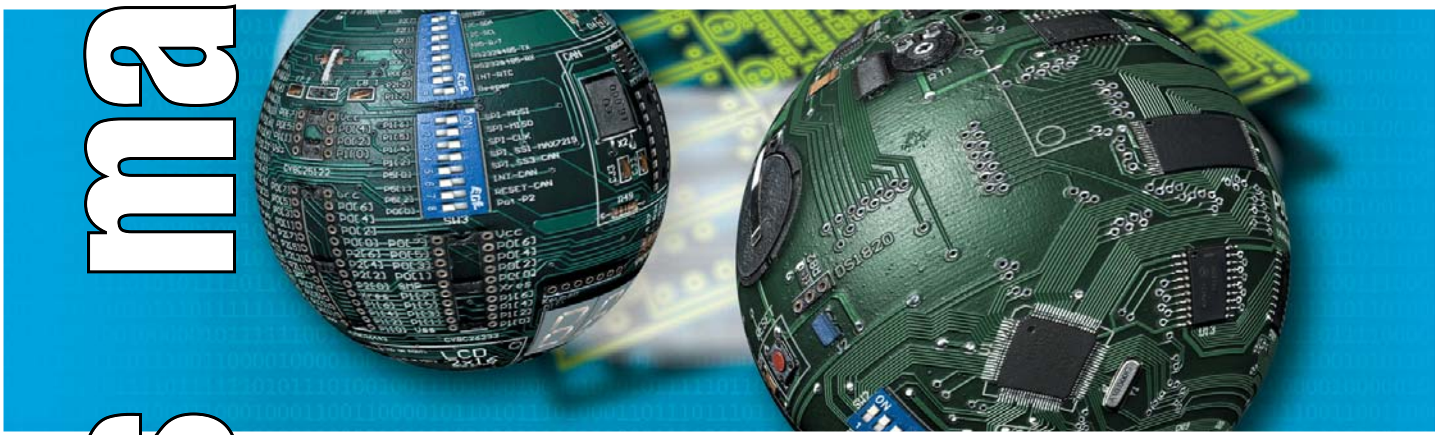
MikroElektronika
Development tools - Books - Compilers
www.mikroelektronika.co.yu

User's manual

BASIC COMPILER FOR MICROCHIP PIC MICROCONTROLLERS

mikroBASIC

making it simple...



Develop your applications quickly and easily with the world's most intuitive Basic compiler for PIC Microcontrollers (families PIC12, PIC16, and PIC18).

Highly sophisticated IDE provides the power you need with the simplicity of a Windows based point-and-click environment.

With useful implemented tools, many practical code examples, broad set of built-in routines, and a comprehensive Help, mikroBasic makes a fast and reliable tool, which can satisfy needs of experienced engineers and beginners alike.

mikroBASIC User's manual

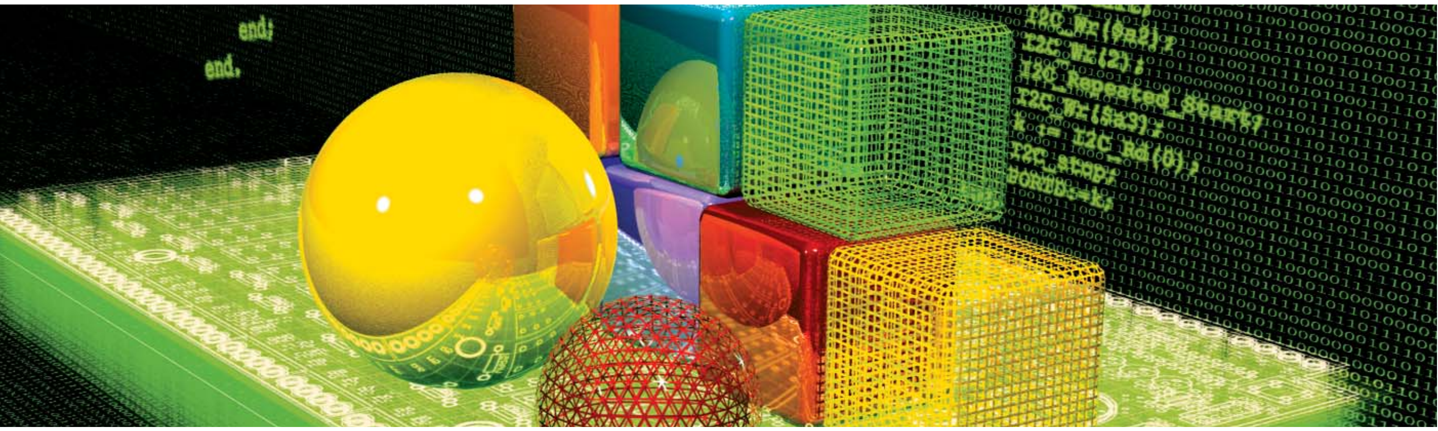


Table of Contents

To readers note

This manual covers mikroBASIC 1.16 and the related topics. New versions may contain changes without prior notice.

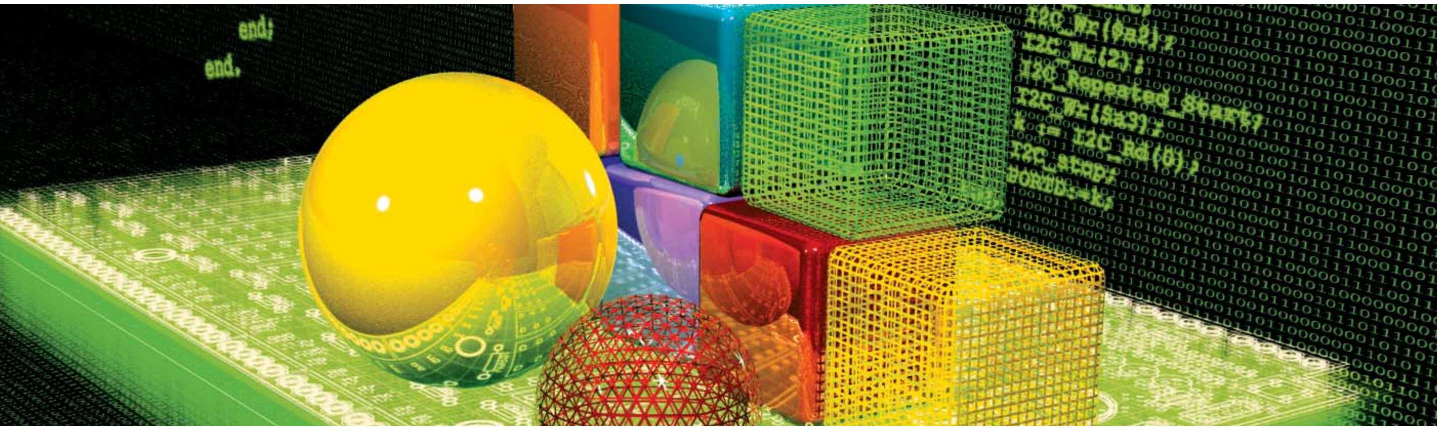
COMPILER BUG REPORTS:
The compiler has been carefully tested and debugged. It is, however, not possible to guarantee a 100 % error free product. If you would like to report a bug, please contact us at the address office@mikroelektronika.co.yu. Please include next information in your bug report:

- Your operating system
- Version of mikroBASIC
- Code sample
- Description of a bug

CONTACT US:
MikroElektronika magazine

Voice:	+381 11 362 04 22, + 381 11 684 919
Fax:	+381 11 362 04 22
Web:	www.MikroElektronika.co.yu
E-mail:	office@MikroElektronika .co.yu

CHAPTER 1

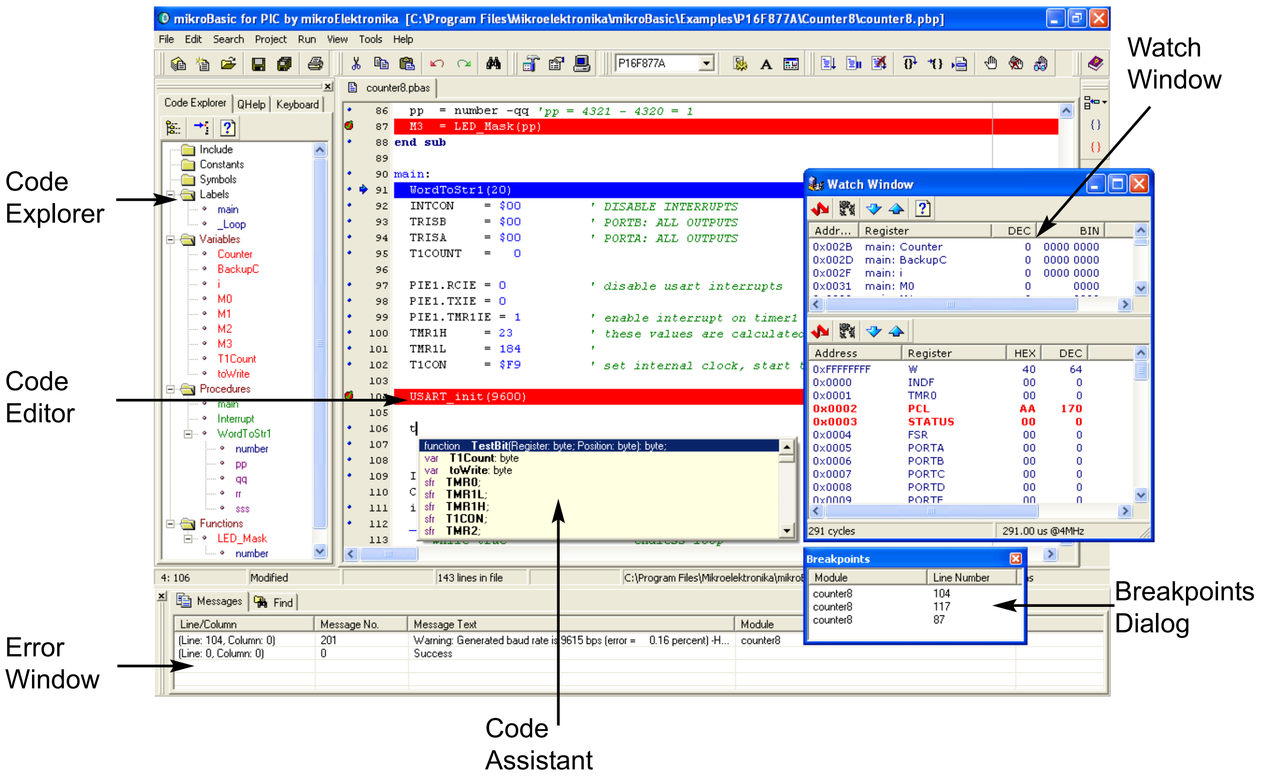


mikroBasic IDE

QUICK OVERVIEW

mikroBasic is a Windows-based Integrated Development Environment, and is much more than just Basic compiler for PIC MCUs. With mikroBasic, you can:

1. Create Basic source code using the built-in Code Editor
2. Compile and link your source code
3. Inspect program flow and debug executable logic with Debugger
4. Monitor variables in Watch Window
5. Get error reports
6. Get detailed statistics (how compiled code utilizes PIC MCU memory, hex map, charts and more...)



```
  pp   = number -qq 'pp = 4321 - 4320 = 1
  M3   = LED_Mask(pp)
end sub

main:
  WordToStr1(20)
  INTCON    = $00          ' DISABLE INTERRUPTS
  TRISB     = $00          ' PORTB: ALL OUTPUTS
  TRISA     = $00          ' PORTA: ALL OUTPUTS
  T1COUNT   =   0

  PIE1.RCIE = 0            ' disable usart interrupts
  PIE1.TXIE = 0
  PIE1.TMR1IE = 1          ' enable interrupt on timer1
  TMR1H     = 23           ' these values are calculated
  TMR1L     = 184          '
  T1CON     = $F9          ' set internal clock, start t

  USART_init(9600)

  t
```

```
function  TestBit(Register: byte; Position: byte): byte;
var   T1Count: byte
var   toWrite: byte
sfr   TMR0;
sfr   TMR1L;
sfr   TMR1H;
sfr   T1CON;
sfr   TMR2;
```

Watch Window

Addr...	Register	DEC	BIN
0x002B	main: Counter	0	0000 0000
0x002D	main: BackupC	0	0000 0000
0x002F	main: i	0	0000 0000
0x0031	main: M0	0	0000

Address	Register	HEX	DEC
0xFFFFFFFF	W	40	64
0x0000	INDF	00	0
0x0001	TMR0	00	0
0x0002	PCL	AA	170
0x0003	STATUS	00	0
0x0004	FSR	00	0
0x0005	PORTA	00	0
0x0006	PORTB	00	0
0x0007	PORTC	00	0
0x0008	PORTD	00	0
0x0009	PORTE	00	0

291 cycles 291.00 us @4MHz

Module	Line Number
counter8	104
counter8	117
counter8	87

Line/Column	Message No.	Message Text	Module
(Line: 104, Column: 0)	201	Warning: Generated baud rate is 9615 bps (error = 0.16 percent) -H...	counter8
(Line: 0, Column: 0)	0	Success	

Code Editor features adjustable Syntax Highlighting, Code Assistant, Parameters Assistant, Auto Correct for common typos, and Code Templates.

Code browser, Keyboard shortcut browser, and Quick Help browser are at your disposal for easier project management.

Error Window displays all errors detected during compiling and linking.

Watch Window enables you to monitor variables, registers and PIC MCU memory.

New Project Wizard is fast, reliable, and easy way to create a project.

Source-level Debugger lets you debug executable logic step-by-step by watching program flow.

Help files are **syntax and context sensitive**.

