lines D0-D7 will be displayed in the message format we are used to- from left to right. In this case, displaying starts from the first field of the first line because the initial address is 00 hex. If more than 16 characters are sent, then all of them will be memorized, but only the first sixteen characters will be visible. In order to display the rest of them, the shift command should be used. Virtually, everything looks as if the LCD display is a window which *shifts* left-right over memory locations containing different characters. In reality, this is how the effect of the message shifting over the screen has been created.

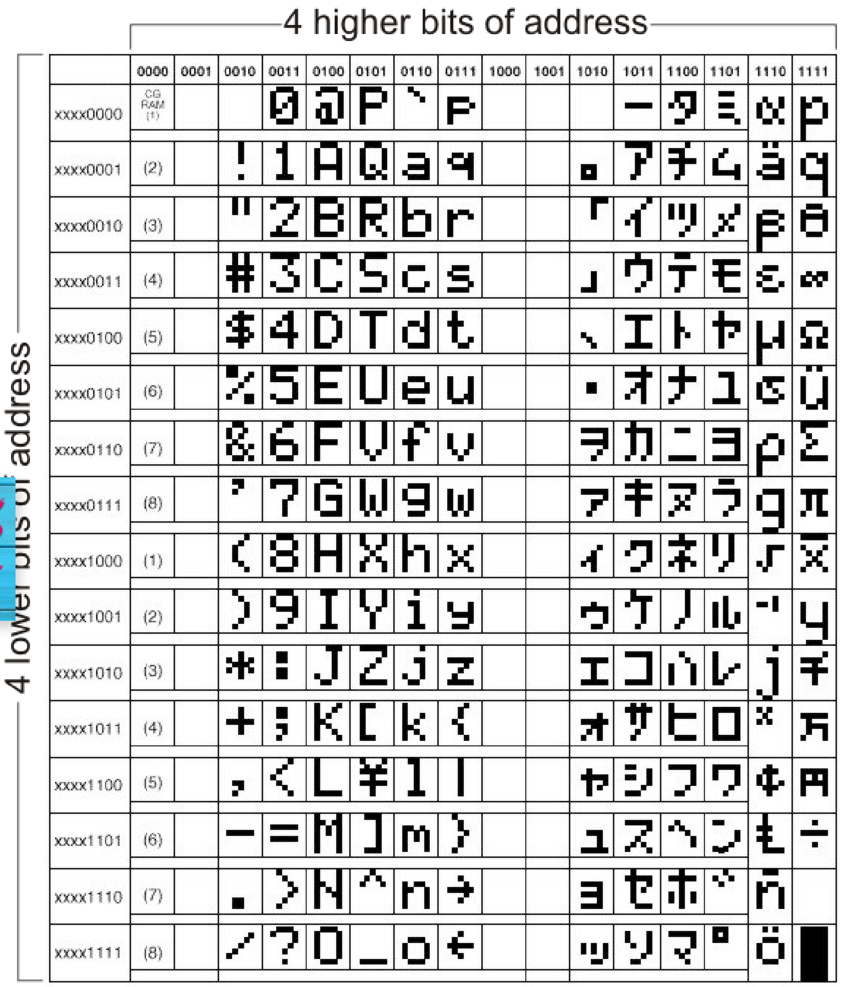

DDRAM Memory
First Line Addresses: 00 - 27 hex.
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27
40 41 42 43 44 45 46 47 48 49 4A 4B 0C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67
LCD Display
Second Line Addresses: 40 - 67 hex.

If the cursor is on, it appears at the currently addressed location. In other words, when a character appears at the cursor position, it will automatically move to the next addressed location.

This is a sort of RAM memory so that data can be written to and read from it, but its content is irretrievably lost when the power goes off.

CGROM Memory

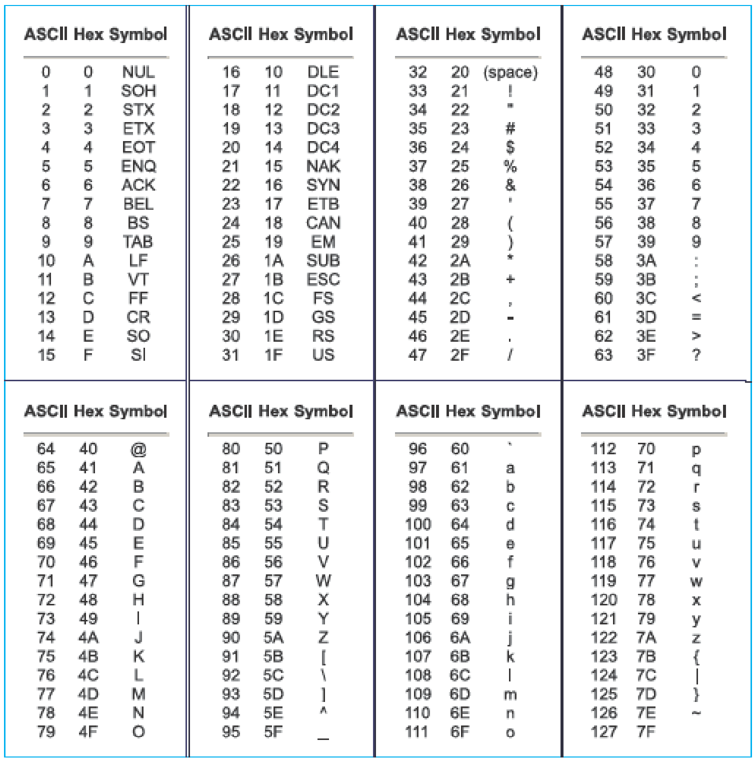
CGROM memory contains a standard character map with all characters that can be displayed on the screen. Each character is assigned to one memory location:


4 higher bits of address
4 lower bits of address

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	ˣ	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

The addresses of CGROM memory locations match the characters of ASCII. If the program being currently executed encounters a command ‘send character P to port’ then the binary value 0101 0000 appears on the port. This value is the ASCII equivalent to the character P. It is then written to an LCD, which results in displaying the symbol from the 0101 0000 location of CGROM. In other words, the character ‘P’ is displayed. This applies to all letters of alphabet (capitals and small), but not to numbers. As seen on the previous map, addresses of all digits are pushed forward by 48 relative to their values (digit 0 address is 48, digit 1 address is 49, digit 2 address is 50 etc.). Accordingly, in order to display digits correctly it is necessary to add the decimal number 48 to each of them prior to being sent to an LCD.

What is ASCII? From their inception till today, computers can recognize only numbers, but not letters. It means that all data a computer swaps with a peripheral device has a binary format even though the same is recognized by the man as letters (the 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 numerical equivalents.



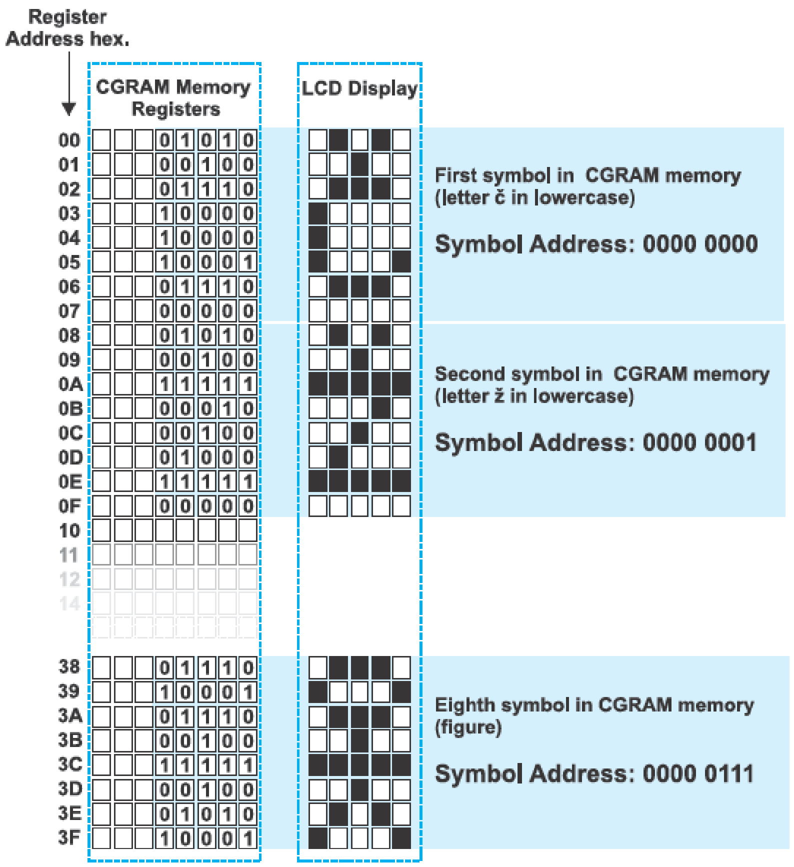
ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

CGRAM Memory

Apart from standard characters, the LCD display can also display symbols defined by the user itself. It can be any symbol in the size of 5x8 pixels. RAM memory called CGRAM in the size of 64 bytes enables it.

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



Symbols are usually defined at the beginnig of the program by simple writing zeros and ones to registers of CGRAM memory so that they form desired shapes. In order to display them it is sufficient to specify their address. Pay attention to the first coloumn in the CGROM map of characters. It doesn't contain RAM memory addresses, but symbols being discussed here. In this example, 'display 0' means - display 'č', 'display 1' means - display 'ž' etc.

LCD Basic Commands

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

- **RS = 1** - Bits D0 - D7 are addresses of the characters to be displayed. LCD processor addresses one character from the character map and displays it. The DDRAM address specifies location on which the character is to be displayed. This address is defined prior to transferring character or the address of the previously transferred character is automatically incremented.
- **RS = 0** - Bits D0 - D7 are commands for setting the display mode.

Here is a list of commands recognized by the 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
M address	0	0	0	1	CGRAM address						40uS
M address	0	0	1	DDRAM address							40uS
" flag (BF)	0	1	BF	DDRAM address							-
M 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
```

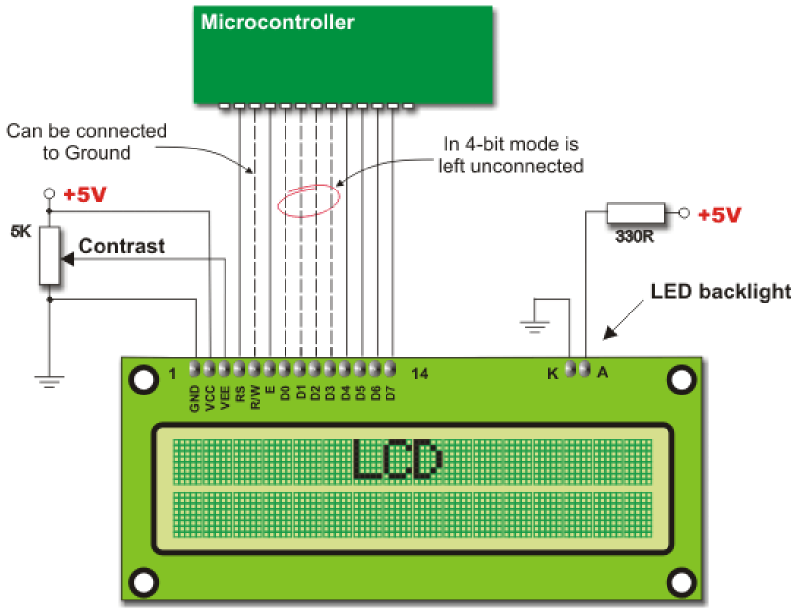
WHAT IS THE BUSY FLAG?

Compared to the microcontroller, the LCD is an extremely slow component. For this reason, it was necessary to provide a signal which would, upon command execution, indicate that the display is ready for the next piece of data. That signal, called the *busy flag*, can be read from the line D7. The 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 LCD modes. The appropriate mode is selected at the beginning of the operation in the process called 'initialization'. The 8-bit LCD mode uses outputs 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, 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 through the lines D4-D7), then four lower bits. Initialization enables the LCD to link and interpret received bits correctly.



Data is rarely read from the LCD (it is mainly transferred from the microcontroller to the LCD) so it is often possible to save an extra I/O pin

ing the R/W pin to the Ground. Such a saving has its price. Messages will be normally displayed, but it will not be possible
ag since it is not possible to read the display either. Fortunately, there is a simple solution. After sending a character or a
ortant to give the LCD enough time to do its job. Owing to the fact that the execution of a command may last for
4mS, it will be sufficient to wait about 2mS for the LCD.

LCD Initialization

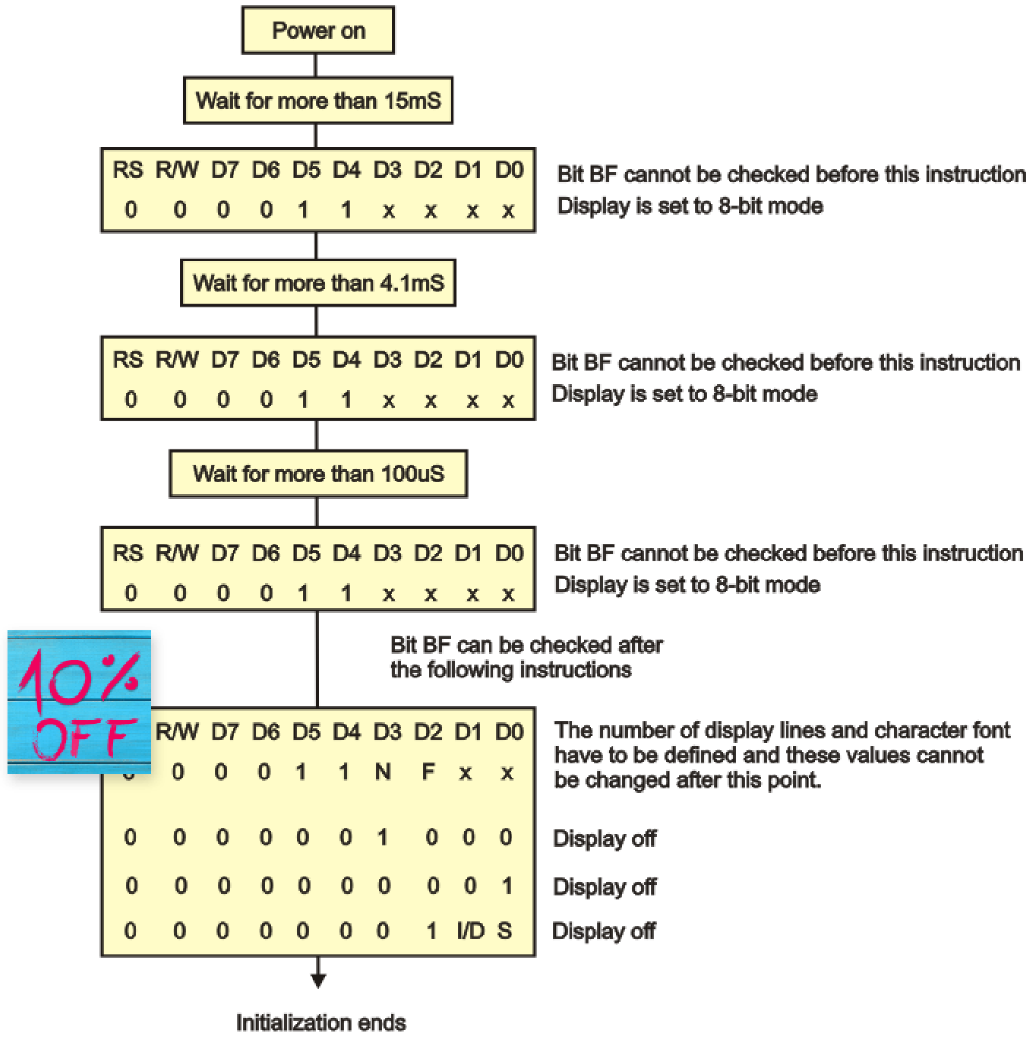
The LCD is automatically cleared when powered up. It lasts for approximately 15mS. After this, it is ready for operation. The mode of operation is set by default, which means that:

1. Display is cleared.
2. Mode
   DL = 1 - Communication through 8-bit interface
   N = 0 - Messages are displayed in one line
   F = 0 - Character font 5 x 8 dots
3. Display/Cursor on/off
   D = 0 - Display off
   U = 0 - Cursor off
   B = 0 - Cursor blink off
4. Character entry
   ID = 1 Displayed addresses are automatically incremented by 1
   S = 0 Display shift off

Automatic reset mostly occurs without any problems. Mostly, but not always! If for any reason the power supply voltage doesn't reach full value within 10mS, the display will start to perform completely unpredictably. If the voltage unit is not able to meet that condition or if it is needed to provide completely safe operation, the process of initialization is applied. Initialization, among other things, causes a new reset by enabling the display to operate normally.

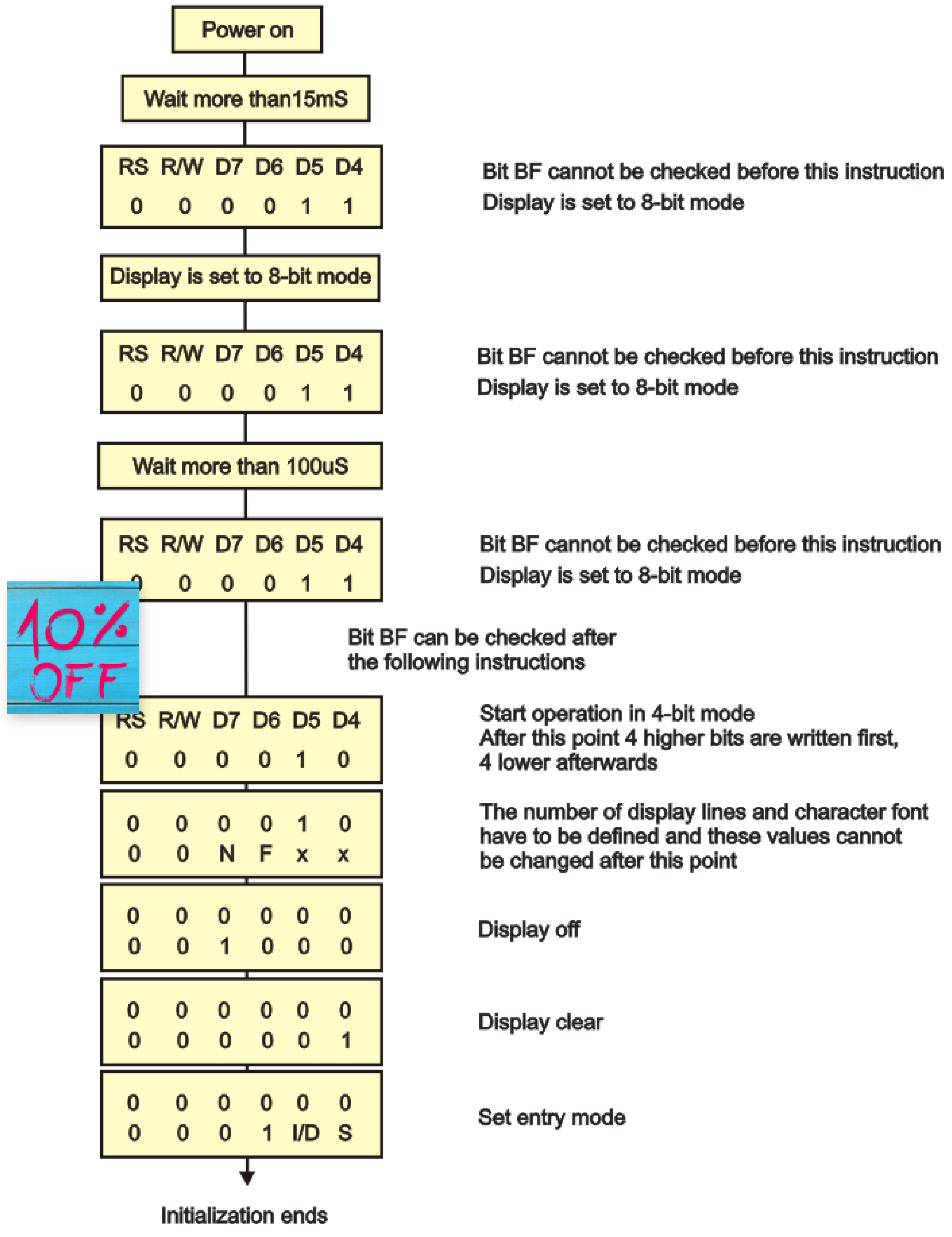
There are two initialization algorithms. Which one is to be performed depends on whether connecting to the microcontroller is through 4- or 8-bit interface. In both cases, all that's left to do after initialization is to specify basic commands and of course - to display messages.

Refer to figure below for the procedure in 8-bit initialization:


Power on
Wait for more than 15mS
RS R/W D7 D6 D5 D4 D3 D2 D1 D0
0 0 0 0 1 1 x x x x
Bit BF cannot be checked before this instruction
Display is set to 8-bit mode
Wait for more than 4.1mS
RS R/W D7 D6 D5 D4 D3 D2 D1 D0
0 0 0 0 1 1 x x x x
Bit BF cannot be checked before this instruction
Display is set to 8-bit mode
Wait for more than 100uS
RS R/W D7 D6 D5 D4 D3 D2 D1 D0
0 0 0 0 1 1 x x x x
Bit BF cannot be checked before this instruction
Display is set to 8-bit mode
Bit BF can be checked after the following instructions
R/W D7 D6 D5 D4 D3 D2 D1 D0
0 0 0 1 1 N F x x
The number of display lines and character font have to be defined and these values cannot be changed after this point.
0 0 0 0 0 0 1 0 0 0
Display off
0 0 0 0 0 0 0 0 0 1
Display off
0 0 0 0 0 0 0 1 I/D S
Display off
Initialization ends
10% OFF

It is not a mistake! In this algorithm, the same value is transferred three times in a row.

The procedure in 4-bit initialization is as follows:


Power on
Wait more than15mS
RS R/W D7 D6 D5 D4
0 0 0 0 1 1
Bit BF cannot be checked before this instruction
Display is set to 8-bit mode
Display is set to 8-bit mode
RS R/W D7 D6 D5 D4
0 0 0 0 1 1
Bit BF cannot be checked before this instruction
Display is set to 8-bit mode
Wait more than 100uS
RS R/W D7 D6 D5 D4
0 0 0 0 1 1
Bit BF cannot be checked before this instruction
Display is set to 8-bit mode
Bit BF can be checked after the following instructions
RS R/W D7 D6 D5 D4
0 0 0 0 1 0
Start operation in 4-bit mode
After this point 4 higher bits are written first, 4 lower afterwards
0 0 0 0 1 0
0 0 N F x x
The number of display lines and character font have to be defined and these values cannot be changed after this point
0 0 0 0 0 0
0 0 1 0 0 0
Display off
0 0 0 0 0 0
0 0 0 0 0 1
Display clear
0 0 0 0 0 0
0 0 0 1 I/D S
Set entry mode
Initialization ends

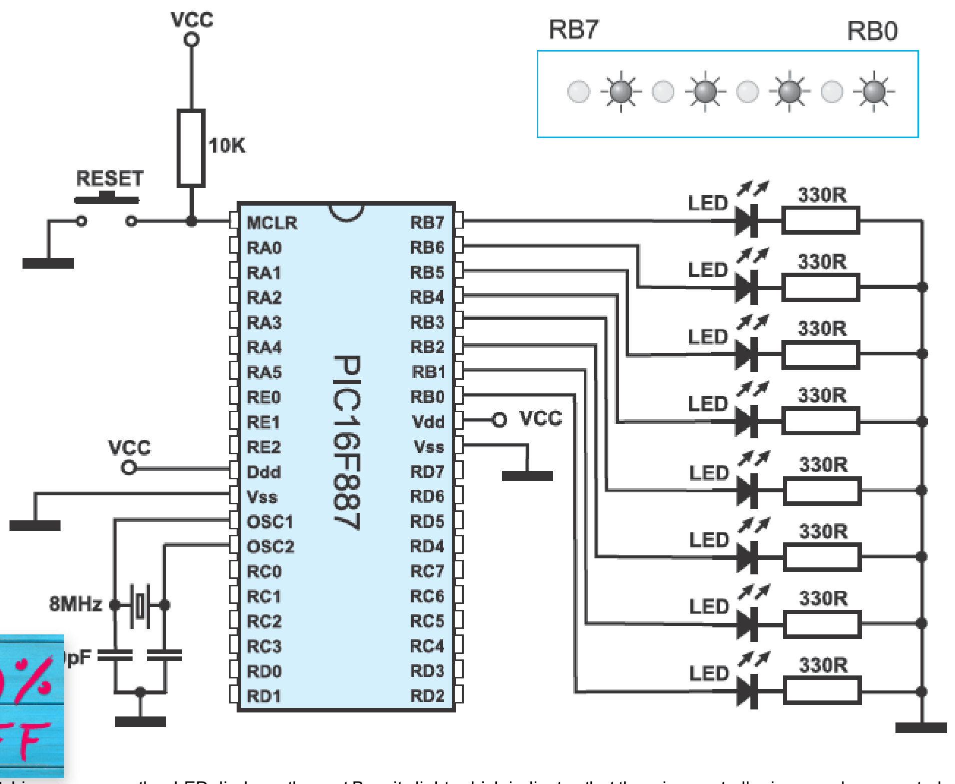
Let's do it in mikroC...

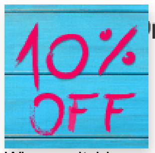
```
/* In mikroC for PIC, it is sufficient to write only one function to perform all
   described operations for LCD initialization. */