CODE EDITOR

Basic Editor Features

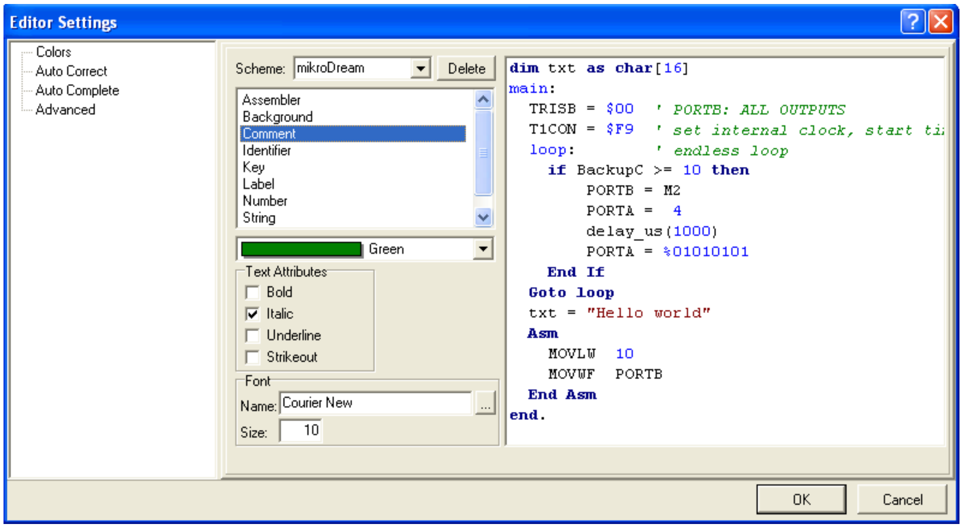
General code editing is same as working with any standard text-editor, including familiar Copy, Paste, and Undo actions, common for Windows environment.

Advanced code editing includes:

- Adjustable Syntax Highlighting
- Code Assistant, Parameters Assistant, Code Templates
- Auto Correct for common typos

You can configure Syntax Highlighting, Code Assistant and Auto Correct from Editor Settings dialog. To access this window, click Tools > Options from drop-down menu, or click Tools icon in Settings toolbar.

Tools Icon.



Scheme: mikroDream

Green

- [ ] Bold
- [x] Italic
- [ ] Underline
- [ ] Strikeout

Name: Courier New

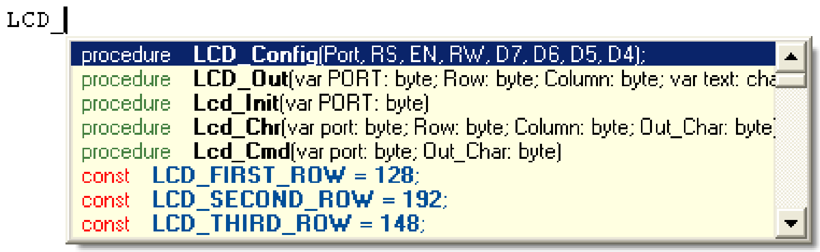
Size: 10

```
dim txt as char[16]
main:
  TRISB = $00  ' PORTB: ALL OUTPUTS
  T1CON = $F9  ' set internal clock, start ti
  loop:        ' endless loop
    if BackupC >= 10 then
        PORTB = M2
        PORTA =  4
        delay_us(1000)
        PORTA = %01010101
    End If
  Goto loop
  txt = "Hello world"
  Asm
    MOVLW  10
    MOVWF  PORTB
  End Asm
end.
```

Advanced Editor Features

Code Assistant [CTRL+SPACE]

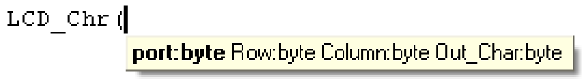
If you type first few letter of a word and then press CTRL+SPACE, all valid identifiers matching the letters you typed will be prompted to you in a floating panel (see the image). Now you can keep typing to narrow the choice, or you can select one from the list using keyboard arrows and Enter.



```
LCD_
procedure   LCD_Config(Port, RS, EN, RW, D7, D6, D5, D4);
procedure   LCD_Out(var PORT: byte; Row: byte; Column: byte; var text: cha
procedure   Lcd_Init(var PORT: byte)
procedure   Lcd_Chr(var port: byte; Row: byte; Column: byte; Out_Char: byte)
procedure   Lcd_Cmd(var port: byte; Out_Char: byte)
const   LCD_FIRST_ROW = 128;
const   LCD_SECOND_ROW = 192;
const   LCD_THIRD_ROW = 148;
```

Parameter Assistant [CTRL+SHIFT+SPACE]

Parameter Assistant will be automatically invoked when you open a parenthesis "(" or press CTRL+SHIFT+SPACE. If name of valid function or procedure precedes the parenthesis, then the expected parameters will be prompted to you in a floating panel. As you type the actual parameter, next expected parameter will become bold.



```
LCD_Chr (
        port:byte Row:byte Column:byte Out_Char:byte
```

Code Template [CTR+J]

You can insert Code Template by typing the name of the template (for instance, *proc*), then press CTRL+J, and Editor will automatically generate code. Or you can click button from Code toolbar and select template from the list.

You can add your own templates to the list. Just select Tools > Options from drop-down menu, or click Tools Icon from Settings Toolbar, and then select Auto Complete Tab. Here you can enter the appropriate keyword, description, and code of your template.

Auto Correct

Auto Correct corrects common typing mistakes. To access the list of recognized typos, select Tools > Options from drop-down menu, or click Tools Icon from Settings Toolbar, and then select Auto Correct Tab. You can also add your own preferences to the list.

Comment /
Uncomment Icon.

Also, Code Editor has feature to comment or uncomment selected block of code by simple click of a mouse, using icons and from Code Toolbar.

Bookmarks

Bookmarks make navigation through large code easier.

CTRL+<number> : Goto bookmark
CTRL+SHIFT+<number> : Set bookmark

Goto Line

Goto Line option makes navigation through large code easier. Select Search > Goto Line from drop-down menu, or use the shortcut CTRL+G.

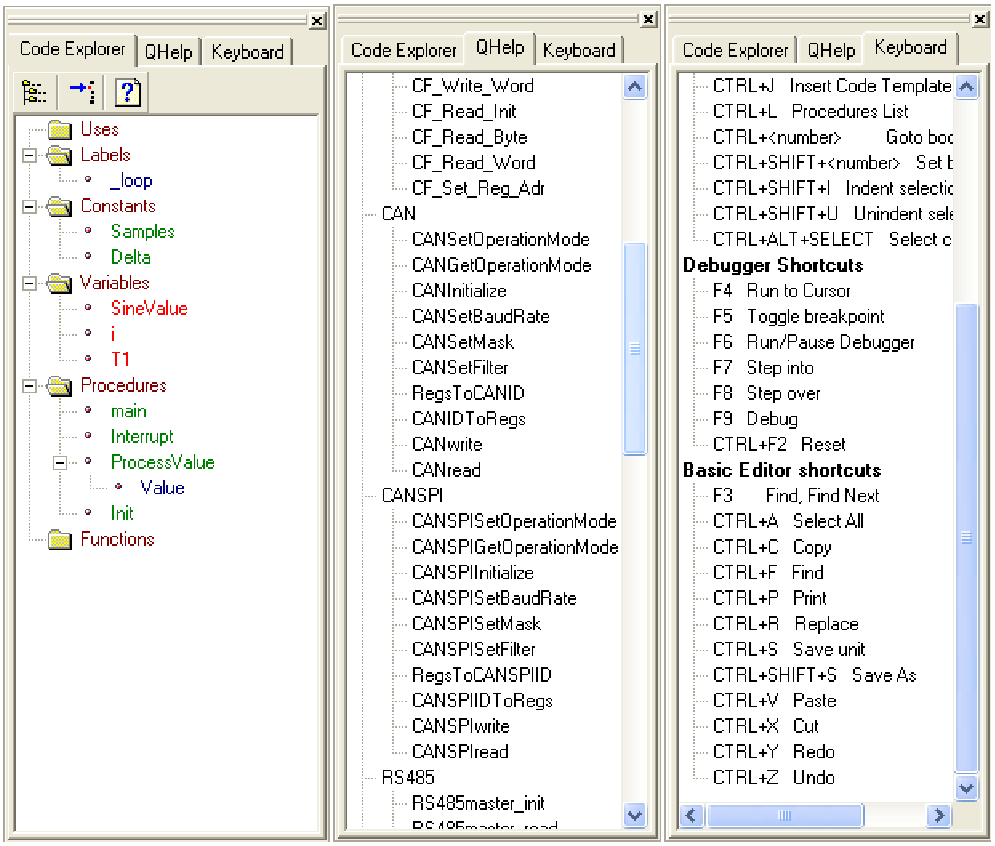
CODE EXPLORER

Find Declaration Icon.

Code Explorer is placed to the left of the main window by default, and gives clear view of every declared item in the source code. You can jump to declaration of any item by right clicking it, or by clicking the Find Declaration icon. To expand or collapse treeview in Code Explorer, use the Collapse/Expand All icon.

Collapse/Expand All Icon.

Also, two more tab windows are available in Code Explorer: Keyboard Tab lists all keyboard shortcuts, and QHelp Tab lists all the available built-in and library functions and procedures, for a quick reference. Double-clicking a routine in QHelp Tab opens an appropriate Help chapter.



```
Code Explorer | QHelp | Keyboard

Uses
Labels
    _loop
Constants
    Samples
    Delta
Variables
    SineValue
    i
    T1
Procedures
    main
    Interrupt
    ProcessValue
        Value
    Init
Functions
```

```
Code Explorer | QHelp | Keyboard

    CF_Write_Word
    CF_Read_Init
    CF_Read_Byte
    CF_Read_Word
    CF_Set_Reg_Adr
CAN
    CANSetOperationMode
    CANGetOperationMode
    CANInitialize
    CANSetBaudRate
    CANSetMask
    CANSetFilter
    RegsToCANID
    CANIDToRegs
    CANwrite
    CANread
CANSPI
    CANSPISetOperationMode
    CANSPIGetOperationMode
    CANSPIInitialize
    CANSPISetBaudRate
    CANSPISetMask
    CANSPISetFilter
    RegsToCANSPIID
    CANSPIIDToRegs
    CANSPIwrite
    CANSPIread
RS485
    RS485master_init
```

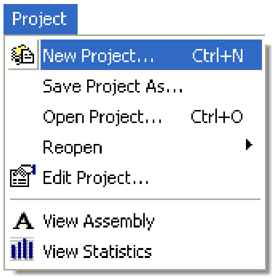
```
Code Explorer | QHelp | Keyboard

    CTRL+J   Insert Code Template
    CTRL+L   Procedures List
    CTRL+<number>       Goto boo
    CTRL+SHIFT+<number>   Set b
    CTRL+SHIFT+I   Indent selectio
    CTRL+SHIFT+U   Unindent sele
    CTRL+ALT+SELECT   Select c
Debugger Shortcuts
    F4   Run to Cursor
    F5   Toggle breakpoint
    F6   Run/Pause Debugger
    F7   Step into
    F8   Step over
    F9   Debug
    CTRL+F2   Reset
Basic Editor shortcuts
    F3      Find, Find Next
    CTRL+A   Select All
    CTRL+C   Copy
    CTRL+F   Find
    CTRL+P   Print
    CTRL+R   Replace
    CTRL+S   Save unit
    CTRL+SHIFT+S   Save As
    CTRL+V   Paste
    CTRL+X   Cut
    CTRL+Y   Redo
    CTRL+Z   Undo
```

CREATING FIRST PROJECT

New Project Icon.

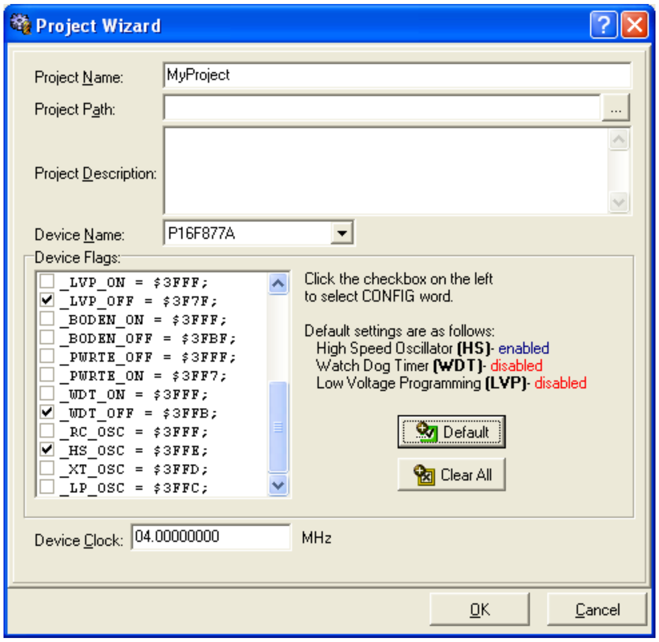
Step 1

From a drop-down menu, select: Project > New Project, or click New Project icon



Step 2

Fill the New Project Wizard dialog with correct values to set up your new project.
- Select a device for your project from the drop-down menu
- Set configuration bits (Device Flags) by clicking Default push-button.
- Select Device Clock by entering appropriate value in edit box.
- Enter a name for your new project
- Enter project description edit box for closer information about your project
- Enter project path



Project Name: MyProject
Project Path:
Project Description:
Device Name: P16F877A
Device Flags:

```
[ ] _LVP_ON = $3FFF;
[✓] _LVP_OFF = $3F7F;
[ ] _BODEN_ON = $3FFF;
[ ] _BODEN_OFF = $3FBF;
[ ] _PWRTE_OFF = $3FFF;
[ ] _PWRTE_ON = $3FF7;
[ ] _WDT_ON = $3FFF;
[✓] _WDT_OFF = $3FFB;
[ ] _RC_OSC = $3FFF;
[✓] _HS_OSC = $3FFE;
[ ] _XT_OSC = $3FFD;
[ ] _LP_OSC = $3FFC;
```

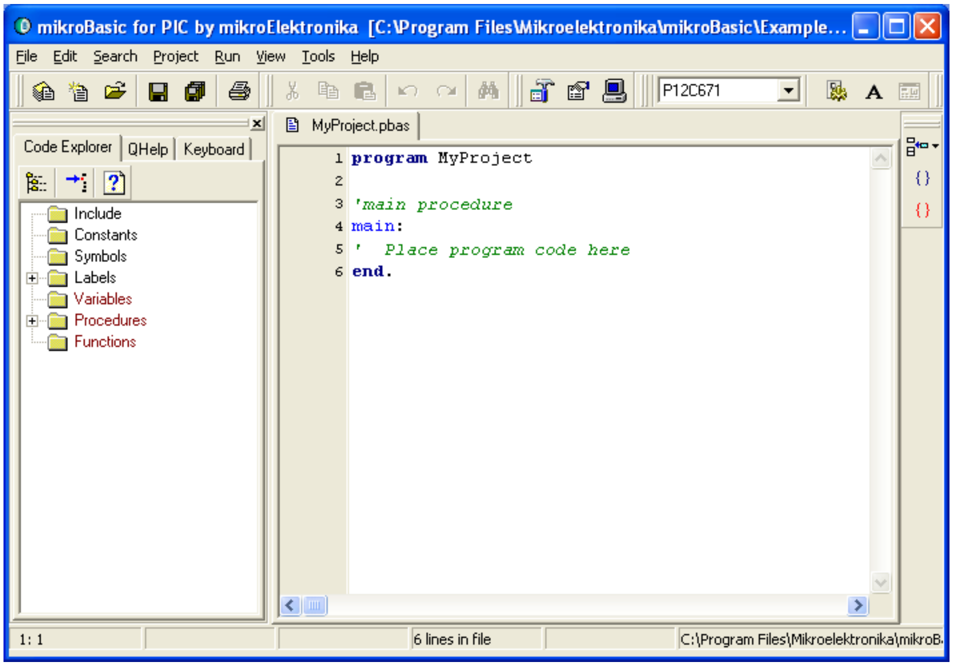
Click the checkbox on the left to select CONFIG word.