...
Lcd_Init(); // Initialize LCD
...
```

4.3 EXAMPLE 1

Writing header, configuring I/O pins, using delay function and switch operator

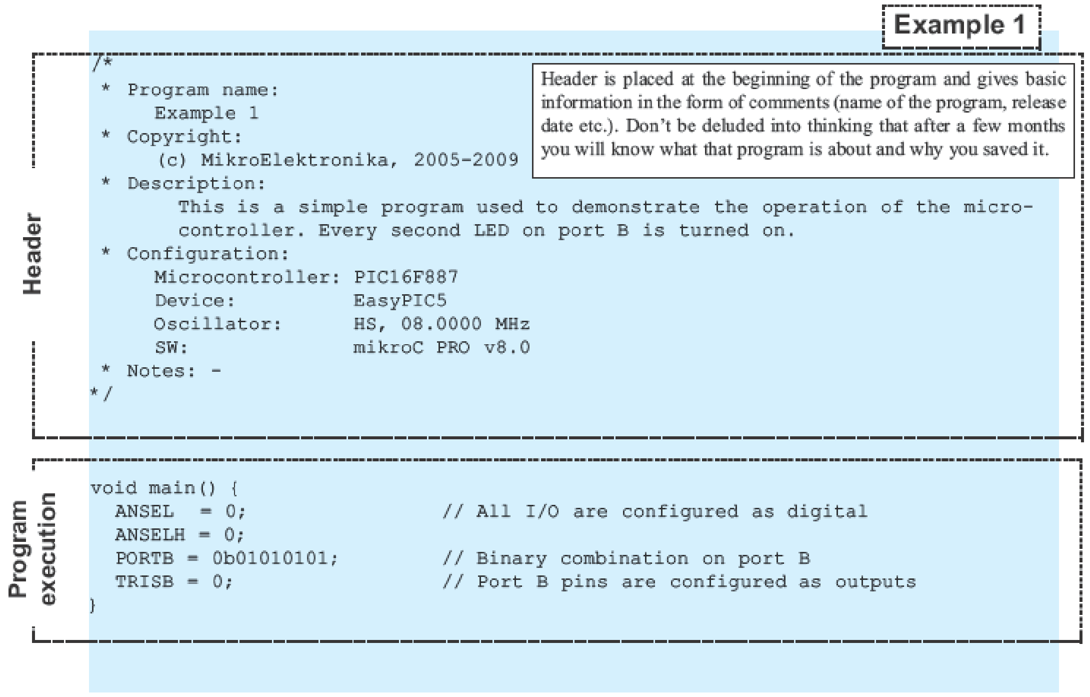
The only purpose of this program is to turn on a few LED diodes on port B. Anyway, use this example to study what a real program looks like. Figure below shows connection schematic, while the program is on the next page.





When switching on, every other LED diode on the port B emits light, which indicates that the microcontroller is properly connected and operates normally.

This example describes a correctly written header. It's the same for all the programs described in this book. To skip repetitiveness, it will not be written in the following examples, but is considered to be at the beginning of every program and marked as 'Header'.


Example 1
Header
Program execution

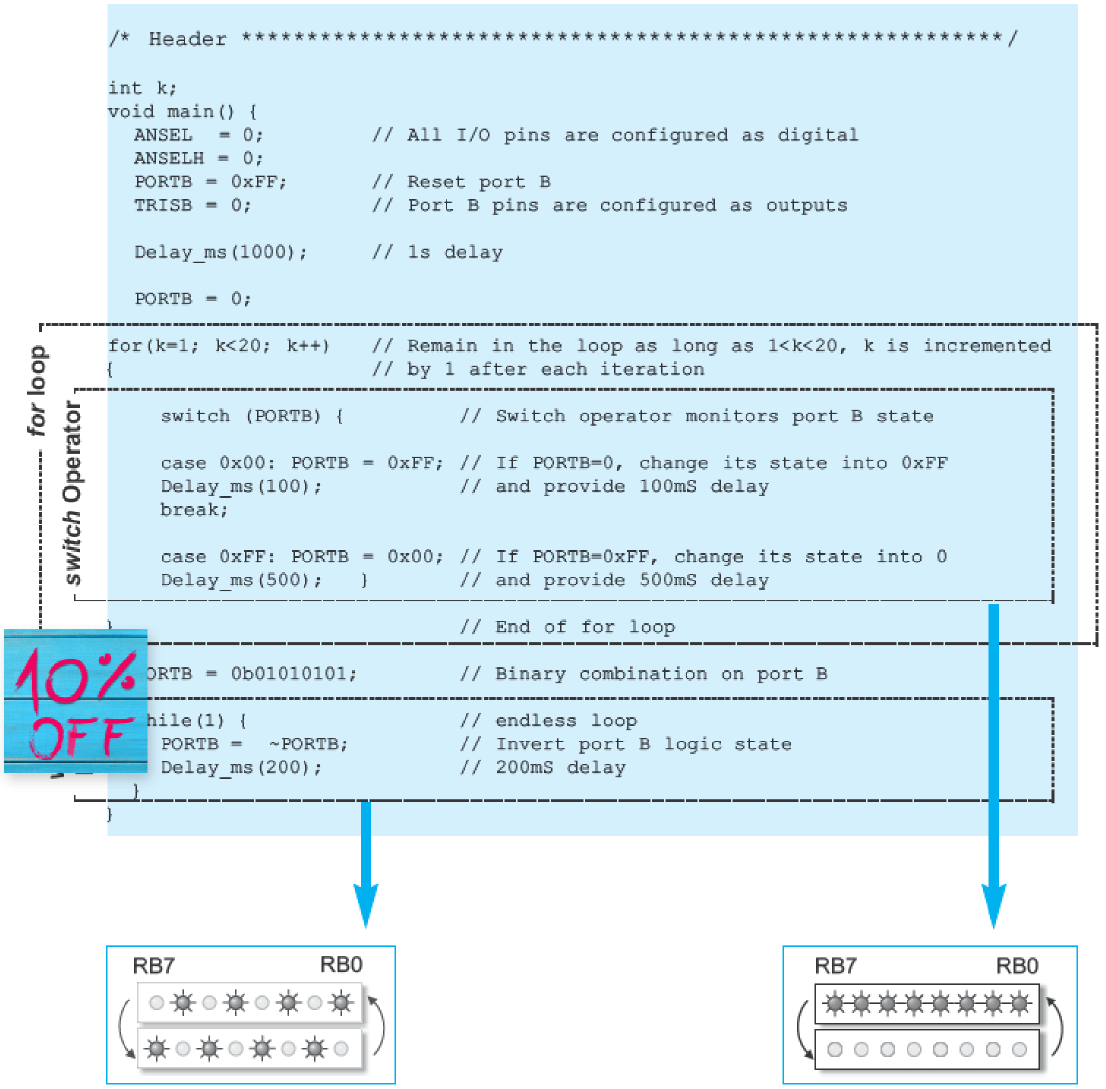
Header is placed at the beginning of the program and gives basic information in the form of comments (name of the program, release date etc.). Don't be deluded into thinking that after a few months you will know what that program is about and why you saved it.

```
/*
 * Program name:
     Example 1
 * Copyright:
     (c) MikroElektronika, 2005-2009
 * Description:
       This is a simple program used to demonstrate the operation of the micro-
       controller. Every second LED on port B is turned on.
 * Configuration:
     Microcontroller:  PIC16F887
     Device:           EasyPIC5
     Oscillator:       HS, 08.0000 MHz
     SW:               mikroC PRO v8.0
 * Notes: -
*/
```

```
void main() {
  ANSEL  = 0;             // All I/O are configured as digital
  ANSELH = 0;
  PORTB = 0b01010101;     // Binary combination on port B
  TRISB = 0;              // Port B pins are configured as outputs
}
```

To make this example more interesting, we will enable LEDs connected to the port B 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. It's only needed to set delay expressed in milliseconds.
2. After one second, the program enters the *for* loop and remains there as long as the variable k is less than 20. The variable is incremented by 1 after each iteration. Within the *for* loop, the *switch* operation monitors port B logic state. If PORTB=0xFF, its state is inverted into 0x00 and vice versa. Any change of these logic states causes all LEDs to blink. Duty Cycle is 5:1 (500mS:100mS).
3. When the program exits the *for* loop, the port B logic state changes (0xb 01010101) and the program enters the endless *while* loop and remains there as long as 1=1. The port B logic state is inverted each 200mS.



```
/*  Header ****************************************************************/

int k;
void main() {
  ANSEL  = 0;          // All I/O pins are configured as digital
  ANSELH = 0;
  PORTB = 0xFF;        // Reset port B
  TRISB = 0;           // Port B pins are configured as outputs

  Delay_ms(1000);      // 1s delay

  PORTB = 0;

for(k=1; k<20; k++)    // Remain in the loop as long as 1<k<20, k is incremented
{                      // by 1 after each iteration

    switch (PORTB) {           // Switch operator monitors port B state

    case 0x00: PORTB = 0xFF; // If PORTB=0, change its state into 0xFF
    Delay_ms(100);             // and provide 100mS delay
    break;

    case 0xFF: PORTB = 0x00; // If PORTB=0xFF, change its state into 0
    Delay_ms(500);    }        // and provide 500mS delay
}                              // End of for loop

ORTB = 0b01010101;             // Binary combination on port B

hile(1) {                      // endless loop
    PORTB =   ~PORTB;          // Invert port B logic state
    Delay_ms(200);             // 200mS delay
  }
}
```

4.4 EXAMPLE 2

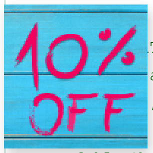
Using assembly instructions and internal oscillator LFINTOSC...

This is actually a sequel to the previous example, but deals with a bit more complicated problem... The idea is to make LED diodes on the port B blink slowly. It can be done by setting large value for delay parameter in the *Delay* function. But there is also another, more efficient manner to make this happen. You remember that this microcontroller has built-in oscillator LFINTOSC which operates at the frequency of 31kHz? Now, it’s time to ‘give it a chance’.

The program starts with the *do-while* loop and remains there for 20 cycles. After each iteration, 100mS delay is provided, which is reflected as relatively fast LED blinking of the port B. When the program exits this loop, the microcontroller starts using the LFINTOSC oscillator as a clock signal source. The LED blinking is considerably slower now even though the program executes the same *do-while* loop with 10 times shorter delay.

For the purpose of making some potentionally dangerous situation more obvious here, control bits are activated by assembly instructions. Simply put, when entering or exiting the assembly instruction in the program, the compiler doesn’t save data on currently active RAM bank, which means that in this program section, bank selection depends on the SFR registers in use. When switching back to the program section written in C, the control bits RP0 and RP1 must return the state they had before ‘assembly language adventure’. In this program, the problem is solved by using the *saveBank* auxiliary variable which saves the state of these two bits.

```
/* Header *********************************************/

int k = 0;
char saveBank;

void main() {
    ANSEL = 0;                      // All I/O pins are configured as digital
    ANSELH = 0;
    PORTB = 0;                      // All port B pins are set to 0
    TRISB = 0;                      // Port B pins are configured as outputs
    do {
        PORTB = ~PORTB;             // Invert port B logic state
        Delay_ms(100);              // 100mS delay
        k++;                        // Increment k by 1
    }
    while(k<20);                    // Remain in loop while k<20

    k=0;                            // Reset variable k
    saveBank = STATUS & 0b01100000; // 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 the OSCCON
        bcf STATUS,RP1              // register
        bcf OSCCON,6                // Select internal oscillator LFINTOSC
        bcf OSCCON,5                // of 31KHz frequency
        bcf OSCCON,4
        bsf OSCCON,0                // Microcontroller uses internal oscillator
    }                               // End of assembly sequence

    STATUS &= 0b10011111;           // Bits RP0 and RP1 return their original state
    STATUS |= saveBank;


           TB = ~PORTB;             // Invert port B logic state
           ay_ms(10);               // 10 mS delay
           ;                        // Increment k by 1

    while(k<20);                    // Remain in loop while k<20
}
```



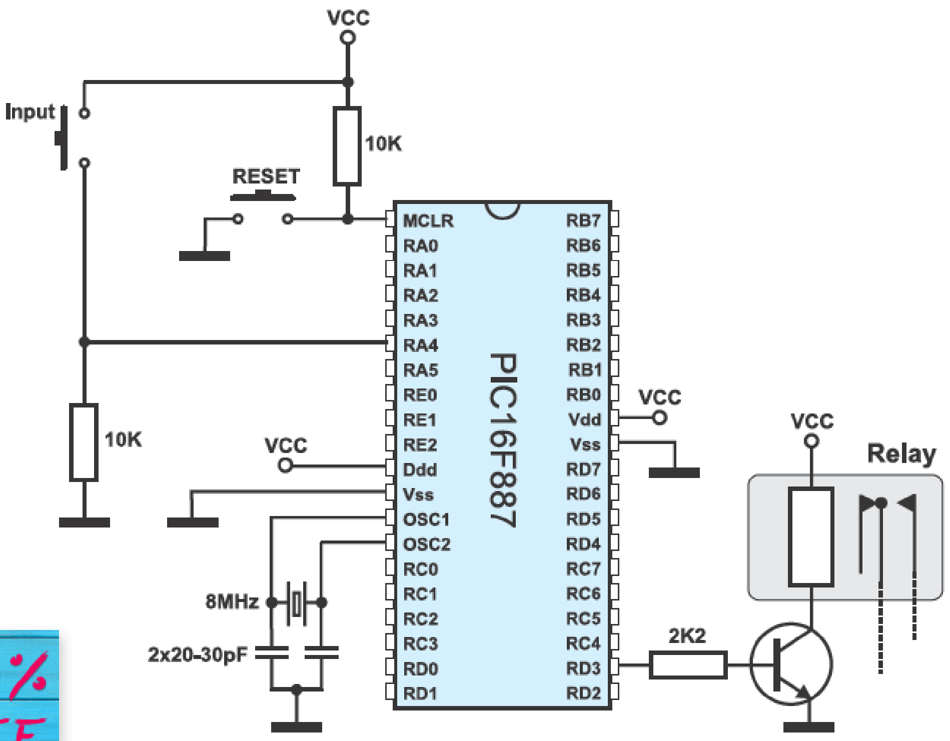
You have noticed that the clock signal source is changed ‘on the fly’. If you want to make sure of it, remove quartz crystal prior to switching the microcontroller on. The microcontroller will not start to operate because the *Config Word* loaded with the program requires the use of crystal on switching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

4.5 EXAMPLE 3

TMR0 as a counter, declaring new variables, enumerated constants, using relay ...

Referring to the previous examples, the microcontroller executes the program without being affected in any way by its surrounding. Practically, devices operating in this manner are very rare (for example, simple neon sign controller). Input pins are also used in this example. There is a schematic 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 so that any button press causes timer TMR0 to count one pulse. When the number of pulses matches the number stored in the TEST register, a logic one (5V) appears on the pin PORTD.3. This voltage activates an electromechanical relay, and this bit is called ‘RELAY’ in the program, therefore.

In this example, 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 activate some other device instead of relay, while the 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 needed, then something else should be turned on or off....



```
/*Header******************************************************/

void main() {
    char TEST = 5;              // Constant TEST = 5
    enum outputs {RELAY = 3}; // Constant RELAY = 3

    ANSEL = 0;                  // All I/O pins are configured as digital
    ANSELH = 0;
    PORTA = 0;                  // Reset port A
    TRISA = 0xFF;               // All portA pins are configured as inputs
    PORTD = 0;                  // Reset port D
    TRISD = 0b11110111;         // Pin RD3 is configured as an output, while the rest are
                                // configured as inputs

    OPTION_REG.F5 = 1;          // Counter TMR0 receives pulses through the RA4 pin
    OPTION_REG.F3 = 1;          // Prescaler rate is 1:1

    TMR0 = 0;                   // Reset timer/counter TMR0

    do {
        if (TMR0 == TEST)       // Does the number in timer match constant TEST?
        (PORTD.RELAY = 1);      // Numbers match. Set the RD3 bit (output RELAY)
    }
    while (1);                  // Remain in endless loop
}
```

Only one enumerated constant RELAY is used in this example. It is assigned a value when declared.

```
enum outputs {RELAY = 3}; // Constant RELAY = 3
```

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

```
enum outputs {RELE = 3, HEATER, MOTOR = 6, PUMP};
```

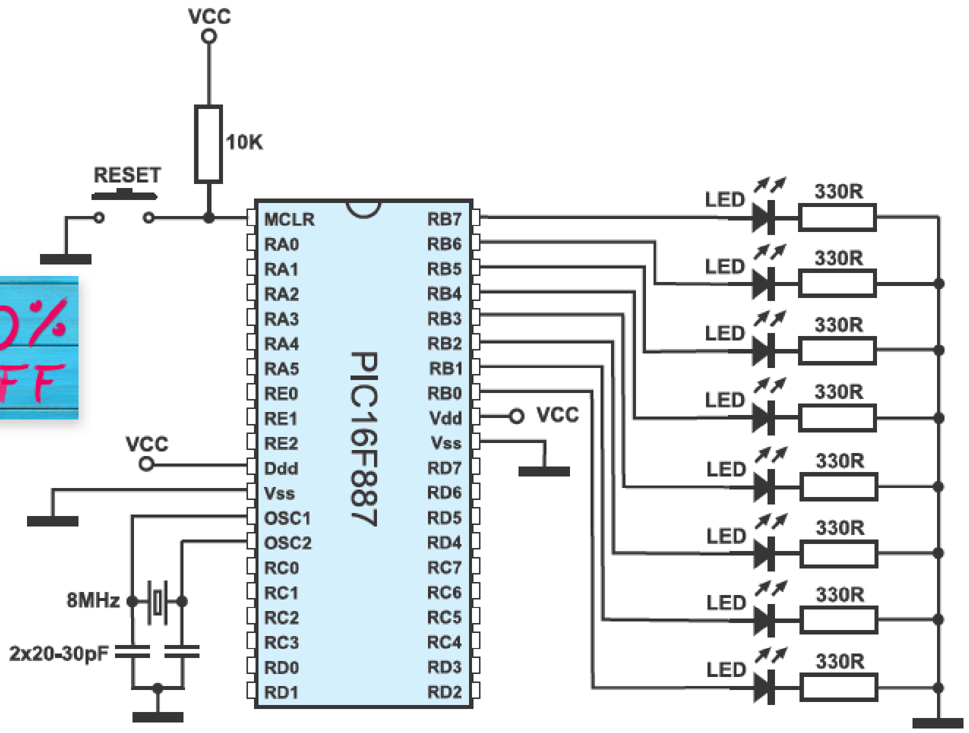
All the constants, following those with assigned values (RELAY = 3 and MOTOR = 6), are automatically assigned values incremented by 1 relative to the previous constant value. In this example, constants HEATER and PUMP will be assigned values 4 and 7, respectively (HEATER = 4 and PUMP = 7).

4.6 EXAMPLE 4

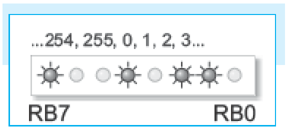
Using timers TMR0, TMR1 and TMR2. Using interrupts, declaring new function...

If you have read the previous example, you probably have noticed a disadvantage of providing delays using loops. In all those cases, the microcontroller is 'captive' and does nothing. It simply waits for some time to pass. Such waste of time is an unacceptable luxury and some other method should be applied therefore.

Do you remember the story about timers? Interrupts? This example makes links between them in a practical way. The schematic is still the same as well as the challenge. It is necessary to provide a 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 automatically increments the **cnt** variable by 1. When it's value reaches 400, the port B is incremented by 1. The whole procedure is performed 'behind the scenes', which enables the microcontroller to do something else.

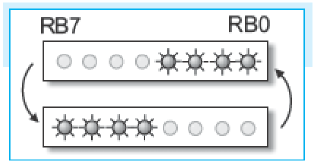


```
/*Header******************************************************/

unsigned cnt;                 // Define variable cnt

void interrupt() {
    cnt++;                    // 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
}

void main() {
    OPTION_REG = 0x84;        // Prescaler is assigned to timer TMR0
    ANSEL = 0;                // All I/O pins are configured as digital
    ANSELH = 0;
    TRISB = 0;                // All port B pins are configured as outputs
    PORTB = 0x0;              // Reset port B
    TMR0 = 96;                // Timer T0 counts from 96 to 255
    INTCON = 0xA0;            // Enable interrupt TMR0
    cnt = 0;                  // Variable cnt is assigned a 0

    do {                      // Endless loop
        if (cnt == 400) {     // Increment port B after 400 interrupts
            PORTB = PORTB++;  // Increment number on port B by 1
            cnt = 0;          // Reset variable cnt
        }
    } while(1);
}
```



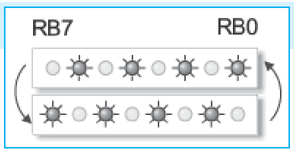
Interrupt occurs on every timer register TMR0 overflow.

```
/*Header******************************************************/

unsigned short cnt; // Define variable cnt

void interrupt() {
    cnt++ ;                     // 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
}

void main() {
    ANSEL = 0;                  // All I/O pins are configured as digital
    ANSELH = 0;
    PORTB = 0xF0;               // Initial value of port B bits
    [illegible] 0;              // Port B pins are configured as outputs
    [illegible] 1;              // Set timer TMR1
    [illegible]R1IF = 0;        // Reset bit TMR1IF
    [illegible] 0x80;           // Set initial value for timer TMR1
    [illegible] 0x00;
    [illegible]R1IE = 1;        // Enable interrupt on overflow
    cnt = 0;                    // Reset variable cnt
    INTCON = 0xC0;              // Enable interrupt (bits GIE and PEIE)

    do {                        // Endless loop
        if (cnt == 76) {        // Change port B state after 76 interrupts
            PORTB = ~PORTB;     // Number in port B is inverted
            cnt = 0;            // Reset variable cnt
        }
    } while (1);
}
```

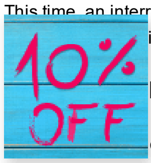


In this case, an interrupt is enabled after the timer register TMR1 (TMR1H, TMR1L) overflow occurs. Combination of bits changing on port B is different from that in the previous example.

```
/*Header******************************************************/

unsigned short cnt;     // Define variable cnt

void Replace() {
    PORTB = ~PORTB;     // Define new function 'Replace'
}                       // Function inverts port state

void interrupt() {
    if (PIR1.TMR2IF) {  // If bit TMR2IF = 1,
        cnt++ ;         // Increment variable cnt by 1
        PIR1.TMR2IF = 0;// Reset bit and
        TMR2 = 0;       // reset register TMR2
    }
}

// main
void main() {
    cnt = 0;            // Reset variable cnt
    ANSEL = 0;          // All I/O pins are configured as digital
    ANSELH = 0;
    PORTB = 0b10101010; // Logic state on port B pins
    TRISB = 0;          // All port B 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) { // Change PORTB after more than 30 interrupts
            Replace();  // Function Replace inverts the port B state
            cnt = 0;    // Reset variable cnt
        }
    }
}
```



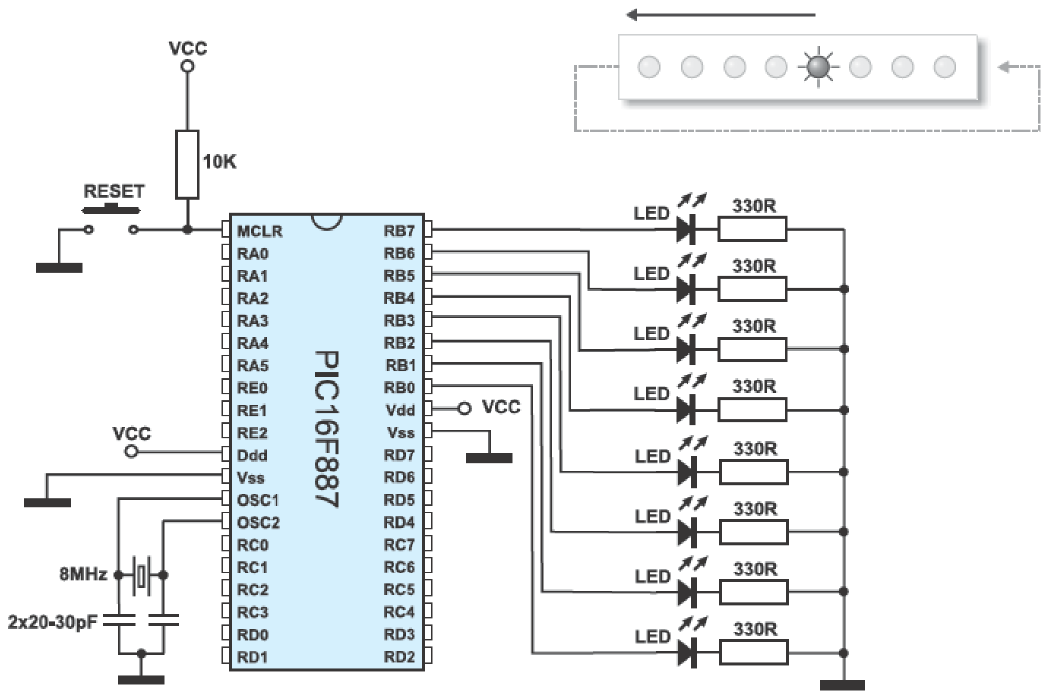
This time, an interrupt occurs after timer register TMR2 overflow occurs. The *Replace* function, which normally doesn't belong to C, is used invert port pins state.



E 5

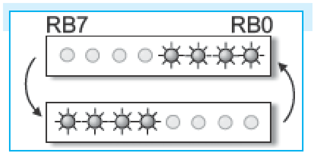
og timer

This example illustrates how the watch-dog timer should not be used. A command used for resetting 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 all the time, which is reflected as PORTB LED blinking.