Default settings are as follows:
High Speed Oscillator **(HS)**- enabled
Watch Dog Timer **(WDT)**- disabled
Low Voltage Programming **(LVP)**- disabled

Device Clock: 04.00000000 MHz

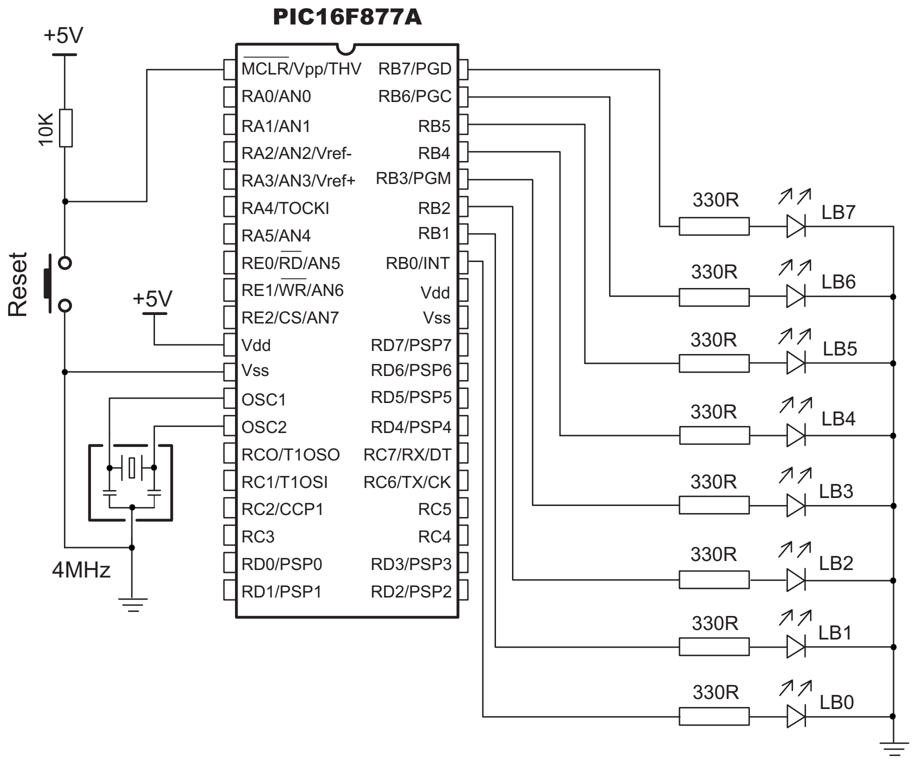
After you have set up your project, select OK push button in New Project Wizard dialog box. mikroBasic will create project for you and automatically open the program file in code editor. Now we can write the source code.

Step 3
After you have successfully created an empty project with New Project Wizard, Code Editor will display an empty program file, named same as your project.


mikroBasic for PIC by mikroElektronika [C:\Program Files\Mikroelektronika\mikroBasic\Example...
File Edit Search Project Run View Tools Help
P12C671
Code Explorer QHelp Keyboard
Include
Constants
Symbols
Labels
Variables
Procedures
Functions
MyProject.pbas
1: 1
6 lines in file
C:\Program Files\Mikroelektronika\mikroB

```
program MyProject

'main procedure
main:
'  Place program code here
end.
```

Now we can write the code for this simple example. We want to make LED diode blink once per second. Assuming we have the configuration given in the following figure, LED diodes are connected to PIC16F877 PORTB pins. (it can be any other PIC that has PORTB)



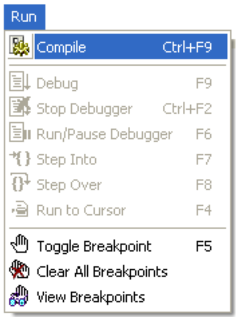
In this configuration, LED will emit light when voltage on pin is high (5V), and will be off when voltage on pin is low (0V). We have to designate PORTB pins as output, and change its value every second. Listing of program is below

```
program My_LED

main:

  TRISB = 0                    ' configure  pins of PORTB as output
  eloop:
      PORTB = $FF              ' turn on diodes on PORTB
      delay_ms(1000)           ' wait 1 second
      PORTB = 0                ' turn of diodes on PORTB
      delay_ms(1000)           ' wait 1 second
  goto eloop                   ' stay in a loop
end.
```

Compile Icon.

Step 4

Before compiling, it is recommended to save the project (menu choice File>Save All). Now you can compile your code by selecting menu Run > Compile, or by clicking the Compile icon.



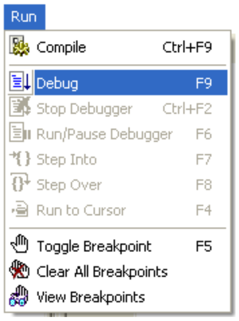
mikroBasic has generated hex file which can be used to program PIC MCU. But before that, let's check our program with the Debugger. Also mikroBasic generates list and assembly files.

Debug Icon.

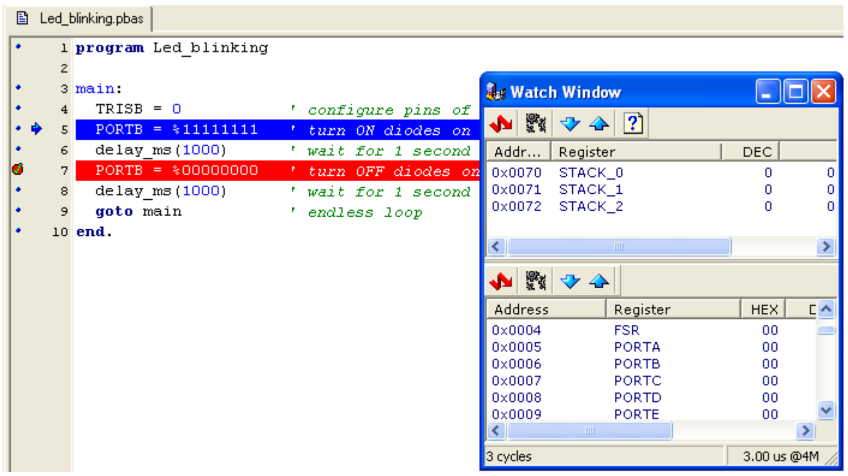
Step 5

After successful compiling, we can use mikroBasic Debugger to check our program behavior before we feed it to the device (PIC16F877 or other). For a simple program such as this, simulation is not really necessary, but it is a requirement for more complex programs.

To start the Debugger, select Run > Debug, or click the Debug icon, or simply hit F9.



Upon starting the Debugger, Watch Window appears, and the active line in Code Editor marks the instruction to be executed next. We will set the breakpoint at line 7 by positioning the cursor to that line and toggling the breakpoint (Run > Toggle Breakpoint or F5). See the following image.


Led_blinking.pbas

```
program Led_blinking

main:
  TRISB = 0               ' configure pins of
  PORTB = %11111111       ' turn ON diodes on
  delay_ms(1000)          ' wait for 1 second
  PORTB = %00000000       ' turn OFF diodes on
  delay_ms(1000)          ' wait for 1 second
  goto main               ' endless loop
end.
```

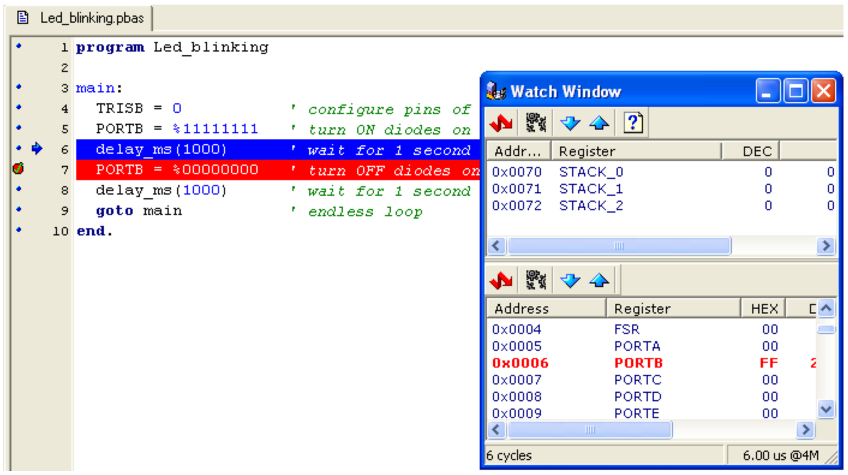
Watch Window

Addr...	Register	DEC	
0x0070	STACK_0	0	0
0x0071	STACK_1	0	0
0x0072	STACK_2	0	0

Address	Register	HEX
0x0004	FSR	00
0x0005	PORTA	00
0x0006	PORTB	00
0x0007	PORTC	00
0x0008	PORTD	00
0x0009	PORTE	00

3 cycles 3.00 us @4M

We will use the Step Over option (Run > Step Over or F8) to execute the current program line. Now, you can see the changes in variables, SFR registers, etc, in the Watch Window – items that have changed are marked red, as shown in the image below.


Led_blinking.pbas

```
program Led_blinking

main:
  TRISB = 0               ' configure pins of
  PORTB = %11111111       ' turn ON diodes on
  delay_ms(1000)          ' wait for 1 second
  PORTB = %00000000       ' turn OFF diodes on
  delay_ms(1000)          ' wait for 1 second
  goto main               ' endless loop
end.
```

Watch Window

Addr...	Register	DEC	
0x0070	STACK_0	0	0
0x0071	STACK_1	0	0
0x0072	STACK_2	0	0

Address	Register	HEX	D
0x0004	FSR	00	
0x0005	PORTA	00	
0x0006	PORTB	FF	2
0x0007	PORTC	00	
0x0008	PORTD	00	
0x0009	PORTE	00	

6 cycles 6.00 us @4M

We could have used Run/Pause (F6) option to execute all the instructions between the active line and the breakpoint (Run > Run/Pause Debugger).

Step 6

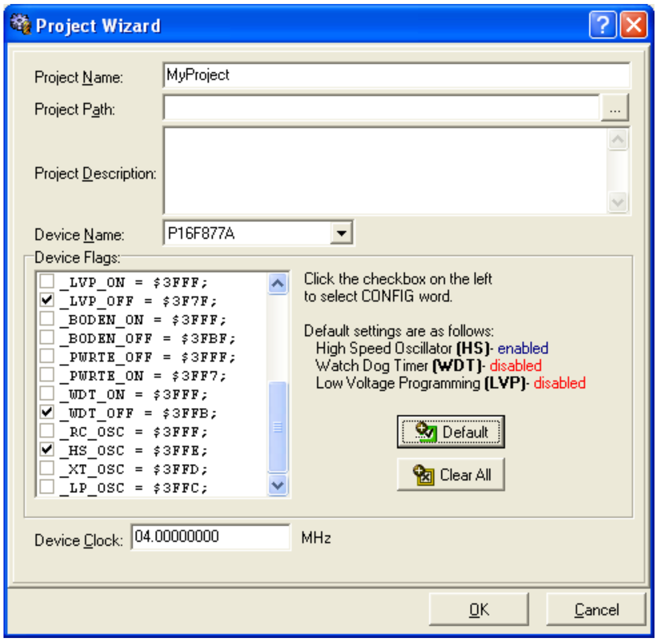
Now we can use hex file and feed it to the device (PIC16F877 or other). In order to do so hex file must be loaded in programmer (PIC Flash by mikroElektronika or any other).

PROJECTS

Each application, or project, consists of a single project file and one or more unit files. You can compile source files only if they are part of the project. First and essential step is creating a project.

We will use New Project Wizard to create our new project.

Select Project > New Project from drop-down menu and follow the dialog:



Project Name: MyProject
Project Path:
Project Description:
Device Name: P16F877A
Device Flags:

```
[ ] _LVP_ON = $3FFF;
[✓] _LVP_OFF = $3F7F;
[ ] _BODEN_ON = $3FFF;
[ ] _BODEN_OFF = $3FBF;
[ ] _PWRTE_OFF = $3FFF;
[ ] _PWRTE_ON = $3FF7;
[ ] _WDT_ON = $3FFF;
[✓] _WDT_OFF = $3FFB;
[ ] _RC_OSC = $3FFF;
[✓] _HS_OSC = $3FFE;
[ ] _XT_OSC = $3FFD;
[ ] _LP_OSC = $3FFC;
```

Click the checkbox on the left to select CONFIG word.