```
/*Header******************************************************/

void main() {
    OPTION_REG = 0x0E; // Prescaler is assigned to timer WDT (1:64)

    asm CLRWDT;        // Assembly command to reset WDT timer

    PORTB = 0x0F;      // Initial value of the PORTB register
    TRISB = 0;         // All port B pins are configured as outputs
    Delay_ms(300);      // 30mS delay
    PORTB = 0xF0;       // Porta B value different from initial

    while (1);          // Endless loop. Program remains here until WDT
                        // timer resets the microcontroller
}
```

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 PWM signal generator

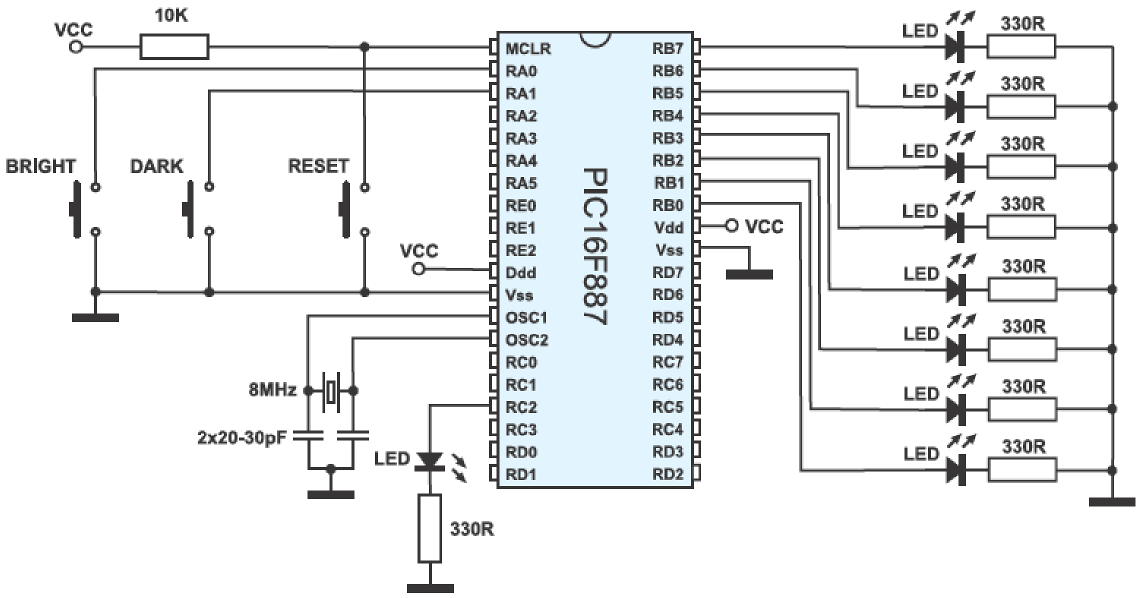
This example illustrates the use of CCP1 module in PWM mode. To make things more interesting, the duration of the P1A output pulses (PORTC,2) may be changed using pushbuttons symbolically marked as 'DARK' and 'BRIGHT', while the set duration is seen as binary combination on port B. The operation of this module is under control of the functions belonging to the specialized *PWM Library*. Three of

e:

has the prototype: `void Pwm1_Init(long freq);`
eq sets the frequency of PWM signal expressed in herz. In this example it amounts to 5kHz.
t has the prototype: `void Pwm1_Start(void);`

3. **PWM1_Set_Duty** has the prototype: `void Pwm1_Set_Duty(unsigned short duty_ratio);`
   Parameter duty_ratio sets pulse duration in pulse sequence.

The PWM library also contains the **PWM_Stop** function used to disable this mode. Its prototype is: `void Pwm1_Stop(void);`



```
/*Header******************************************************/

unsigned short current_duty, old_duty;    // Define variables
                                          // current_duty and old_duty

void initMain() {
    ANSEL = 0;                            // All I/O pins are configured as digital
    ANSELH = 0;
    PORTA = 255;                          // Port A initial state
    TRISA = 255;                          // All port A pins are configured as inputs
    PORTB = 0;                            // Initial state of port B
    TRISB = 0;                            // All port B pins are configured as outputs
    PORTC = 0;                            // Port C initial state
    TRISC = 0;                     // All port C pins are configured as outputs
    PWM1_Init(5000);               // PWM module initialization (5KHz)
}

void main() {
    initMain();
    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 (Button(&PORTA, 0,1,1))        // If the button connected to RA0 is pressed
            current_duty++ ;              // increment variable current_duty

        if (Button(&PORTA, 1,1,1))        // If the pressed button is connected to RA1
            current_duty-- ;              // decrement value current_duty

        if (old_duty != current_duty) {   // 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 port B
        }

            ay_ms(200);                   // 200mS delay
```

is example work properly, it is necessary to tick off the following libraries in the *Library Manager* prior to compiling:

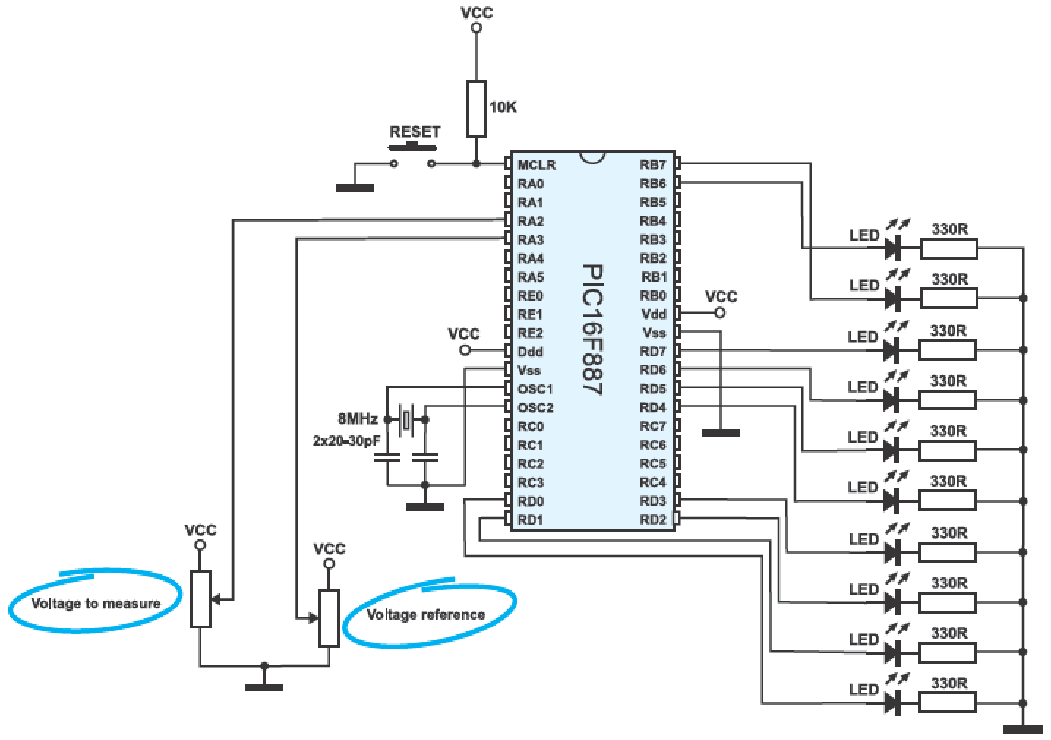
- PWM
- Button

4.9 EXAMPLE 7

Using A/D converter

The PIC16F887 A/D converter is used in this example. Is it necessary to mention that everything is rather simple?! A variable analog signal is applied to the AN2 pin, while the 10-bit result of conversion is shown on ports B and D (8 LSBs on port D and 2 MSBs on port B). GND is used as negative voltage reference Vref-, while positive voltage reference is applied to the AN3 pin. It enables voltage measurement scale to 'stretch and shrink'.

In other words, the A/D converter always generates a 10-bit binary result, which means that it detects a total of 1024 voltage levels (210=1024). 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 seen, the A/D converter is able to detect slight changes in voltage.



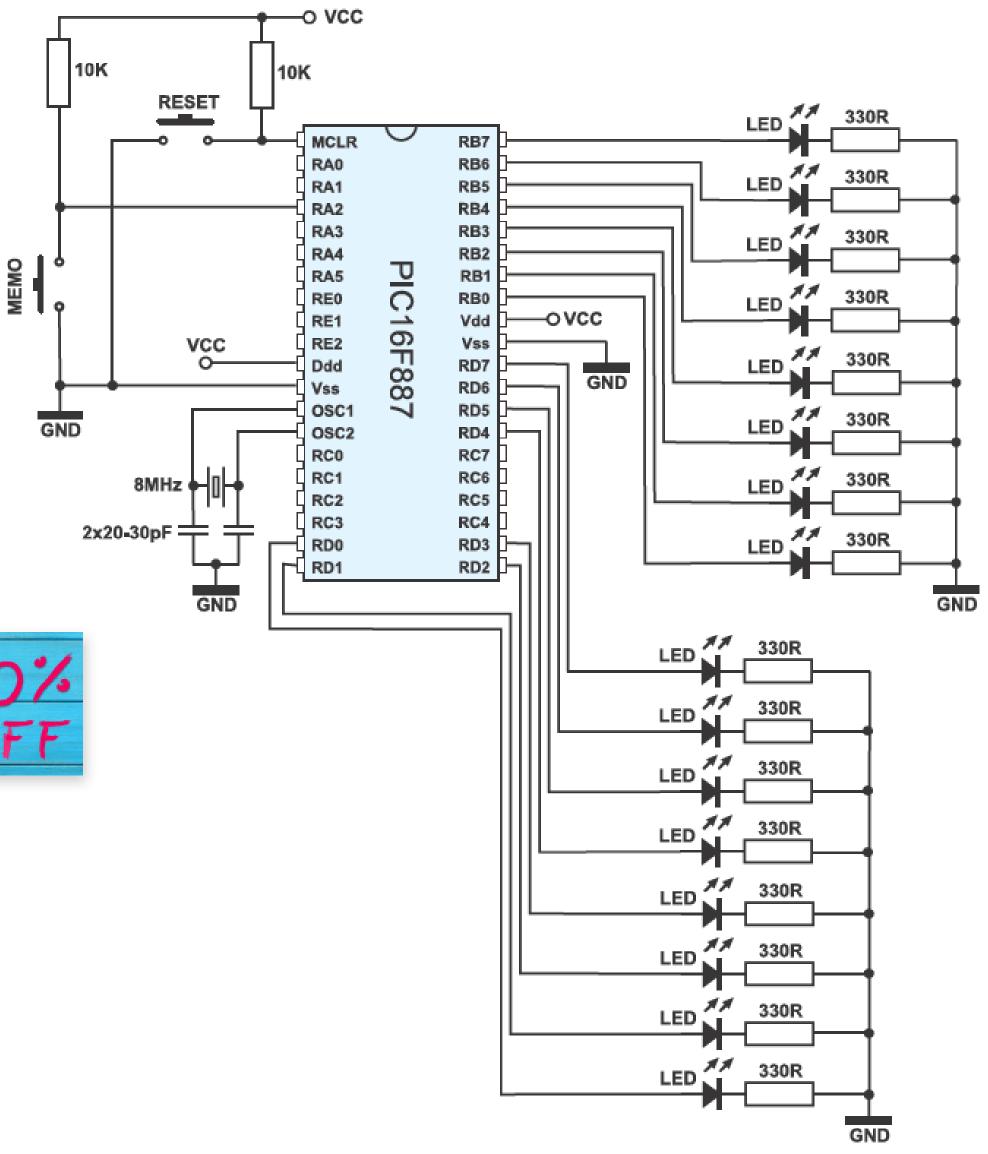
```
/*Header******************************************************/

unsigned int temp_res;

void main() {
    ANSEL = 0x0C;                // Pins AN2 and AN3 are configured as analog
    TRISA = 0xFF;                // All port A pins are configured as inputs
    ANSELH = 0;                  // Rest of pins is configured as digital
    TRISB = 0x3F;                // Port B pins RB7 and RB6 are configured as
                                 // outputs
    TRISD = 0;                   // All port D pins are configured as outputs
    ADCON1.F4 = 1 ;              // Voltage reference is brought to the RA3 pin.

    do {
        temp_res = ADC_Read(2); // Result of A/D conversion is copied to temp_res
        PORTD = temp_res;        // 8 LSBs are moved to port D
        PORTB = temp_res >> 2;   // 2 MSBs are moved to bits RB6 and RB7
    } while(1);                  // Endless loop
}
```

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

LE 8

M Memory

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



```
/*Header******************************************************/

void main() {
    ANSEL = 0;                      // All I/O pins are configured as digital
    ANSELH = 0;

    PORTB = 0;                      // Port B initial value
    TRISB = 0;                      // All port B pins are configured as outputs
    PORTD = 0;                      // Port B initial value
    TRISD = 0;                      // All port D pins are configured as outputs
    TRISA = 0xFF;                   // All port A pins are configured as inputs
    PORTD = EEPROM_Read(5);         // Read EEPROM memory at address 5

    do {
        PORTB=PORTB++;              // Increment port B by 1
        Delay_ms(100);              // 100mS delay
        if (PORTA.F2){
            EEPROM_Write(5,PORTB);  // If MEMO is pressed, save PORTB
            PORTD = EEPROM_Read(5); // Read written data
            do;
            while (PORTA.F2);       // Remain in this loop as long as the button is
                                    // pressed
        }
    }
    while(1);                       // Endless loop
}
```

In order to check this circuit, it is sufficient to press the MEMO button and turn off the device. After restarting the device, the program will display the saved value on port D. Remember that at the moment of writing, this value was displayed on port B).

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

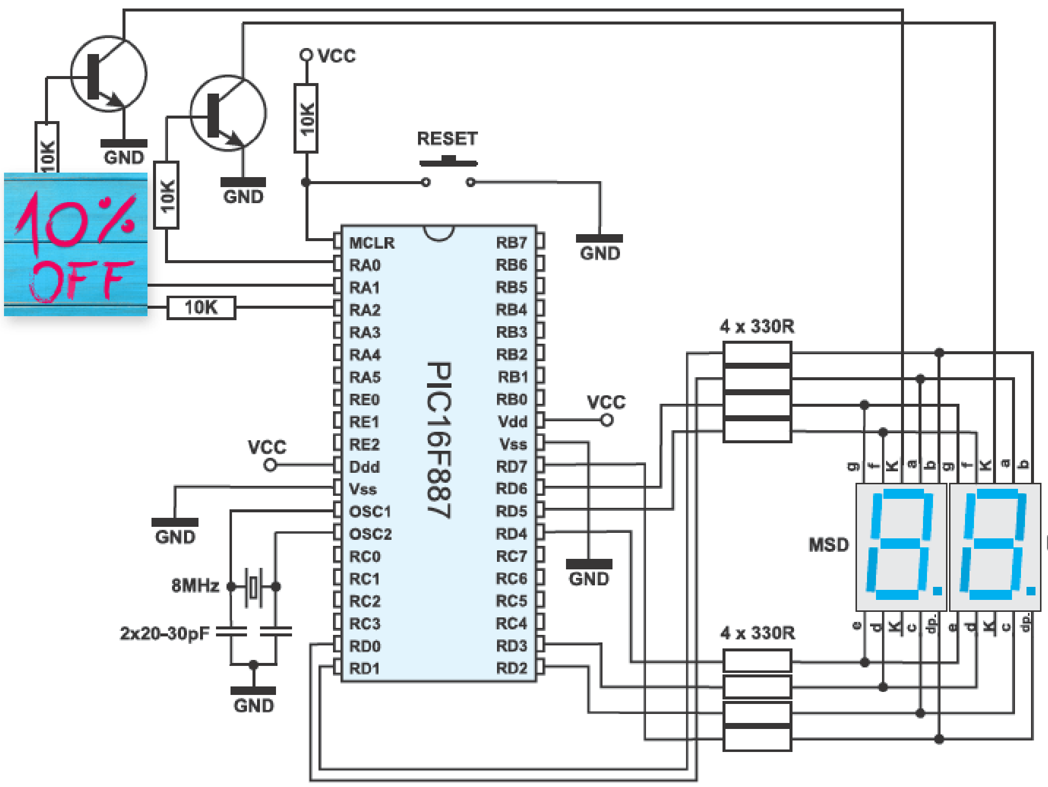
- EEPROM

4.11 EXAMPLE 9

Two-digit LED counter, multiplexing

The microcontroller operates as a two-digit counter here. The variable i is incremented (slow enough to be visible) and its value is displayed on a two-digit LED display (99-0). The challenge is to enable a binary number to be converted in decimal and split it in two digits (tens and ones). Since the LED display segments are connected in parallel, it is necessary to ensure that they change fast in order to make impression of simultaneous light emission (time-division multiplexing).

In this example, timer TMR0 is in charge of the time-division multiplexing, while the *mask* function converts a binary number into decimal format.



```
/*Header******************************************************/

unsigned short mask(unsigned short num);
unsigned short digit_no, digit10, digit1, digit, i;

void interrupt() {
    if (digit_no==0) {
        PORTA = 0;                  // Turn off both displays
        PORTD = digit1;             // Set mask for displaying ones on PORTD
        PORTA = 1;                  // Turn on display for ones (LSD)
        digit_no = 1;
    } else {
        PORTA = 0;                  // Turn off both displays
        PORTD = digit10;            // Set mask for displaying tens on PORTD
        PORTA = 2;                  // Turn on display for tens (MSD)
        digit_no = 0;
    }
    TMR0 = 0;                       // Reset counter TMRO
    INTCON = 0x20;                  // Bit T0IF=0, T0IE=1
}

void main() {
    OPTION_REG = 0x80;              // Set timer TMR0
    TMR0 = 0;
    INTCON = 0xA0;                  // Disable interrupt PEIE,INTE,RBIE,T0IE
    PORTA = 0;                      // Turn off both displays
    TRISA = 0;                      // All port A pins are configured as outputs
    PORTD = 0;                      // Turn off all display segments
    TRISD = 0;                      // All port D pins are configured as outputs

    do {
        for (i = 0; i<=99; i++) {   // Count from 0 to 99
            digit = i % 10u;
            digit1 = mask(digit);   // Prepare mask for displaying ones
            digit = (char)(i / 10u) % 10u;
            digit10 = mask(digit); // Prepare mask for displaying tens
            Delay_ms(1000);
        }
    } while (1);                    // Endless loop
}
```

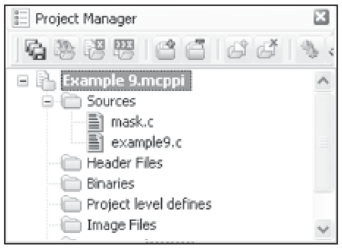
mask.c file:

```
/*Header******************************************************/
unsigned short mask(unsigned short num) {
           ) {
           turn 0x3F;
           turn 0x06;
           turn 0x5B;
           turn 0x4F;
case 4 : return 0x66;
case 5 : return 0x6D;
case 6 : return 0x7D;
case 7 : return 0x07;
case 8 : return 0x7F;
case 9 : return 0x6F;
}
}
```

In order to make this example work properly, it is necessary to include document *mask.c* into the project prior to compiling:

Example9.mcppi - *Sources* - **Add File To Project**

- mask.c
- example9.c



4.12 EXAMPLE 10

Using LCD display

This example illustrates the use of an alphanumeric LCD display. The function libraries simplify this program, which means that the effort made to create software pays off in the end.

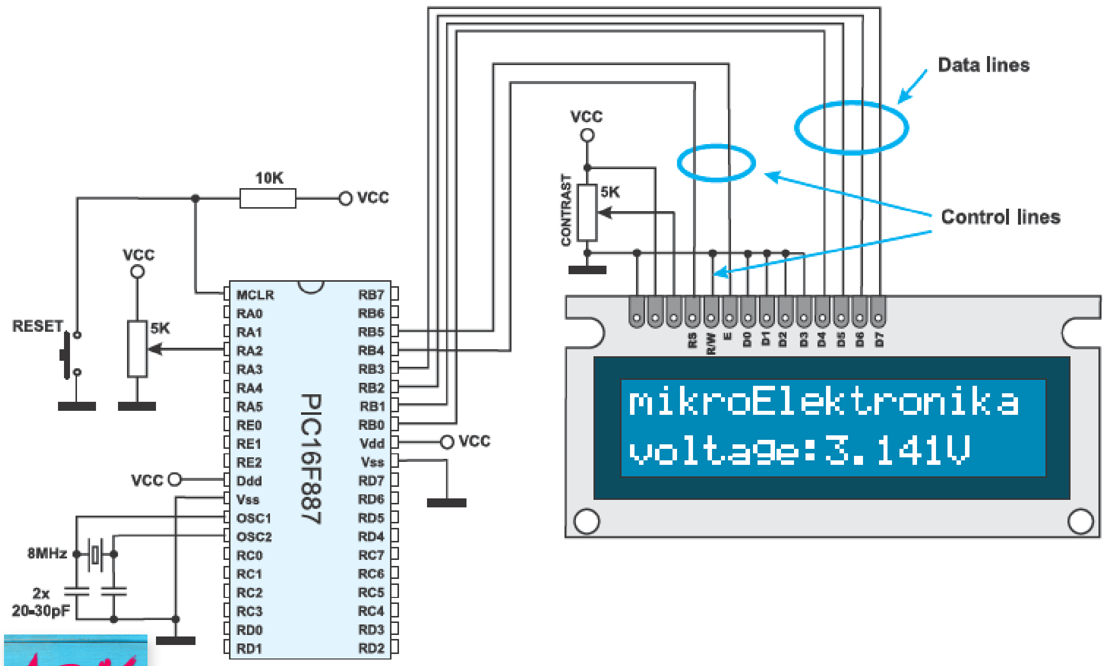
A message written in two lines appears on the display:

mikroElektronika
LCD example

Two seconds later, the message in the second line is changed and displays voltage present on the A/D converter input (the RA2 pin). For example:

mikroElektronika
voltage:3.141V

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



Data lines
Control lines
VCC
CONTRAST
5K
10K
VCC
RESET
5K
VCC
8MHz
2x
20-30pF
PIC16F887
MCLR RA0 RA1 RA2 RA3 RA4 RA5 RE0 RE1 RE2 Ddd Vss OSC1 OSC2 RC0 RC1 RC2 RC3 RD0 RD1
RB7 RB6 RB5 RB4 RB3 RB2 RB1 RB0 Vdd Vss RD7 RD6 RD5 RD4 RC7 RC6 RC5 RC4 RD3 RD2
RS R/W E D0 D1 D2 D3 D4 D5 D6 D7
mikroElektronika
voltage:3.141V

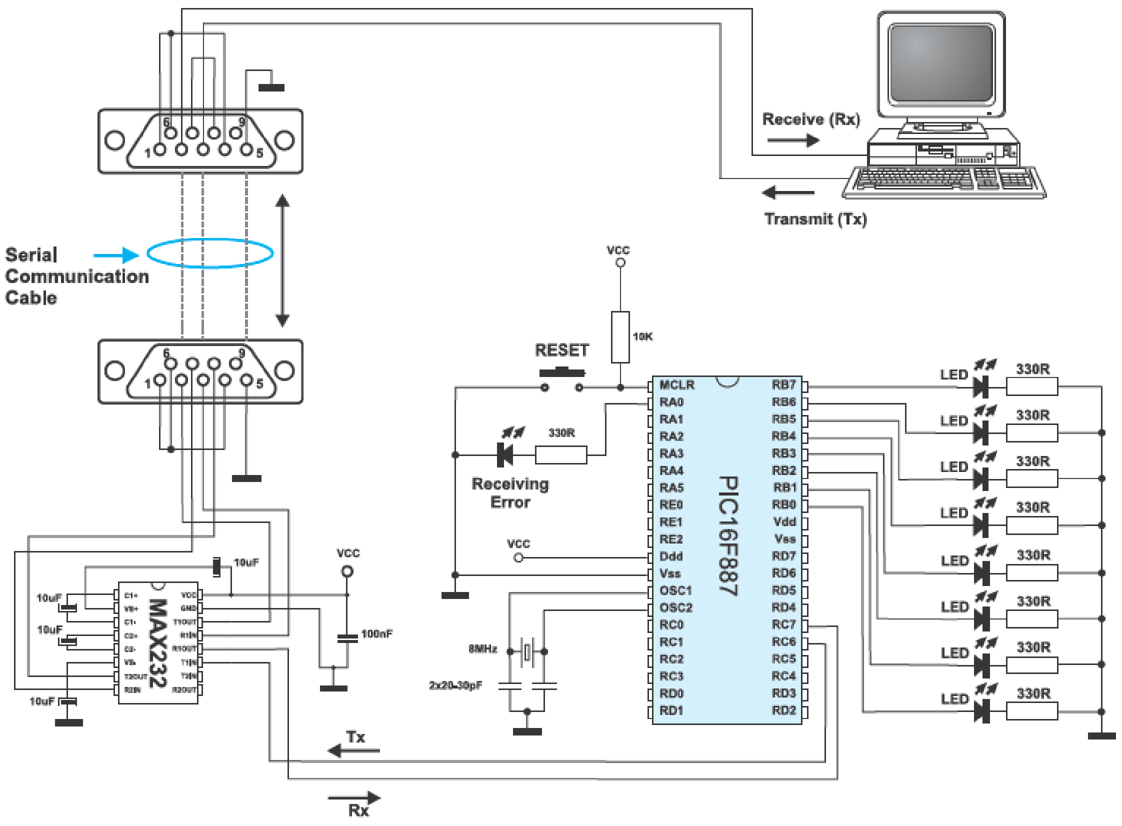
is example work properly, it is necessary to tick off the following libraries in the *Library Manager* prior to compiling:

```
/*Header******************************************************/

// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;
sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

unsigned char ch;                    //
unsigned int adc_rd;                 // Declare variables
char *text;                          //
long tlong;                          //

void main() {
    INTCON = 0;                      // All interrupts disabled
    ANSEL = 0x04;                    // Pin RA2 is configured as an analog input
    TRISA = 0x04;
    ANSELH = 0;                      // Rest of pins are 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);               // Define the first message

    ADCON1 = 0x82;                   // A/D voltage reference is VCC
    TRISA = 0xFF;                    // All port A pins are configured as inputs
    Delay_ms(2000);
    text = "voltage:";               // Define the third message

    while (1) {
        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 = (long)adc_rd * 5000; // Convert the result in millivolts
        tlong = tlong / 1023;        // 0..1023 -> 0-5000mV
        ch = tlong / 1000;           // Extract volts (thousands of millivolts)
                                     // from result
        Lcd_Chr(2,9,48+ch);          // Write result in ASCII format
        Lcd_Chr_CP('.');
        ch = (tlong / 100) % 10;     // Extract hundreds of millivolts
        Lcd_Chr_CP(48+ch);           // Write result in ASCII format
        ch = (tlong / 10) % 10;      // Extract tens of millivolts
        Lcd_Chr_CP(48+ch);           // Write result in ASCII format
        ch = tlong % 10;             // Extract digits for millivolts
        Lcd_Chr_CP(48+ch);           // Write result in ASCII format
        Lcd_Chr_CP('V');
        Delay_ms(1);
    }
}
```

LE 11

ommunication

trates the use of the microcontroller's EUSART module. Connection to a PC is enabled through the RS232 standard. The program works in the following manner. Every byte received via serial communication is displayed using LED diodes connected to port B and is automatically returned to the transmitter after that. If an error occurs on receive, it will be signalled by switching the LED diode on. The easiest way to test the device operation practically is by using a standard Windows program called *Hyper Terminal*.