Default settings are as follows:
High Speed Oscillator **(HS)**- enabled
Watch Dog Timer **(WDT)**- disabled
Low Voltage Programming **(LVP)**- disabled

Device Clock: 04.00000000 MHz

(select PIC MCU device, device clock, setup configuration bits, set project name, location and description)

Later, if you want to change some project settings, select Project > Edit from drop-down menu. To save your project , select Project > Save All from drop-down menu. To save your project under different name, select Project > Save Project As from drop-down menu. To open a project, select Project > Open, or Project > Reopen from drop-down menu.

When you create new project, mikroBasic automatically creates an empty main unit file in which you'll write your source code.

Managing Source Files

Source files created in mikroBasic have the extension pbas. By default, main module file is named same as the project.

Location of the main unit source file and other project information are stored in project file with extension pbp.

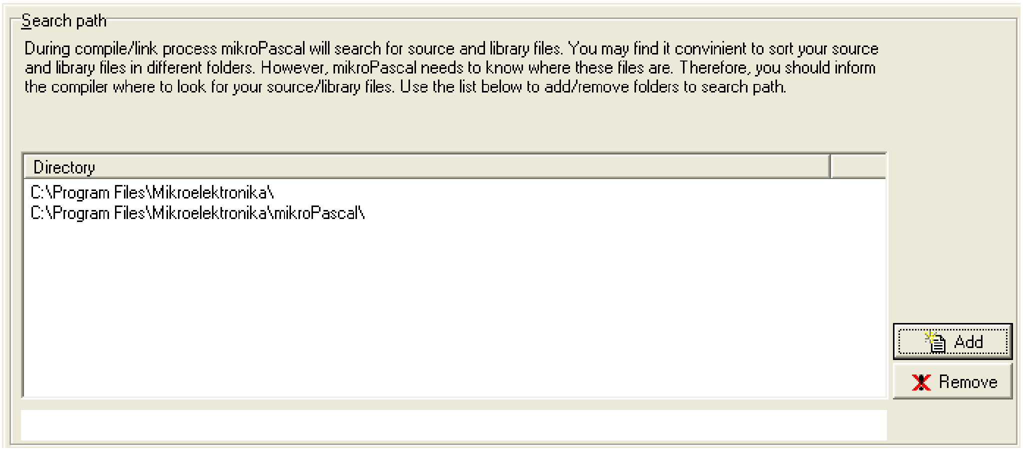
Creating Main Module File

Main module file is created simultaneously with the project and is named same as the project, with extension pbas. You should not change the name of this file as mikroBasic might not be able to compile it. Project file and main module file must be saved in the same folder.

Creating a New Unit File

Select File > New module from drop-down menu, or press CTRL+N, or click the New File icon. A new tab will open, named "Untitled1". This is your new module file. Select File > Save As from drop-down menu to name it the way you want.

Keyword `include` instructs compiler which unit beside main module should be compiled. Module other than main must be in same folder with project file or in folder specified by search path. Search path can be configured by selecting menu choice Options > Settings from drop-down menu and then tab window Advanced.


Search path

During compile/link process mikroPascal will search for source and library files. You may find it convinient to sort your source and library files in different folders. However, mikroPascal needs to know where these files are. Therefore, you should inform the compiler where to look for your source/library files. Use the list below to add/remove folders to search path.

Directory
C:\Program Files\Mikroelektronika\
C:\Program Files\Mikroelektronika\mikroPascal\

Opening an Existing File

Open File Icon.

Select File > Open from drop-down menu, or press CTRL+O, or click the Open File icon. The Select Input File dialog opens. In the dialog, browse to the location of the file you want to open and select it. Click the Open button.
The selected file is displayed in its own tab. If the selected file is already open, its current Editor tab will become active.

Printing an Open File

Print File Icon.

Make sure that window containing the file you want to print is the active window. Select File > Print from drop-down menu, or press CTRL+P, or click the Print icon. In the Print Preview Window, set the desired layout of the document and click the OK button. The file will be printed on the selected printer.

Saving File

Save File Icon.

Make sure that window containing the file you want to save is the active window. Select File > Save from drop-down menu, or press CTRL+S, or click the Save icon. The file will be saved under the name on its window.

Saving File Under a Different Name

Make sure that window containing the file you want to save is the active window. Select File > Save As from drop-down menu, or press SHIFT+CTRL+S. The New File Name dialog will be displayed.
In the dialog, browse to the folder where you want to save the file.
In the File Name field, modify the name of the file you want to save.

Closing a File

Click the Save button. Make sure that tab containing the file you want to close is the active tab. Select File > Close from drop-down menu, or right click the tab of the file you want to close in Code Editor. If the file has been changed since it was last saved, you will be prompted to save your changes.

Compile Source Code

Compile Icon.

When you have created the project and written the source code, you will want to compile it. Select Run > Compile from drop-down menu, or click Compiler Icon from Compiler Toolbar.

Progress bar will appear to inform you about the status of compiling. If no errors are encountered, mikroBasic will produce hex file, assembly file, and list for the appropriate PIC MCU.

DEBUGGER

Source-level Debugger is integral component of mikroBasic development environment. It is designed to simulate operations of Microchip Technology's PIC MCU's and to assist users in debugging Basic software written for these devices.

Debug Icon.

Debugger simulates program flow and execution of instruction lines, but does not fully emulate PIC device behavior: it does not update timers, interrupt flags, etc. Jump to interrupt is performed by clicking the Interrupt icon .

Debug Icon.

After you have successfully compiled your project, you can run Debugger by selecting Run > Debug from drop-down menu, or by clicking Debug Icon . Starting the Debugger makes more options available: Step Into, Step Over, Run to Cursor etc. Line that is to be executed is color highlighted (blue).

Debug [F9]
Starts Debugger.

Step Into [F7]
Execute the current Basic instruction (single or multiple cycle instructions) and then halt. After execution, all windows are updated. If the instruction is a procedure or function call, execute it enters routine and halt at the first following instruction after the call.

Step Over [F8]
Execute the current Basic instruction (single or multiple cycle instructions) then halt. If the instruction is a procedure or function call, execute the called routine and halt at the instruction following the call.

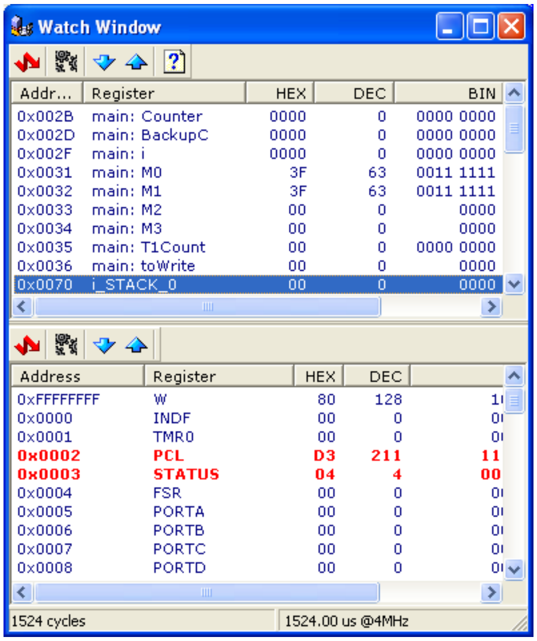
Run to cursor [F4]
Executes all instructions between the current instruction and the cursor position.

Toggle Breakpoints [F5]
Toggle breakpoint at current cursor position.

Run/Pause Debugger [F6]
Run or pause Debugger.

Run > View Breakpoints
Invoke breakpoints window, with list of breakpoints. Double clicking item in window list locates breakpoint.

Watch Window
Watch Window allows you to monitor program items while running your program. It displays variables and special function registers of PIC MCU, their addresses and values. Values are updated as you go through the simulation. See the image below.

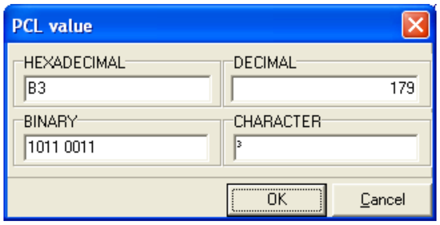

Watch Window

Addr...	Register	HEX	DEC	BIN
0x002B	main: Counter	0000	0	0000 0000
0x002D	main: BackupC	0000	0	0000 0000
0x002F	main: i	0000	0	0000 0000
0x0031	main: M0	3F	63	0011 1111
0x0032	main: M1	3F	63	0011 1111
0x0033	main: M2	00	0	0000
0x0034	main: M3	00	0	0000
0x0035	main: T1Count	00	0	0000 0000
0x0036	main: toWrite	00	0	0000
0x0070	i_STACK_0	00	0	0000

Address	Register	HEX	DEC	
0xFFFFFFFF	W	80	128	1
0x0000	INDF	00	0	0
0x0001	TMR0	00	0	0
0x0002	**PCL**	**D3**	**211**	**11**
0x0003	**STATUS**	**04**	**4**	**00**
0x0004	FSR	00	0	0
0x0005	PORTA	00	0	0
0x0006	PORTB	00	0	0
0x0007	PORTC	00	0	0
0x0008	PORTD	00	0	0

1524 cycles | 1524.00 us @4MHz

Double clicking one of the items opens a window in which you can assign new value to the selected variable or register.

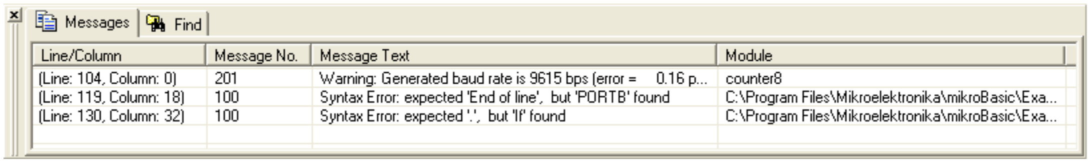

PCL value

HEXADECIMAL: B3
DECIMAL: 179
BINARY: 1011 0011
CHARACTER: ³

ERROR WINDOW

In case that errors were encountered during compiling, compiler will report them and won't generate a hex file. Error Window will be prompted at the bottom of the main window.

Error Window is located under message tab, and displays location and type of errors compiler has encountered. Compiler also reports warnings, but these do not affect generating hex code. Only errors can interefere with generation of hex.



Line/Column	Message No.	Message Text	Module
(Line: 104, Column: 0)	201	Warning: Generated baud rate is 9615 bps (error = 0.16 p...	counter8
(Line: 119, Column: 18)	100	Syntax Error: expected 'End of line', but 'PORTB' found	C:\Program Files\Mikroelektronika\mikroBasic\Exa...
(Line: 130, Column: 32)	100	Syntax Error: expected '.', but 'If' found	C:\Program Files\Mikroelektronika\mikroBasic\Exa...

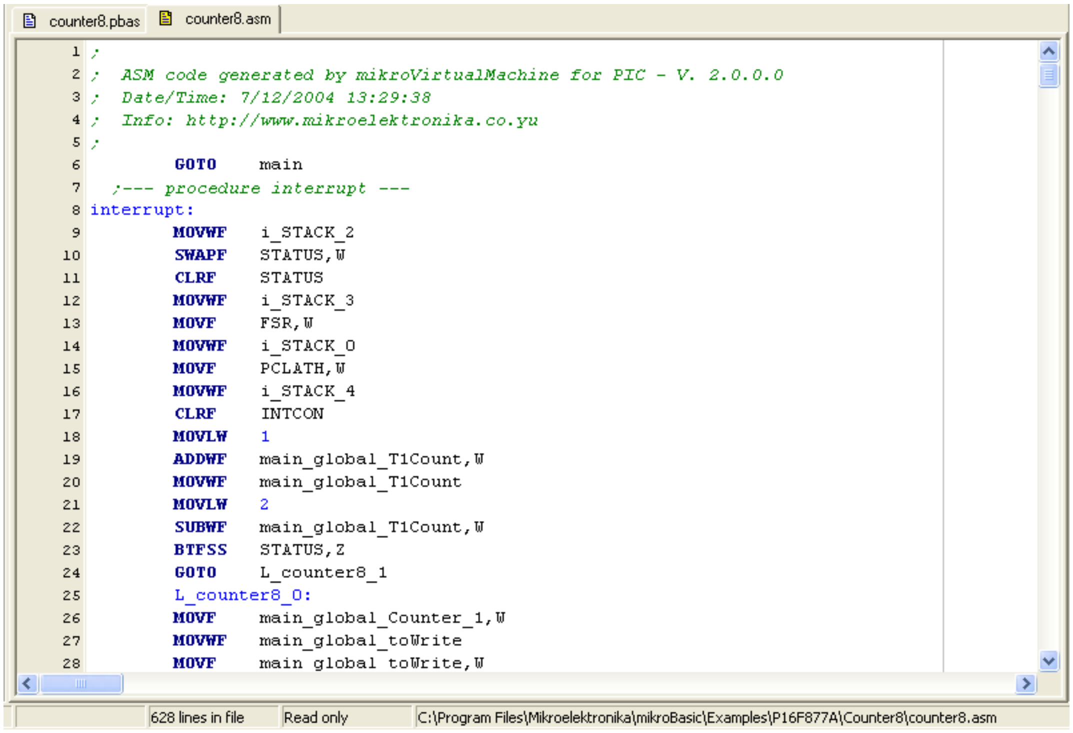
Double clicking the message line in Error Window results in highlighting the line of source code where the error took place.

ASSEMBLY VIEW

Assembly Icon.