```
/*Header******************************************************/

unsigned short i;

void main() {
    UART1_Init(19200);            // Initialize USART module
```

```
                               // (8 bit, 19200 baud rate, no parity bit...)
    while (1) {
        if (UART1_Data_Ready()) { // If data has been received
            i = UART1_Read();     // read it
            UART1_Write(i);       // and send it back
        }
    }
}
```

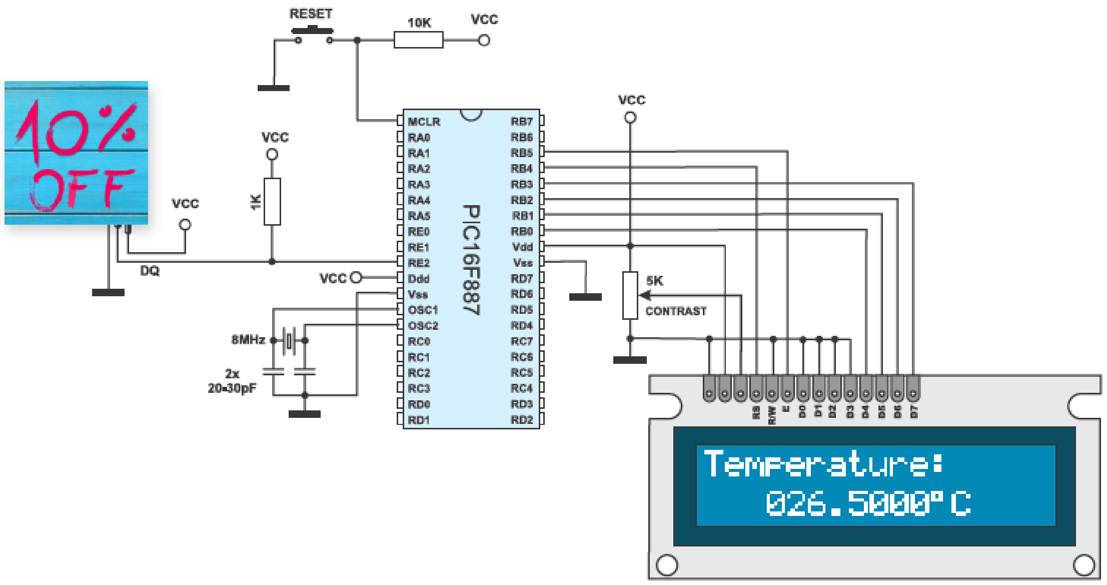
In order to make this example work properly, it is necessary to tick off the UART library in the *Library Manager* prior to compiling:

- UART

4.14 EXAMPLE 12

Temperature measurement using DS1820 sensor. Use of '1-wire' protocol...

Temperature measurement is one of the most common tasks performed by the microcontroller. A DS1820 sensor is used for measurement here. It is capable of measuring temperature in the range of -55 °C to 125 °C with 0.5 °C accuracy. For the purpose of transferring data to the microcontroller, a special type of serial communication called *1-wire* is used.



Due to a simple and wide use of these sensors, commands used to run and control them are in the form of functions stored in the One_Wire library. There are three functions in total:

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

This example implies the advantage in using libraries with ready-to-use functions. Concretely, you don't have to study documentation provided by the manufacturer in order to use this sensor. It is sufficient to copy some of these functions in the program. If you want to know how any of them is declared, just right click on it and select the *Help* option.

```
/*Header*************************************************************/

// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;
sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

const unsigned short TEMP_RESOLUTION = 9;
```

```
char *text = "000.0000";
unsigned temp;

void Display_Temperature(unsigned int temp2write) {
    const unsigned short RES_SHIFT = TEMP_RESOLUTION - 8;
    char temp_whole;
    unsigned int temp_fraction;

    // check if temperature is negative
    if (temp2write & 0x8000) {
        text[0] = '-';
        temp2write = ~temp2write + 1;
    }
    // extract temp_whole
    temp_whole = temp2write >> RES_SHIFT ;

    // convert temp_whole to characters
    if (temp_whole/100)
        text[0] = temp_whole/100 + 48;
    else
        text[0] = '0';

    text[1] = (temp_whole/10)%10 + 48; // Extract tens digit
            = temp_whole%10 + 48;      // Extract ones digit

           act temp_fraction and convert it to unsigned int
           action = temp2write << (4-RES_SHIFT);
           action &= 0x000F;
    temp_fraction *= 625;

    // convert temp_fraction to characters
    text[4] = temp_fraction/1000 + 48;     // Extract thousands digit
    text[5] = (temp_fraction/100)%10 + 48; // Extract hundreds digit
    text[6] = (temp_fraction/10)%10 + 48;  // Extract tens digit
    text[7] = temp_fraction%10 + 48;       // Extract ones digit

    // Display temperature on LCD
    Lcd_Out(2, 5, text);
}

void main() {
    ANSEL = 0;    // Configure AN pins as digital I/O
    ANSELH = 0;
    C1ON_bit = 0; // Disable comparators
    C2ON_bit = 0;

    Lcd_Init();   // Initialize LCD
    Lcd_Cmd(_LCD_CLEAR);      // Clear LCD
    Lcd_Cmd(_LCD_CURSOR_OFF); // Turn the cursor off
    Lcd_Out(1, 1, " Temperature: ");

    // Print degree character, 'C' for Centigrades
    Lcd_Chr(2,13,223);        // different LCD displays have different char code for degree
    // if you see greek alpha letter try typing 178 instead of 223

    Lcd_Chr(2,14,'C');

    //--- main loop
    do {
        //--- perform temperature reading
        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;
```

```
        //--- Format and display result on Lcd
        Display_Temperature(temp);
        Delay_ms(500);
    } while (1);
}
```

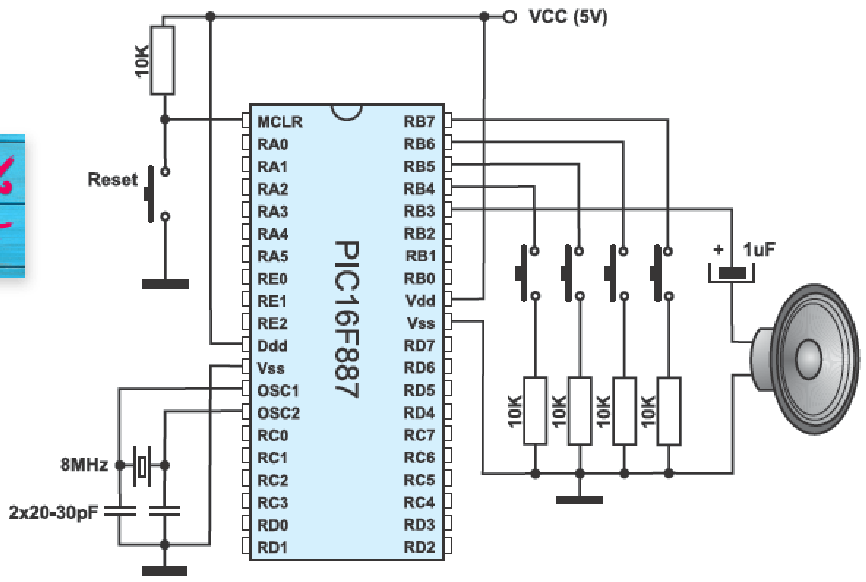
In order to make this example work properly, it is necessary to tick off the following libraries in the Library Manager prior to compiling:

- 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, confirm that some of the push buttons is pressed, warn that minimum or maximum values are reached etc. It can be just a 'beep' signal as well as longer or shorter melody. This example demonstrates sound generation using functions belonging to the *Sound* library.



In adition to these functions, the *Button* function belonging to the same library is used for testing push buttons.

```
/*Header******************************************************/

void Tone1() {
    Sound_Play(659, 250);          // Frequency = 659Hz, duration = 250ms
}

void Tone2() {
    Sound_Play(698, 250);          // Frequency = 698Hz, duration = 250ms
}

void Tone3() {
    Sound_Play(784, 250);          // Frequency = 784Hz, duration = 250ms
}

void Melody1() {                   // Make funny melody 1
    Tone1(); Tone2(); Tone3(); Tone3();
    Tone1(); Tone2(); Tone3(); Tone3();
    Tone1(); Tone2(); Tone3();
    Tone1(); Tone2(); Tone3(); Tone3();
    Tone1(); Tone2(); Tone3();
    Tone3(); Tone3(); Tone2(); Tone2(); Tone1();
}

void ToneA() {                     // Tone A
    Sound_Play(880, 50);
}

void ToneC() {                     // Tone C
```

```
    Sound_Play(1046, 50);
}

void ToneE() {                    // Tone E
    Sound_Play(1318, 50);
}

void Melody2() {                  // Make funny melody 2
    unsigned short i;
    for (i = 9; i > 0; i--) {
        ToneA(); ToneC(); ToneE();
    }
}

void main() {
    ANSEL = 0;                    // All I/O pins are digital
    ANSELH = 0;
    TRISB = 0xF0;                 // Pins RB7-RB4 are configured as inputs,
                                  // RB3 is configured as an output

    Sound_Init(&PORTB, 3);
    Sound_Play(1000, 500);

           1) {
           (Button(&PORTB,7,1,1)) // RB7 generates Tone1
            Tone1();
           le (PORTB & 0x80) ;    // Wait for push button release

        if (Button(&PORTB,6,1,1)) // RB6 generates Tone2
            Tone2();
        while (PORTB & 0x40) ;    // Wait for push button release

        if (Button(&PORTB,5,1,1)) // RB5 generates melody 2
            Melody2();
        while (PORTB & 0x20) ;    // Wait for push button release

        if (Button(&PORTB,4,1,1)) // RB4 generates melody 1
            Melody1();
        while (PORTB & 0x10) ;    // Wait for push button release
    }
}
```

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

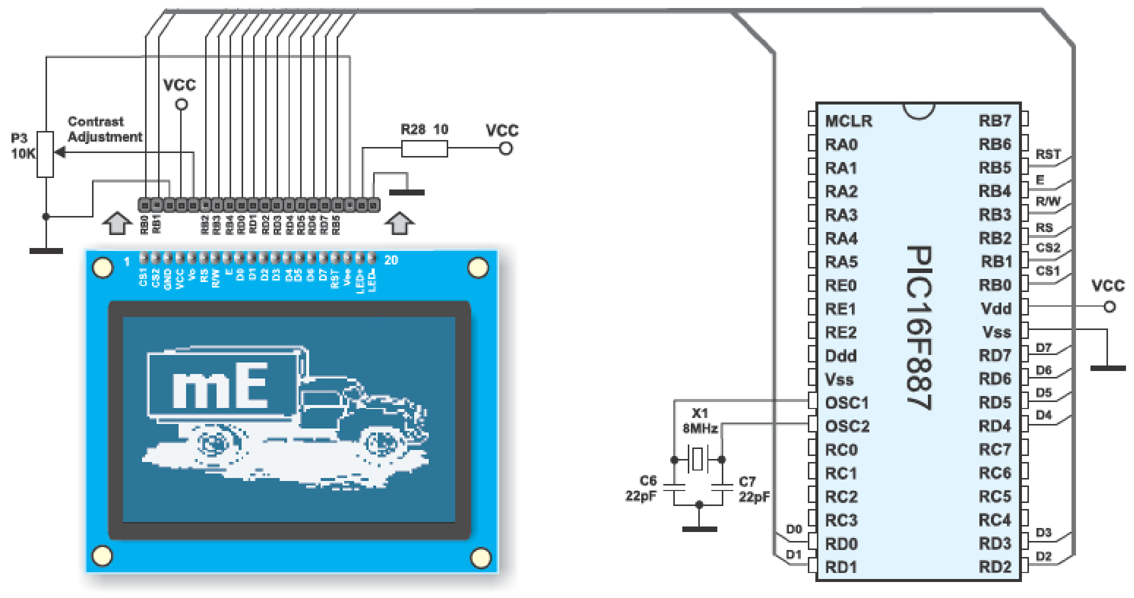
- Button
- Sound

4.16 EXAMPLE 14

Using 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 display messages in the form of drawings and bitmaps. The most commonly used graphic LCD has 128x64 pixels screen resolution. The GLCD contrast can be adjusted using the potentiometer P1.

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


VCC
Contrast Adjustment
P3 10K
R28 10
VCC
RB0 RB1 RB2 RB3 RB4 RD0 RD1 RD2 RD3 RD4 RD5 RD6 RD7 RB5
1
CS1 CS2 GND VCC Vo RS R/W E D0 D1 D2 D3 D4 D5 D6 D7 RST Vee LED+ LED-
20
mE
MCLR RA0 RA1 RA2 RA3 RA4 RA5 RE0 RE1 RE2 Ddd Vss OSC1 OSC2 RC0 RC1 RC2 RC3 RD0 RD1
RB7 RB6 RB5 RB4 RB3 RB2 RB1 RB0 Vdd Vss RD7 RD6 RD5 RD4 RC7 RC6 RC5 RC4 RD3 RD2
PIC16F887
RST E R/W RS CS2 CS1
VCC
D7 D6 D5 D4 D3 D2 D0 D1
X1 8MHz
C6 22pF
C7 22pF

```
*****************************************************/

ons---------------------------------------------------------------------
char truck_bmp[1024];
//--------------------------------------------------------------end-declarations

// Glcd module connections
char GLCD_DataPort at PORTD;
sbit GLCD_CS1 at RB0_bit;
sbit GLCD_CS2 at RB1_bit;
sbit GLCD_RS at RB2_bit;
sbit GLCD_RW at RB3_bit;
sbit GLCD_EN at RB4_bit;
sbit GLCD_RST at RB5_bit;
sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;
// End Glcd module connections

void delay2S(){                                        // 2 second delay function
    Delay_ms(2000);
}

void main() {
    unsigned short ii;
    char *someText;