After compiling your program in mikroBasic, you can click toolbar Assembly icon or select Project > View Assembly from drop-down menu to review generated assembly code in a new tab window. Assembly is human readable with symbolic names. All physical addresses and other information can be found in Statistics or in list file.

If program is not compiled and there is no assembly file, starting this option will compile your code and then display assembly.



```
;
;  ASM code generated by mikroVirtualMachine for PIC - V. 2.0.0.0
;  Date/Time: 7/12/2004 13:29:38
;  Info: http://www.mikroelektronika.co.yu
;
        GOTO    main
  ;--- procedure interrupt ---
interrupt:
        MOVWF   i_STACK_2
        SWAPF   STATUS,W
        CLRF    STATUS
        MOVWF   i_STACK_3
        MOVF    FSR,W
        MOVWF   i_STACK_0
        MOVF    PCLATH,W
        MOVWF   i_STACK_4
        CLRF    INTCON
        MOVLW   1
        ADDWF   main_global_T1Count,W
        MOVWF   main_global_T1Count
        MOVLW   2
        SUBWF   main_global_T1Count,W
        BTFSS   STATUS,Z
        GOTO    L_counter8_1
        L_counter8_0:
        MOVF    main_global_Counter_1,W
        MOVWF   main_global_toWrite
        MOVF    main_global_toWrite,W
```

628 lines in file | Read only | C:\Program Files\Mikroelektronika\mikroBasic\Examples\P16F877A\Counter8\counter8.asm

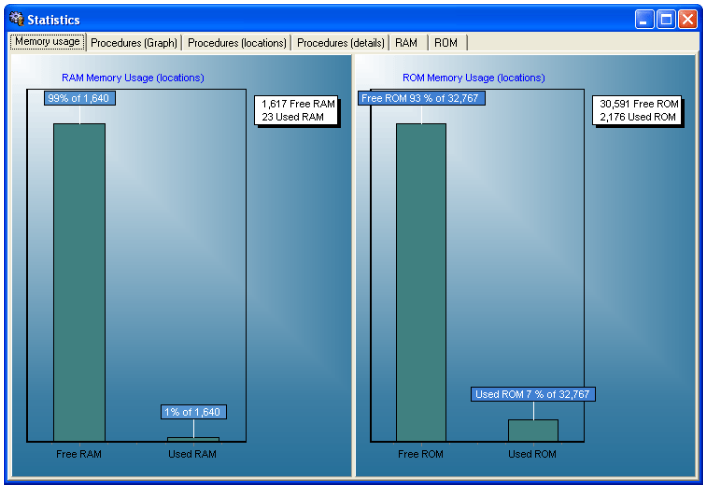
STATISTICS

Statistics Icon.

After successful compiling, you can review statistics on your code. Select Project > View Statistics from drop-down menu, or click the Statistics icon. There are five tab windows:

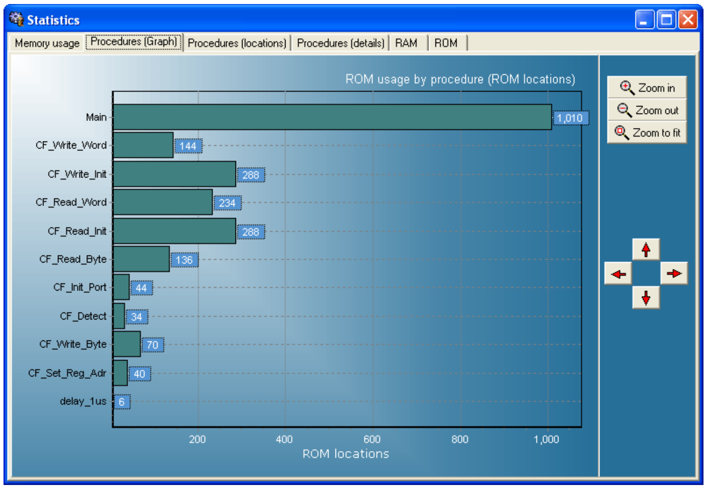
Memory Usage Window

Provides overview of RAM and ROM memory usage in form of histogram.



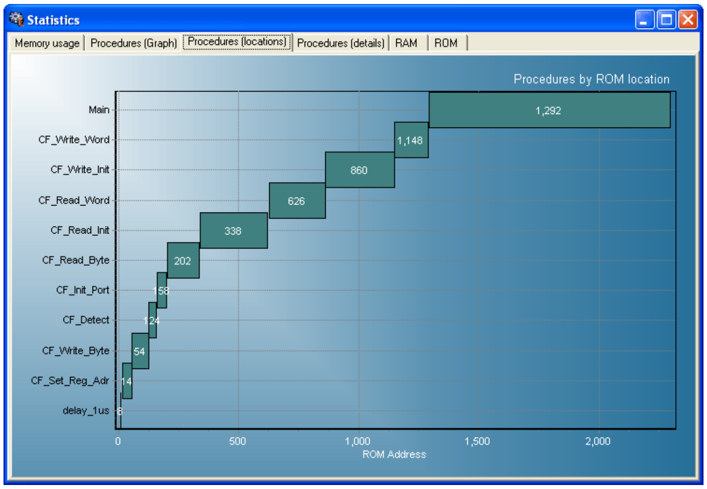
Procedures (Graph) Window

Displays procedures and functions in form of histogram, according to their memory allotment.



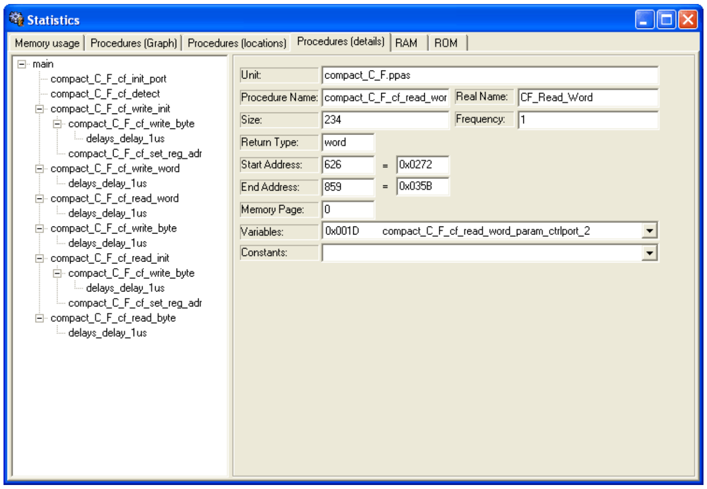
Procedures (Locations) Window

Displays how procedures and functions are located in microcontroller's memory.



Procedures (Details) Window

Displays complete call tree, along with details for each procedure and function: size, start and end address, frequency in program, return type, etc.

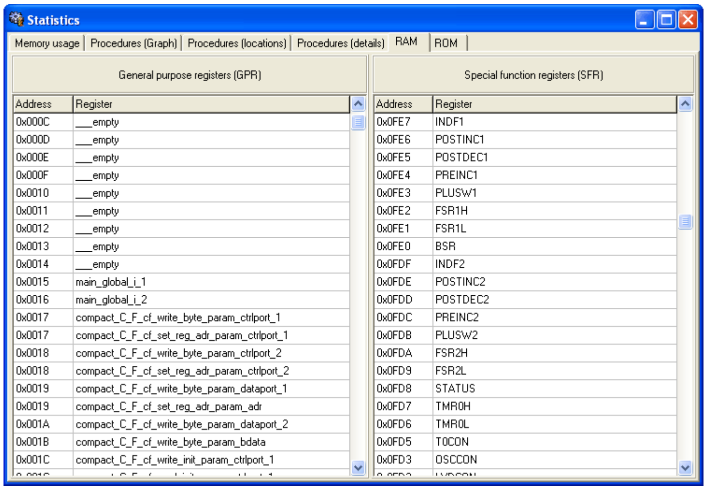


```
main
  compact_C_F_cf_init_port
  compact_C_F_cf_detect
  compact_C_F_cf_write_init
    compact_C_F_cf_write_byte
      delays_delay_1us
    compact_C_F_cf_set_reg_adr
  compact_C_F_cf_write_word
    delays_delay_1us
  compact_C_F_cf_read_word
    delays_delay_1us
  compact_C_F_cf_write_byte
    delays_delay_1us
  compact_C_F_cf_read_init
    compact_C_F_cf_write_byte
      delays_delay_1us
    compact_C_F_cf_set_reg_adr
  compact_C_F_cf_read_byte
    delays_delay_1us
```

Unit:	compact_C_F.ppas		
Procedure Name:	compact_C_F_cf_read_wor	Real Name:	CF_Read_Word
Size:	234	Frequency:	1
Return Type:	word		
Start Address:	626	=	0x0272
End Address:	859	=	0x035B
Memory Page:	0		
Variables:	0x001D compact_C_F_cf_read_word_param_ctrlport_2		
Constants:			

RAM Window

Summarizes all GPR and SFR registers and their addresses. Also displays symbolic names of variables and their addresses.



General purpose registers (GPR)

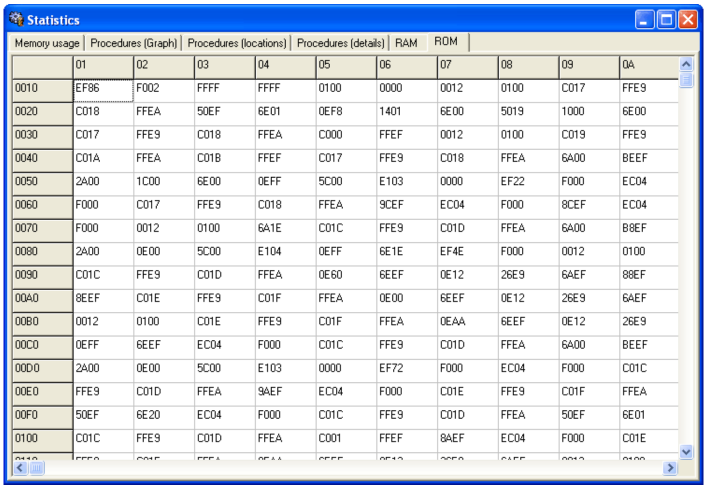
Address	Register
0x000C	___empty
0x000D	___empty
0x000E	___empty
0x000F	___empty
0x0010	___empty
0x0011	___empty
0x0012	___empty
0x0013	___empty
0x0014	___empty
0x0015	main_global_i_1
0x0016	main_global_i_2
0x0017	compact_C_F_cf_write_byte_param_ctrlport_1
0x0017	compact_C_F_cf_set_reg_adr_param_ctrlport_1
0x0018	compact_C_F_cf_write_byte_param_ctrlport_2
0x0018	compact_C_F_cf_set_reg_adr_param_ctrlport_2
0x0019	compact_C_F_cf_write_byte_param_dataport_1
0x0019	compact_C_F_cf_set_reg_adr_param_adr
0x001A	compact_C_F_cf_write_byte_param_dataport_2
0x001B	compact_C_F_cf_write_byte_param_bdata
0x001C	compact_C_F_cf_write_init_param_ctrlport_1

Special function registers (SFR)

Address	Register
0x0FE7	INDF1
0x0FE6	POSTINC1
0x0FE5	POSTDEC1
0x0FE4	PREINC1
0x0FE3	PLUSW1
0x0FE2	FSR1H
0x0FE1	FSR1L
0x0FE0	BSR
0x0FDF	INDF2
0x0FDE	POSTINC2
0x0FDD	POSTDEC2
0x0FDC	PREINC2
0x0FDB	PLUSW2
0x0FDA	FSR2H
0x0FD9	FSR2L
0x0FD8	STATUS
0x0FD7	TMR0H
0x0FD6	TMR0L
0x0FD5	T0CON
0x0FD3	OSCCON

ROM Window

Lists op-codes and their addresses in form of a human readable hex code.

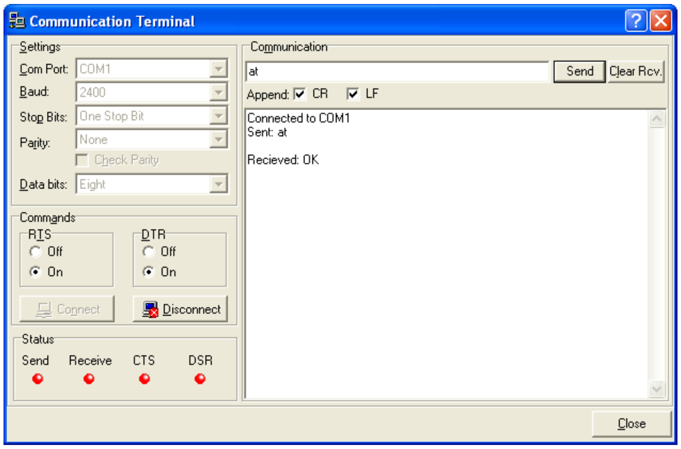


	01	02	03	04	05	06	07	08	09	0A
0010	EF86	F002	FFFF	FFFF	0100	0000	0012	0100	C017	FFE9
0020	C018	FFEA	50EF	6E01	0EF8	1401	6E00	5019	1000	6E00
0030	C017	FFE9	C018	FFEA	C000	FFEF	0012	0100	C019	FFE9
0040	C01A	FFEA	C01B	FFEF	C017	FFE9	C018	FFEA	6A00	BEEF
0050	2A00	1C00	6E00	0EFF	5C00	E103	0000	EF22	F000	EC04
0060	F000	C017	FFE9	C018	FFEA	9CEF	EC04	F000	8CEF	EC04
0070	F000	0012	0100	6A1E	C01C	FFE9	C01D	FFEA	6A00	B8EF
0080	2A00	0E00	5C00	E104	0EFF	6E1E	EF4E	F000	0012	0100
0090	C01C	FFE9	C01D	FFEA	0E60	6EEF	0E12	26E9	6AEF	88EF
00A0	8EEF	C01E	FFE9	C01F	FFEA	0E00	6EEF	0E12	26E9	6AEF
00B0	0012	0100	C01E	FFE9	C01F	FFEA	0EAA	6EEF	0E12	26E9
00C0	0EFF	6EEF	EC04	F000	C01C	FFE9	C01D	FFEA	6A00	BEEF
00D0	2A00	0E00	5C00	E103	0000	EF72	F000	EC04	F000	C01C
00E0	FFE9	C01D	FFEA	9AEF	EC04	F000	C01E	FFE9	C01F	FFEA
00F0	50EF	6E20	EC04	F000	C01C	FFE9	C01D	FFEA	50EF	6E01
0100	C01C	FFE9	C01D	FFEA	C001	FFEF	8AEF	EC04	F000	C01E

INTEGRATED TOOLS

USART Terminal

mikroBasic includes USART (Universal Synchronous Asynchronous Receiver Transmitter) communication terminal for RS232 communication. You can launch it from drop-down menu Tools > Terminal or by clicking the icon .


Communication Terminal

Settings
Com Port: COM1
Baud: 2400
Stop Bits: One Stop Bit
Parity: None
Check Parity
Data bits: Eight

Commands
RTS: Off [✓ On]
DTR: Off [✓ On]

Connect | Disconnect

Status
Send ● Receive ● CTS ● DSR ●

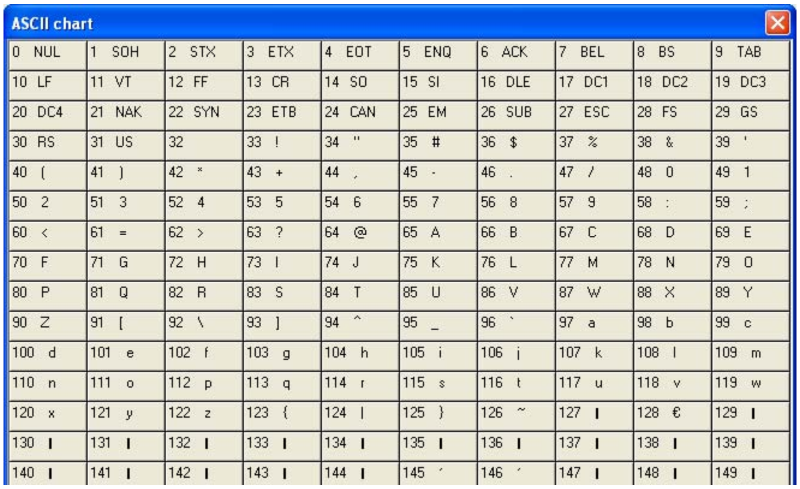
Communication
at
Send | Clear Rcv.
Append: [✓ CR] [✓ LF]

```
Connected to COM1
Sent: at

Recieved: OK
```

Close

ASCII Chart

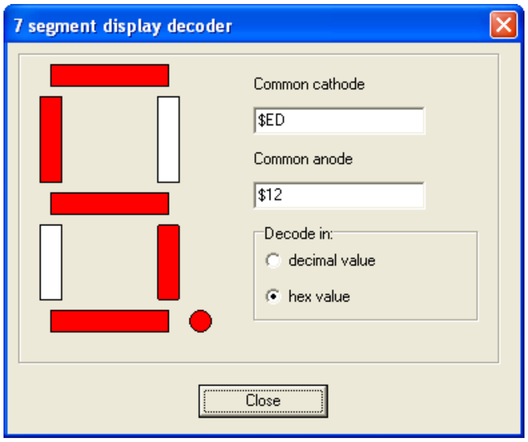
ASCII Chart is a handy tool, particularly useful when working with LCD display. You can launch it from drop-down menu Tools > ASCII chart.


ASCII chart

0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7 BEL	8 BS	9 TAB
10 LF	11 VT	12 FF	13 CR	14 SO	15 SI	16 DLE	17 DC1	18 DC2	19 DC3
20 DC4	21 NAK	22 SYN	23 ETB	24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS
30 RS	31 US	32	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /	48 0	49 1
50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;
60 <	61 =	62 >	63 ?	64 @	65 A	66 B	67 C	68 D	69 E
70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y
90 Z	91 [	92 \	93]	94 ^	95 _	96 `	97 a	98 b	99 c
100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m
110 n	111 o	112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 ▮	128 €	129 ▮
130 ▮	131 ▮	132 ▮	133 ▮	134 ▮	135 ▮	136 ▮	137 ▮	138 ▮	139 ▮
140 ▮	141 ▮	142 ▮	143 ▮	144 ▮	145 ‘	146 ’	147 ▮	148 ▮	149 ▮

7 Segment Display Decoder

7seg Display Decoder is a convenient visual panel which returns decimal/hex value for any viable combination you would like to display on 7seg. Click on the parts of 7 segment image to the left to get the desired value in the edit boxes. You can launch it from drop-down menu Tools > 7 Segment Display.



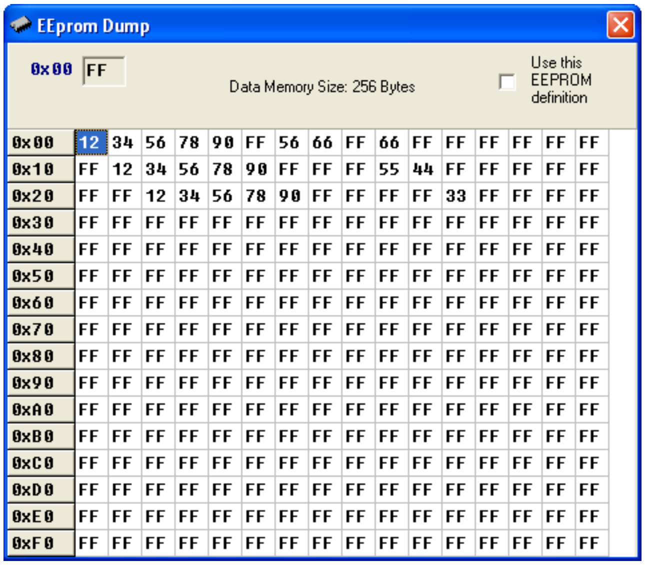
Common cathode: $ED

Common anode: $12

Decode in: decimal value [✓ hex value]

EEPROM Editor

EEPROM Editor allows you to easily manage EEPROM of PIC microcontroller.



```
0x00 FF          Data Memory Size: 256 Bytes     [ ] Use this EEPROM definition

0x00  12 34 56 78 90 FF 56 66 FF 66 FF FF FF FF FF FF
0x10  FF 12 34 56 78 90 FF FF FF 55 44 FF FF FF FF FF
0x20  FF FF 12 34 56 78 90 FF FF FF FF 33 FF FF FF FF
0x30  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x40  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x50  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x60  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x70  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x80  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x90  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0xA0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0xB0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0xC0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0xD0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0xE0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0xF0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

KEYBOARD SHORTCUTS

Complete list of keyboard shortcuts is available from Code Explorer window, tab Keyboard.

IDE Shortcuts

F1	Help
CTRL+N	New Unit
CTRL+O	Open
CTRL+F9	Compile
CTRL+F11	Code Explorer on/off
CTRL+SHIFT+F5	View breakpoints

Advanced Editor shortcuts

CTRL+SPACE	Code Assistant
CTRL+SHIFT+SPACE	Parameters Assistant
CTRL+D	Find declaration
CTRL+G	Goto line
CTRL+J	Insert Code Template
CTRL+<number>	Goto bookmark
CTRL+SHIFT+<number>	Set bookmark
CTRL+SHIFT+I	Indent selection
CTRL+SHIFT+U	Unindent selection
CTRL+ALT+SELECT	Select columns

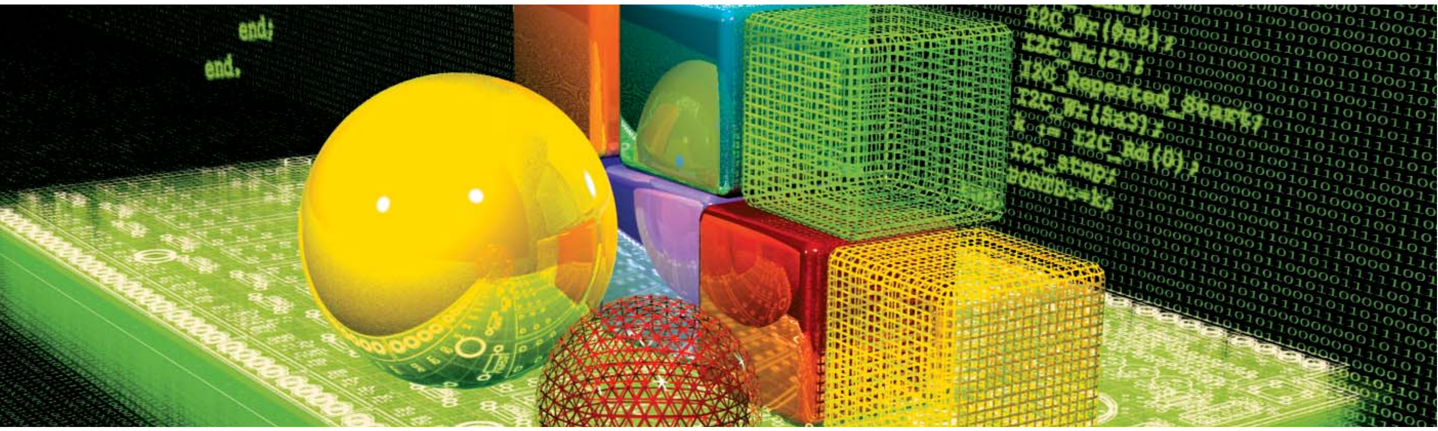
Debugger Shortcuts

F4	Run to Cursor
F5	Toggle breakpoint
F6	Run/Pause Debugger
F7	Step into
F8	Step over
F9	Debug
CTRL+F2	Reset

Basic Editor shortcuts

F3	Find, Find Next
CTRL+A	Select All
CTRL+C	Copy
CTRL+F	Find
CTRL+P	Print
CTRL+R	Replace
CTRL+S	Save unit
CTRL+SHIFT+S	Save As
CTRL+V	Paste
CTRL+X	Cut
CTRL+Y	Redo
CTRL+Z	Undo

CHAPTER 2



mikroBasic Reference

"Why Basic?", you may wonder. Well, the answer is simple: it is legible, easy-to-learn, procedural programming language, with sufficient power and flexibility needed for programming microcontrollers. Whether you had any previous programming experience, you will find that writing programs in mikroBasic is very easy. This chapter will help you learn or recollect Basic syntax, along with the specifics of programming PIC microcontrollers.

IDENTIFIERS

Identifiers are names used for referencing the stored values, such as variables and constants. Every program, procedure, and function must be identified (hence the term) by an identifier.

Rules

Valid identifier:
1. must begin with a letter of English alphabet or possibly the underscore (_)
2. can be followed by alphanumeric characters and the underscore (_)
3. may not contain special characters:
~ ! @ # $ % ^ & * ( ) + ` - = { } [ ] : " ; ' < > ? , . / | \

mikroBasic is not case sensitive. First, FIRST, and fIrST are an equivalent identifier.

Note

Elements ignored by the compiler include spaces, new lines, and tabs. All these elements are collectively known as the white space. White space serves only to make the code more legible; it does not affect the actual compiling.

Several identifiers are reserved in mikroBasic - you cannot use them as your own identifiers. Please refer to Kewords. Also, mikroBasic has several pre-defined identifiers. Pre-defined identifiers are listed in the chapter Library Functions and Procedures.

Examples

```
' Valid identifier examples

temperature_V1
Pressure
no_hit
dat
sum
vtext

' Some invalid identifier examples

7temp          ' cannot begin with a numeral
%higher        ' cannot contain special characters
xor            ' cannot match reserved word
j23.07.04      ' cannot contain special characters
```

KEYWORDS

The following keywords (reserved words) cannot be redefined or used as identifiers.

absolute	abs
and	array
asm	begin
boolean	case
char	chr
clear	const
div	do
double	else
end	exit
for	function
goto	gosub
if	in
int	interrupt
is	loop
mod	new
next	not
or	print
procedure	program
float	read
select	step
string	switch
then	to
module	until
include	dim
wend	while
with	xor

In mikroBasic, all SFR (Special Function Registers) are defined as global variables and represent special reserved words that cannot be redefined. For example - TMR0, PCL, STATUS, etc.

Also, mikroBasic has a number of predefined identifiers (refer to Library Routines). These can be replaced by your own definitions, but that would impede the functionality of mikroBasic.

DATA TYPES

Type determines the allowed range of values for variable, and which operations may be performed on it. It also determines the amount of memory used for one instance of that variable.

Simple

Type	Size	Range of values
byte	8-bit	0 .. 255
char*	8-bit	0 .. 255
word	16-bit	0 .. 65535
short	8-bit	-128 .. 127
integer	16-bit	-32768 .. 32767
longint	32-bit	-2147483648 ..147483647

* char type can be treated as byte type in every aspect

Structured

Array represents an indexed collection of elements of the same type, often called the base type. Base type can be any simple type.

String represents a sequence of characters. It is an array that holds characters and the first element of string holds the number of characters (max number is 255).

Sign

Sign is important attribute of data types, and affects the way variable is treated by the compiler.

Unsigned can hold only positive numbers:

byte	0 .. 255
word	0 .. 65535

Signed can hold both positive and negative numbers:

short	-128 .. 127
integer	-32768 .. 32767
longint	-2147483648 .. 214748364

Array

Array is a set of data stored in consecutive memory locations. Defining an array and manipulating its elements is simple. Elements of array are always of same data type (any simple).