    #define COMPLETE_EXAMPLE
    ANSEL = 0;                                         // Configure AN pins as digital
    ANSELH = 0;
    C1ON_bit = 0;                                      // Disable comparators
    C2ON_bit = 0;
    Glcd_Init();                                       // Initialize GLCD
    Glcd_Fill(0x00);                                   // Clear GLCD

    while(1) {
        #ifdef COMPLETE_EXAMPLE
            Glcd_Image(truck_bmp);                     // Draw image
            delay2S(); delay2S();
        #endif

        Glcd_Fill(0x00);                               // Clear GLCD
        Glcd_Box(62,40,124,56,1);                      // Draw box
        Glcd_Rectangle(5,5,84,35,1);                   // Draw rectangle
        Glcd_Line(0, 0, 127, 63, 1);                   // Draw line
        delay2S();

        for(ii = 5; ii < 60; ii+=5 ){                  // Draw horizontal and vertical lines
            Delay_ms(250);
            Glcd_V_Line(2, 54, ii, 1);
            Glcd_H_Line(2, 120, ii, 1);
        }
        delay2S();

        Glcd_Fill(0x00);                               // Clear GLCD

        #ifdef COMPLETE_EXAMPLE
            Glcd_Set_Font(Character8x7, 8, 7, 32);     // Choose font, see __Lib_GLCDFonts.c
                                                       // in Uses folder
        #endif

        Glcd_Write_Text("mikroE", 1, 7, 2);            // Write string

        for (ii = 1; ii <= 10; ii++)                   // Draw circles
            Glcd_Circle(63,32, 3*ii, 1);
        delay2S();
```

```
           d_Box(12,20, 70,57, 2);                     // Draw box
           ay2S();

           def COMPLETE_EXAMPLE
            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();
        #endif
    }
}
```

truck_bmp.c file:

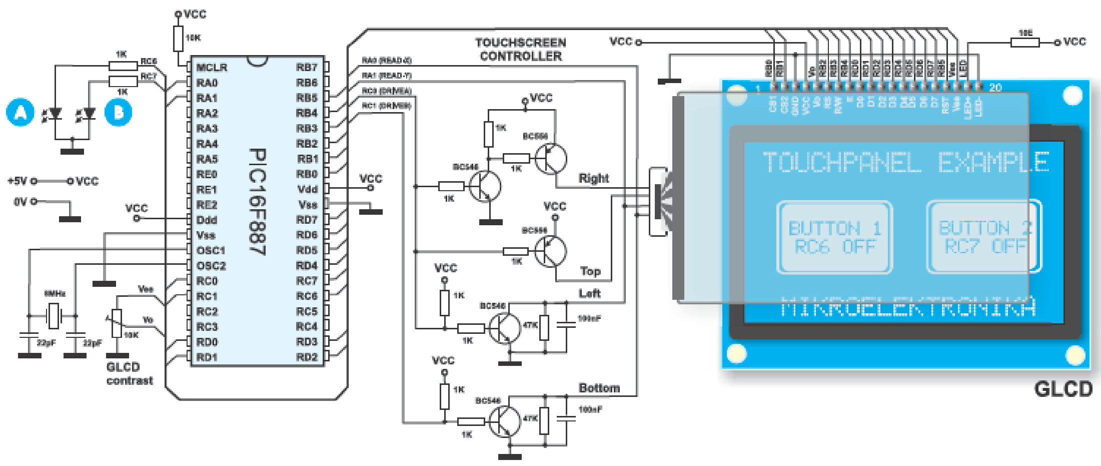
```
// ------------------------------------------------------
// GLCD Picture name: truck.bmp
// GLCD Model: KS0108 128x64
// ------------------------------------------------------

unsigned char const truck_bmp[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,
 137,137,137,137,137,137,137,  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,  0,255,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 240,240,240,240,240,224,224,240,240,240,240,240,224,192,192,224,
 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, 32, 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,  0,255,  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,  0,255,255,255,255,255,129,
 129,129,129,129,129,129,128,  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, 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, 96,
  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, 62, 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,
               7,127,127,127,255,255,247,251,123,191, 95, 93,125,189,
               3, 89,177,115,243,229,207, 27, 63,119,255,207,191,255,
               5,255,255,255,255,127,127,127,127,127,127,127,127,255,
               7,127,125,120,120,120,120,120,248,120,120,120,120,120,
 120,248,248,232,143,  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, 96,192,
 143,168,216,136, 49, 68, 72, 50,160, 96,  0,  0,  0,  0,  0,  0,
   0,  0,  0,128,192,248,248,248,248,252,254,254,254,254,254,254,
 254,254,254,254,254,255,255,255,255,255,246,239,208,246,174,173,
 169,128,209,208,224,247,249,255,255,252,220,240,127,255,223,255,
 255,255,255,255,255,254,254,255,255,255,255,255,255,255,254,255,
 255,255,255,255,255,255,254,254,254,254,254,254,254,254,254,254,
 254,254,254,254,255,255,255,255,255,255,254,255,190,255,255,253,
 240,239,221,223,254,168,136,170,196,208,228,230,248,127,126,156,
 223,226,242,242,242,242,242,177, 32,  0,  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, 47,111,239,255,253,253,255,254,255,255,255,255,255,
 191,191,239,239,239,191,255,191,255,255,255,255,255,255,255,255,
 255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
 255,255,255,255,127,127,127,127,255,255,191,191,191,191,255,254,
 255,253,255,255,255,251,255,255,255,127,125, 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,  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
};
```

In order to make this example work properly, it is necessary to tick off the GLCD library in the *Library Manager* prior to compiling. Also, it is necessary to include document *truck_bmp.c* into the project.

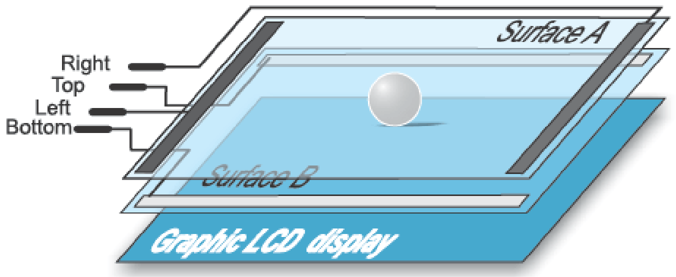
4.17 EXAMPLE 15

Using 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 output signal. There are a few types of touch panel. The simplest one is the resistive touch panel.

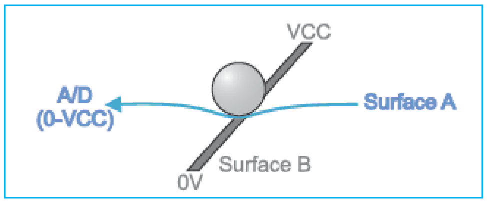


It consists of two transparent rigid foils, forming a ‘sandwich’ structure, that have resistive layers on their inner sides. The resistance of these layers usually does not exceed 1K. The opposite sides of the foils have contacts available for use through a flat cable.



The process of determining coordinates of the point in which the touch panel is pressed can be broken up 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 obtained by pressing the touch panel. The value of the divider is read on the bottom contact of the *surface B*. Voltage can be in the range of 0V to the power supply 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 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 create a circuit for touch panel control. By means of this circuit, the microcontroller connects appropriate contacts of the touch panel to ground and the power supply (as described above) in order to determine the X and Y coordinates. 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 software consists of writing a menu on graphic LCD, turning the circuit for touch panel control on/off (driving touch panel) and reading the values of A/D converter which actually represent the X and Y coordinates of the point.

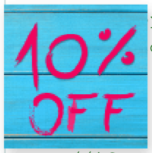
Once the coordinates are determined, it is possible to decide what we want the microcontroller to do. In this example, microcontroller turns on/off two digital pins, connected to LED diodes A and B.

This example use functions belonging to the *Glcd* and *ADC* library.

Considering that the touch panel surface is slightly larger than the surface of the graphic LCD, in case you want greater accuracy when determining the coordinates, it is necessary to perform the software calibration of the touch panel.

```
const char msg1[] = "TOUCHPANEL EXAMPLE";
const char msg2[] = "MIKROELEKTRONIKA";
const char msg3[] = "BUTTON1";
const char msg4[] = "BUTTON2";
const char msg5[] = "RC6 OFF";
const char msg6[] = "RC7 OFF";
const char msg7[] = "RC6 ON ";
const char msg8[] = "RC7 ON ";

long x_coord, y_coord, x_coord128, y_coord64;  // scaled x-y position
char msg[16];

char * CopyConst2Ram(char * dest, const char * src){
    for(;*dest++ = *src++;)
    ;
    return dest;
}

// Glcd module connections
char GLCD_DataPort at PORTD;

sbit GLCD_CS1 at RB0_bit;
sbit GLCD_CS2 at RB1_bit;
sbit GLCD_RS  at RB2_bit;
sbit GLCD_RW  at RB3_bit;
sbit GLCD_EN  at RB4_bit;
sbit GLCD_RST at RB5_bit;

sbit GLCD_CS1_Direction at TRISB0_bit;
           S2_Direction at TRISB1_bit;
           S_Direction  at TRISB2_bit;
           W_Direction  at TRISB3_bit;
           N_Direction  at TRISB4_bit;
           ST_Direction at TRISB5_bit;
// End Glcd module connections

unsigned int GetX() {
//reading X
  PORTC.F0 = 1;                                // DRIVEA = 1 (LEFT drive on, RIGHT drive on
                                               // , TOP drive off )
  PORTC.F1 = 0;                                // DRIVEB = 0 (BOTTOM drive off )
  Delay_ms(5);
  return ADC_read(0);                          // reading X value from RA0 (BOTTOM)
}

unsigned int GetY() {
  //reading Y
  PORTC.F0 = 0;                                // DRIVEA = 0 (LEFT drive off , RIGHT drive off
                                               // , TOP drive on)
  PORTC.F1 = 1;                                // DRIVEB = 1 (BOTTOM drive on)
  Delay_ms(5);
  return ADC_read(1);                          // reading Y value from RA1 (from LEFT)
}

void main() {
  PORTA = 0x00;
  TRISA = 0x03;                                // RA0 i RA1 are analog inputs
  ANSEL = 0x03;
  ANSELH = 0;                                  // Configure other AN pins as digital I/O
  PORTC = 0 ;
  TRISC = 0 ;                                  // PORTC is output
  Glcd_Init();                                 // Glcd_Init_EP5
  Glcd_Set_Font(FontSystem5x7_v2, 5, 7, 32);   // Choose font size 5x7
  Glcd_Fill(0);                                // Clear GLCD
  CopyConst2Ram(msg,msg1);                     // Copy "TOUCHPANEL EXAMPLE" string to RAM
  Glcd_Write_Text(msg,10,0,1);
  CopyConst2Ram(msg,msg2);                     // Copy "MIKROELEKTRONIKA" string to RAM
  Glcd_Write_Text(msg,17,7,1);

  //Display Buttons on GLCD:
  Glcd_Rectangle(8,16,60,48,1);
  Glcd_Rectangle(68,16,120,48,1);
  Glcd_Box(10,18,58,46,1);
  Glcd_Box(70,18,118,46,1);
  CopyConst2Ram(msg,msg3);                     // Copy "BUTTON1" string to RAM
  Glcd_Write_Text(msg,14,3,0);
```

```
  CopyConst2Ram(msg,msg5);                     // Copy "RC6 OFF" string to RAM
  Glcd_Write_Text(msg,14,4,0);
  CopyConst2Ram(msg,msg4);                     // Copy "BUTTON2" string to RAM
  Glcd_Write_Text(msg,74,3,0);
  CopyConst2Ram(msg,msg6);                     // Copy "RC7 OFF" string to RAM
  Glcd_Write_Text(msg,74,4,0);

  while (1) {
    // 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) && (x_coord128 <= 58) && (y_coord64 >= 18) && (y_coord64 <= 46)) {
      if(PORTC.F6 == 0) {
        PORTC.F6 = 1;
        CopyConst2Ram(msg,msg7);               // Copy "RC6 ON " string to RAM
        Glcd_Write_Text(msg,14,4,0);
      }
      else {
        PORTC.F6 = 0;
           yConst2Ram(msg,msg5);               // Copy "RC6 OFF" string to RAM
           d_Write_Text(msg,14,4,0);
```



```
    //if BUTTON2 is selected
    if ((x_coord128 >= 70) && (x_coord128 <= 118) && (y_coord64 >= 18) && (y_coord64 <= 46)) {
      if(PORTC.F7 == 0) {
        PORTC.F7 = 1;
        CopyConst2Ram(msg,msg8);               // Copy "RC7 ON " string to RAM
        Glcd_Write_Text(msg,74,4,0);
      }
      else {
        PORTC.F7 = 0;
        CopyConst2Ram(msg,msg6);               // Copy "RC7 OFF" string to RAM
        Glcd_Write_Text(msg,74,4,0);
      }
    }
    Delay_ms(100);
  }
}
```

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

- GLCD
- ADC
- C_Stdlib

← previous chapter | table of contents | next chapter →

Support Forum mikroBUS Lets make Press Legal Archive About Us Customization