```
dim  days_of_the_week      as byte[7]
dim  months                as byte[12]
dim  AD_Conversion_result  as word[10]
```

First declaration above generates 7 variables of byte type. These can be accessed by array name followed by number in the square brackets [] (this number is also known as index). Indexing is zero based, meaning that in our example, index spans numbers from 0 to 6. Instead of byte, you can define array of any other simple type (word, short, integer or longint).

Note that:

```
dim something as integer[10]
```

occupies 20 RAM locations (bytes), not 10.

Array and Operators

You can use any kind of operator with array elements - Arithmetic Operators, Logical (Bitwise) Operators, and Relation (Comparison) Operators. Technically, array element is treated as a simple type. Also, instead of a number, index can be any expression with result type of byte. For example:

```
m[a + b] = 90
m[1] = m[2] + 67
m[1] = m[2] div m[3]
```

Array and PIC

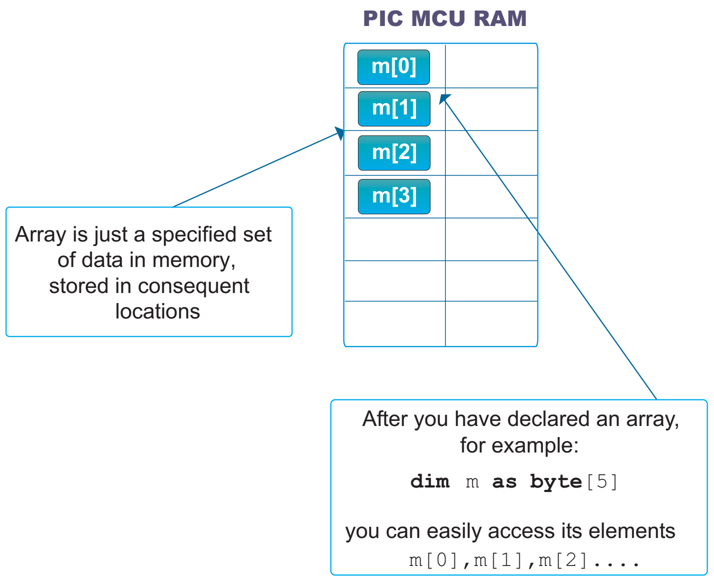
When you declare an array, mikroBasic allocates a certain amount of RAM for it. Elements of array consume consecutive RAM locations; in case of array of bytes, if the address of m[0] is 0x23, m[1] will be at 0x24, and so on.

Accessing these elements is almost as fast as accessing any variable of simple type. Instead of byte you can define array of any other simple type (word, short, integer or longint). Don't forget that you are restricted by the amount of free space in PIC RAM memory.

For example:

```
dim size as longint[10]
```

occupies 40 RAM locations (bytes).


PIC MCU RAM
m[0]
m[1]
m[2]
m[3]
Array is just a specified set of data in memory, stored in consequent locations
After you have declared an array, for example:
dim m as byte[5]
you can easily access its elements m[0],m[1],m[2]....

Example

```
program Array_test

dim  m  as byte[13]
dim  j  as byte[5]

j[0] = m[3] + 6
m[4] = m[2] mod 3
j[2] = not j[0]

if m[0] > 0 then
    m[1] = 9
  else
      m[1] = 90
end if
end.
```

Strings

String represents a sequence of characters. String type is similar to array, but can hold only characters.

```
dim M_name as string[16]
dim Start_message as string[6]
```

For each string declaration, compiler will reserve the appropriate amount of memory locations. For example, string M_name will take 16+1 locations; additional memory location is reserved to contain the length of the string. If we assign string literal to variable M_name, `M_name = "mik"`, then:

`M_name[0]` will be 3 (contains length of the string)
`M_name[1]` will be 'm'
`M_name[2]` will be 'i'
`M_name[3]` will be 'k'

and all other locations will be undefined.

Strings and assignment

Assignment operator can be used with string variables:

```
dim M as string[20]
    S as string[8]
main:
  M = "port"        ' Assign 'port' to M
  S = "port1"       ' Assign 'port1' to S
end.
```

Length

mikroBasic includes a built-in function Length for working with strings:

```
sub function Length(dim text as string) as byte
```

It returns string length as byte, and is quite useful for handling characters within string:

```
M = "mikroElektronika"
for i = 1 to Length(M)
   LCD_Chr(1,i,M[i])
next i
```

NUMERALS AND CHARACTER STRINGS

Numerals

Numeric constants can be represented in decimal, binary, or hexadecimal number system.

In decimal notation, they are represented as a sequence of digits, without commas or spaces, and can be prefixed with + or - operator to indicate the sign. Values default to positive (67258 is equivalent to +67258).

The dollar-sign prefix or a 0x prefix indicates a hexadecimal numeral (for example $8F or 0xC9).

The percent-sign indicates a binary numeral (for example %0101).

Example:

```
123                     Decimal
$1fc                    Hex
0xb9                    Hex
%101                    Binary
```

Character Strings

Character string, also called a string literal or a string constant, consists of a quoted string. Separators can be used only within quoted strings. A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written in one line and enclosed by apostrophes.

Quoted string with nothing between the apostrophes is a null string. Apostrophe itself cannot be used as part of the string. For example:

```
"mikroBasic"          ' mikroBasic
""                    ' null string
" "                   ' a space
```

Length of character string is the number of characters it consists of. Character string of length 1 is compatible with the char type. You can assign string literal to a string variable or to array of char.

CONSTANTS

Constant is data whose value cannot be changed during the runtime. Every constant is declared under unique name which must be a valid identifier. It is a good practice to write constant names in uppercase.

In mikroBasic, constants have to be of simple data type (no arrays or strings are allowed).

Example of constant declaration:

```
const MAXVALUE = 237
```

Constants can be used in any legal expression, but they cannot be assigned a new value. Therefore, they cannot appear on the left side of the assignment operator.

Note

If you frequently use the same value throughout the program and its value is fixed, you should declare it a constant (for example, maximum number allowed is 1000). This is a good practice since the value can be changed simply by modifying the declaration, instead of going trough the entire program and adjusting each instance manually. As simple as this:

```
const MAX = 1000
```

Constants and PIC

It is important to understand why constants should be used and how this affects the MCU. Using a constant in a program consumes no RAM memory. This is very important due to the limited RAM space (PIC16F877 has 368 locations/bytes).

Examples

```
const MaxAllowed = 234
const K_a = -32766
const Max = 1000
const Min = 0

ADC_Res = ADC_Read(2)
if ADC_Res > Max then
  portb = 1
else
  portb = Min
end if
```

Examples of invalid use

```
const 7time = 123
 ' Wrong constant name, it must be
 ' a valid identifier

const Max = 1123456
 ' Assigned value exceeds the allowed
 ' range for integer

Max = A
Max = 123
 ' You cannot assign new value to a constant,
 ' compiler will report an error
```

ARRAY CONSTANTS

Array constant is a set of data stored in ROM memory. Elements of array are always of the same data type (any simple).

```
const   CTEXT         as char[6]  = "trial1"
const   CMONTHS       as byte[12]  = (1,2,3,4,5,6,7,8,9,10,11,12)
const   MATCHVALUE    as word[8]  = (123,4566,56000,324,54,7878,876,0)
```

To declare an array constant holding numerals, enclose the values of the array's elements, separated by commas, in parentheses. For arrays of char type, use a string literal (text enclosed by quotes), as shown in the example above. Number of values in parentheses, i.e. a number of characters in quotes, must match the specified value in the square brackets.

Note

Accessing elements of array constant is simple, but be careful not to exceed the array range (if index is greater than array size, program won't work correctly). Indexing is zero based, so if you declare:

```
const width as byte[3] = (23,5,67)
```

then `width[0]` is the first element and index needs to be less or equal than 2.

Special case is array constant of char which holds the size of the array in the first location. Array constant of char is limited to 255 characters.

Array and Operators

You can use any kind of operator with array constant elements. Technically, array constant element is treated as a simple constant. Also, instead of a number, index can be any expression. But, you cannot assign new value to array constant element. For example:

```
m = CMONTHS[a + b]
m = CMONTHS[2] + 67
vv = vv div CMONTHS[3]
```

Also, it is possible for array constants of type char to copy whole array to variable of same type by simple assignment. For example:

```
const CTXT as char[4] = "dota"
dim    txt as char[4]
...
txt = CTXT    ' this is legitimate
```

Array Constants and PIC

Elements of array constant are located in R0M. PIC16 family restricts array constants to 255 elements of byte type, while PIC18 family is limited only by ROM size.

Instead of byte, you can define array of any other simple type (`word, short, integer` or `longint`). Don't forget that you are restricted by the amount of PIC ROM memory.

For example:

```
const foo as longint[5] = (36732, 32442, 19901, 82, 27332724)
```

Example

```
program Array_test

const m  as byte[3] = (0,1,2)
dim   j  as byte[5]
main:
  j[0] = m[3] + 6
  j[4] = m[2] mod 3
  j[2] = not j[0]

  if m[0] > 0 then
    j[1] = 9
  else
    j[1] = 90
  end if
end.
```

SYMBOLS

Symbol makes possible to replace expression with a single identifier alias. Use of symbols increases the reusability and flexibility of code.

BASIC syntax restricts you to single line expressions, allowing shortcuts for constants, simple statements, function calls, etc. Scope of symbol identifier is a whole source file in which it is declared.

Symbol is declared as:

```
symbol alias = single_line_expression
```

where *alias* must be a valid identifier which you will be using throughout the code.

Symbols and PIC

Using a symbol in a program technically consumes no RAM memory - compiler simply replaces each instance of a symbol with the appropriate code from the declaration.

Example

```
symbol MaxAllowed = 234          ' symbol as alias for numeral
symbol PORT = PORTC              ' symbol as alias for SFR
symbol DELAY1S = delay_ms(1000)  ' symbol as alias for proc. call

if  teA > MaxAllowed then
     teA = teA - 100
end if
PORT.1  = 0
DELAY1S
  ...
```

VARIABLES

Variable is data whose value can be changed during the runtime. Every variable is declared under unique name which must be a valid identifier. This name is used for accessing the memory location occupied by the variable.

Variable can be seen as a container for data and because it is typed, it instructs the compiler how to interpret the data it holds. For more details refer to Data Types and Type Conversion.

For more information on variables' scope refer to the chapter Scope (Variable Visibility).

In mikroBasic, variable needs to be declared before it can be used. Specifying a data type for each variable is mandatory. Basic syntax for variable declaration is:

```
dim variable as type
```

where *variabe* is any valid identifier, and *type* can be any valid data type.

For example:

```
dim A as byte      ' declare variable tA of byte type
dim BB as word     ' declare variable tB of word type
```

Variables and PIC

Every declared variable consumes part of MCU RAM memory. Data type of variable determines not only the allowed range of values, but also the space variable occupies in RAM memory. Bear in mind that operations using different types of variables take different time to be completed. For example:

Variable A (byte) occupies 1 byte (8 bit) of RAM memory, while variable BB (word) occupies 2 bytes (16 bit) of RAM memory.

Therefore, A = A + A is faster to execute than BB = BB + BB.

Note

mikroBasic recycles local variable memory space - local variables declared in different functions and procedures share same memory space, if possible.

Additional info

Variable declaration has to be properly placed to have a correct meaning. Variables can be declared in a program block or implementation section of a module. Variable declaration must be placed ahead of the keyword begin. You can also declare variables in function or procedure block. Refer to Program Organization, and see the following example.

There is no need to declare PIC SFR (Special Function Registers), as they are already declared as global variables of byte type - for example: TMR0, PCL, STATUS, PORTA, TRISA, etc. These variables may be used anywhere within the code.

For closer information on how to use variables and build valid expressions refer to the chapter Operators.

Examples

```
program TRIAL

include "other.pbas"

  ' You can declare variables in the program block

dim tA as integer
dim tD as integer
dim tF as integer
dim tR as word
dim tT as word
dim tY as word

main:
   tA = tD and tF

  ' STATUS and TMR0 are PIC registers
   tR = STATUS and $03
   TMR0 = 45
end.

...

module other

  ' You can declare variables at the
  ' beginning of a module

dim Sss as longint
dim Ddd as longint
...
end.


sub function Sum( dim R as byte) as byte
  ' You can also declare variables in
  ' function or procedure block.

dim B as char
dim K as byte

  ...
end sub
```

Any valid variable can be used after it has been declared:

```
tA = 36
 ' assign new value to the existing variable
tC = tA + tB
 ' perform any kind of arithmetical or
 ' logical operation

tE = pr_function(1,tA)
 ' pass variable to function or procedure,
 ' by value or address

pr_procedure(1,2,tD,tE)

 ' use them in conditional and/or
 ' loop statements and more ...

select case tb
case 1
     tA = tD + 4
case 2
     tB = tC + 6
case 3
 tC = $ff
 tb = tc - tA
case else
      pr_procedure(1,2,tD,tE)
end select

for tA = 0 to 7
  tC = tB >> 1
next tA
```

COMMENTS

Comments are text that is added to the code for purpose of description or clarification, and are completely ignored by the compiler.

```
' Any text between an apostrophe and the end of the
' line constitutes a comment. May span one line only.
```

It is a good practice to comment your code, so that you or anybody else can later re-use it. On the other hand, it is often useful to comment out a troublesome part of the code, so it could be repaired or modified later.

mikroBasic Code Editor features syntax color highlighting - it is easy to distinguish comments from the code due to different color, and comments are also italicized.

Example

```
dim teC as byte   ' declare variable teC,
                  ' variable type is byte
dim teB as byte
dim teA as byte


main:
teC = 12           ' assign value 12 to variable C

if teA > 0 then
    teC = 9
  else
    teA = teB
end if
       ' you can also comment out part of the
       ' code you don't want to compile:

       ' E = gosub pr_function(1,2)

       ' This function call won't be compiled
end.
```

EXPRESSIONS

Expression is a construction that returns a value. The simplest expressions are variables and constants, while more complex expressions are constructed from simpler ones using operators, function calls, indexes, and typecasts.

Rules for creating legal expressions are presented in chapter Implicit Conversion and Legal Expressions.

These are all expressions:

```
X                    ' variable
15                   ' integer constant
Calc(X, Y)           ' function call
X * Y                ' product of X and Y
```

Legal Expressions

We will present in short notice rules for building expressions here. But, we should recollect some information beforehand:

Simple data types include: byte, word, short, integer and longint.

Byte and word types hold only positive values so we'll call them unsigned. Ranges are:

byte	0 .. 255
word	0 .. 65535

Short, integer, and longint types can hold both positive and negative numbers so we'll call them signed. Ranges are:

short	-128 .. 127
integer	-32768 .. 32767
longint	-2147483648 .. 214748364

You cannot mix signed and unsigned data types in expressions with arithmetic or logical operators. You can use explicit conversion though.

```
dim Sa as short
dim A as byte
dim Bb as word
dim Sbb as integer
dim Scccc as longint
...
A = A + Sa          ' compiler will report an error
A = A and Sa        ' compiler will report an error

 ' But you can freely mix byte with word..
         Bb = Bb + (A * A)

 ' ..and short with integer and longint
         Scccc = Sbb * Sa + Scccc
```

You can assign signed to unsigned or vice versa only using the explicit conversion.

```
Sa = short(A)
 ' this can be done; convert A to short,
 ' then assign to Sa

Sa = A
 ' this can't be done,
 ' compiler will report an error
```

Relation operators can freely be used even when mixing signed and unsigned data. For example:

```
if Sa > B then
  Sa = 0
end if
```

Note

Comparing variable or constant to variable or constant will always produce correct results.

Comparing expressions requires a little more attention. For more information refer to the chapter Implicit Conversion and Relation Operators.

DECLARATIONS AND STATEMENTS

Aside from the `include` clause, program consists entirely of declarations and statements, which are organized into blocks.

Declarations

Names of variables, constants, types, procedures, functions, programs and units are called identifiers (numeric constant like 321 is not an identifier).

Identifiers need to be declared before you can use them. Only exceptions are few predefined types, library functions and procedures, PIC MCU SFR (PIC Special Function Registers), and constants; these are understood by the compiler automatically.

Declaration defines an identifier and, where appropriate, allocates memory for it. For example:

```
dim Right as word
```

declares a variable called Right that holds a word value, while:

```
sub function Each(dim X as integer, dim Y as integer) as integer
```

declares a function called Each which collects two integers as arguments and returns an integer.

Each declaration ends with a semicolon (separator). When declaring several variables, constants, or types at the same time, you need to write the appropriate reserved word only once :

```
dim Height as integer
dim Description as string[10]
```

The syntax and placement of a declaration depends on the kind of identifier you are defining. In general, declarations take place only at the beginning of a block, or at the beginning of the implementation section of a unit (after the `include` clause). Specific conventions for declaring variables, constants, types, functions, and so forth can be found in the appropriate chapters.

Statements

Statements define algorithmic actions within a program. Simple statements - like assignments and procedure calls - can be combined to form loops, conditional statements, and other structured statements. Refer to Implicit Conversion and Assignment.

Simple Statements

Simple statement does not contain any other statements. Simple statements include assignments, and calls to procedures and functions.

Structured Statements

Structured statements are constructed from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

Conditional statements `if` and `case` execute at most one of their constituents, depending on a specified criteria.

Loop statements `repeat`, `while`, and `for` execute a sequence of constituent statements repeatedly.

DIRECTIVES

Directives are words of special significance for the mikroBasic, but unlike other reserved words, appear only in contexts where user-defined identifiers cannot occur.

You cannot define an identifier that looks exactly like a directive.

Overview

Directive	Meaning
Absolute	specify exact location of variable in RAM
Org	specify exact location of routine in ROM

Absolute directive specifies the starting address in RAM for variable (if variable is multi-byte, higher bytes are stored at consecutive locations).

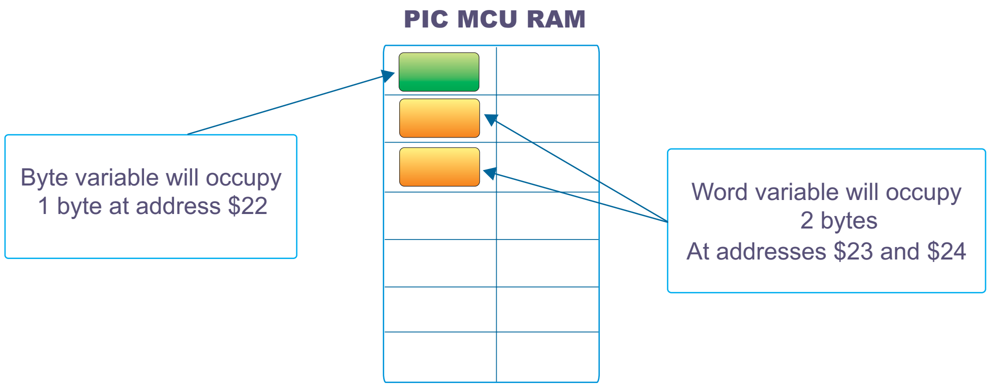
Org directive specifies the starting address of routine in ROM. For PIC16 family, routine must fit in one page - otherwise, compiler will report an error.

Directive `absolute` is appended to the declaration of variable:

```
dim rem as byte absolute $22
 ' Variable will occupy 1 byte at address $22

dim dot as word absolute $23
 ' Variable will occupy 2 bytes at addresses $23 and $24
```

Directive `org` is appended to the declaration of routine:

```
sub procedure test org $200
 ' Procedure will start at address $200
...
end sub
```


PIC MCU RAM
Byte variable will occupy 1 byte at address $22
Word variable will occupy 2 bytes At addresses $23 and $24

Important

We recommend careful use of absolute directive, because you may overlap two variables by mistake. For example:

```
dim  Ndot as byte absolute $33
  ' Variable will occupy 1 byte at address $33

dim Nrem as longint absolute $30
  ' Variable will occupy 4 bytes at $30, $31, $32, $33,
  ' so changing Ndot changes Nrem highest
  ' byte at the same time
```

Runtime Behavior

mikroBasic uses internal algorithm to distribute variables within RAM. If there is a need to have variable at specific predefined address, use the directive `absolute`. Also if, for some reason, you want to overlap existing variables, use the directive `absolute`.

Example

```
program lite

  ' example for P16F877A

dim image_trisa as byte absolute 133

main:
   image_trisa = $ff
end.
```

PROCEDURES AND FUNCTIONS

Procedures and functions, collectively referred to as routines, are self-contained statement blocks that can be called from different locations in a program. Function is a routine that returns a value when it is executed. Procedure is a routine that does not return a value.

Once these routines have been defined, you can call them once or multiple times. Procedure is called upon to perform a certain task, while function is called to compute a certain value. Function calls, because they return a value, can be used as expressions in assignments and operations.

Procedures

Procedure declaration has the form:

```
sub procedure procedureName(parameterList)
  localDeclarations
  statements
end sub
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that are executed upon the calling the procedure, and (*parameterList*) and *localDeclarations* are optional declaration of variables and/or constants.

```
sub procedure pr1_procedure(dim par1 as byte, dim par2 as byte,
                           dim byref vp1 as byte, dim byref vp2 as byte)
dim locS as byte
  par1 = locS + par1 + par2
  vp1  = par1 or par2
  vp2  = locS xor par1
end sub
```

par1 and *par2* are passed to the procedure by the value, but variables marked by keyword `byref` are passed by the address.

This means that the procedure call

```
pr1_procedure(tA, tB, tC, tD)
```

passes tA and tB by the value: it first creates par1 = tA and par2 = tB, then manipulates par1 and par2 so that tA and tB remain unchanged;

passes tC and tD by the address: whatever changes are made upon vp1 and vp2 are also made upon tC and tD.

Note that a procedure without parameters can be substituted by label which marks the beginning of "procedure" and keyword `return` that marks the end of "procedure". To call such subroutine, use the keyword `gosub`. These subroutines must be placed between the label `main:` and the `end` of the source file.

```
main:
if PORTC.1 = 1 then
  gosub TogglePortb
end if

...                  ' some code

TogglePortb:         ' routine
  portb = not portb
return
end.
```

Functions

Function declaration is similar to procedure declaration, except it has a specified return type and a return value. Function declaration has the form:

```
sub function functionName(parameterList) as returnType
  localDeclarations
  statements
end sub
```

where *functionName* is any valid identifier, *returnType* is any simple type, *statements* is a sequence of statements to be executed upon calling the function, and (*parameterList)* and *localDeclarations* are optional declarations of variables and/or constants.

In mikroBasic, use the keyword Result to assign the return value of a function.

Example:

```
sub function some_function(dim par1 as byte, dim par2 as word) as word
dim locS as word
  locS = par1 + par2
  Result = locS
end sub
```

Function Calls

As functions return a value, function calls are technically expressions. For example, if you have defined a function called Calc, which collects two integer arguments and returns an integer, then the function call `Calc(24, 47)` is an integer expression. If I and J are integer variables, then `I + Calc(J, 8)` is also an integer expression. Here are a few examples of function calls:

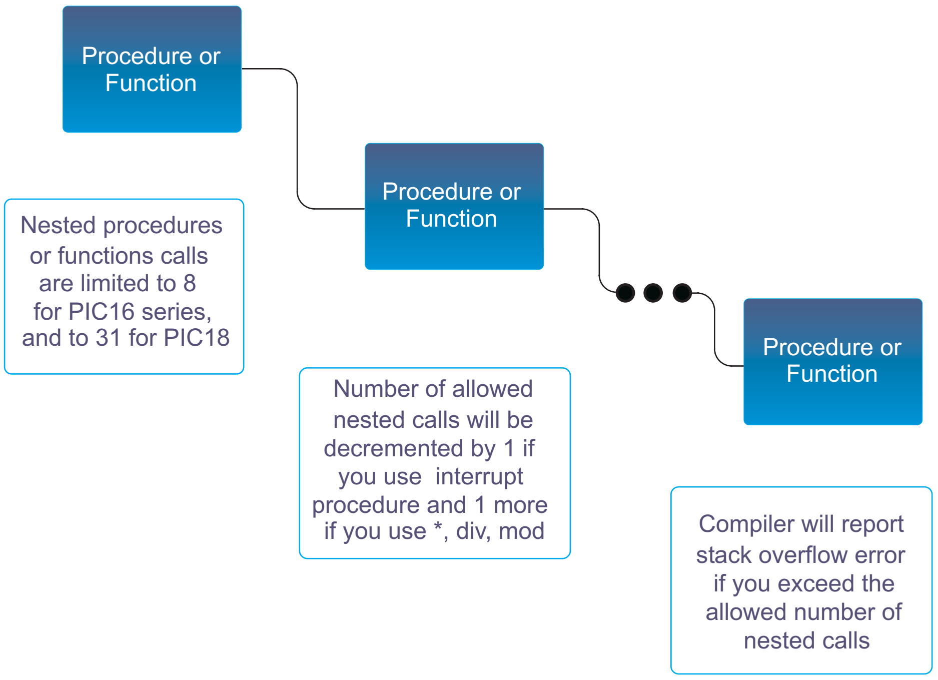
```
Sum(tA,63)
Maximum(147,J)
GetValue
```

Important

Note that *cross-calling* and *recursive calls* are not allowed in mikroBasic. Cross-calling is an instance of procedure A calling procedure B, and then procedure B calling procedure A. Recursive call is an instance of procedure or function calling itself. Compiler will report error if cross-calling or recursive calls are encountered in the code.

mikroBasic has a number of built-in and predefined library routines. For example, procedure `interrupt` is the interrupt service routine.

Nested calls are limited to 8-level depth for PIC16 series and 31-level depth for PIC18 series. Nested call represent call of another function or procedure within a function or procedure. For closer information, refer to the chapter PIC Specifics.


Procedure or Function
Procedure or Function
Procedure or Function
Nested procedures or functions calls are limited to 8 for PIC16 series, and to 31 for PIC18
Number of allowed nested calls will be decremented by 1 if you use interrupt procedure and 1 more if you use *, div, mod
Compiler will report stack overflow error if you exceed the allowed number of nested calls

Example

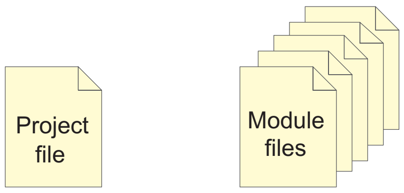
```
sub function mask(dim byref num as byte) as byte
                                 ' This function returns code for digit
  select case num                '  for common cathode 7 seg. display.
  case  0  result = $3F
  case  1  result = $06          ' Note that the value of result is not
  case  2  result = $5B          '  initialized for values greater than 9
  case  3  result = $4F
  case  4  result = $66
  case  5  result = $6D
  case  6  result = $7D
  case  7  result = $07
  case  8  result = $7F
  case  9  result = $6f
  end select                     ' case end
end sub
```

Example of Stack Overflow

```
sub program Stack_overflow

sub procedure interrupt
  nop
end sub

sub procedure proc0
  nop
end sub

sub procedure proc1
  proc0
end sub

sub procedure proc2
  proc1
end sub

sub procedure proc3
  proc2
end sub

sub procedure proc4
  proc3
end sub

sub procedure proc5
  proc4
end sub

sub procedure proc6
  proc5
end sub

sub procedure proc7
  proc6
end sub

main:
  proc7
end.
```

MODULES

Each project consists of a single project file, and one or more module files. To build a project, the compiler needs either a source file or a compiled file for each module.



Every project consist of single project file and one or more module files

Each module is stored in its own file and compiled separately; compiled modules are linked to create an application.

Modules allow you to:

- Break large programs into parts that can be edited separately.
- Create libraries that can be used in different programs.
- Distribute libraries to other developers without disclosing the source code.

In mikroBasic programming, all source code including the main program is stored in .pbas files.

If you perform circular unit references, compiler will give a warning. A simple instance of circular unit references would be, for example, situation in which Module1 uses Module2, but in the same time it is specified that Module2 uses Module1.

Newly created blank unit contains the following :

```
module Module1

end.
```

Unit Influence on Scope (Visibility)

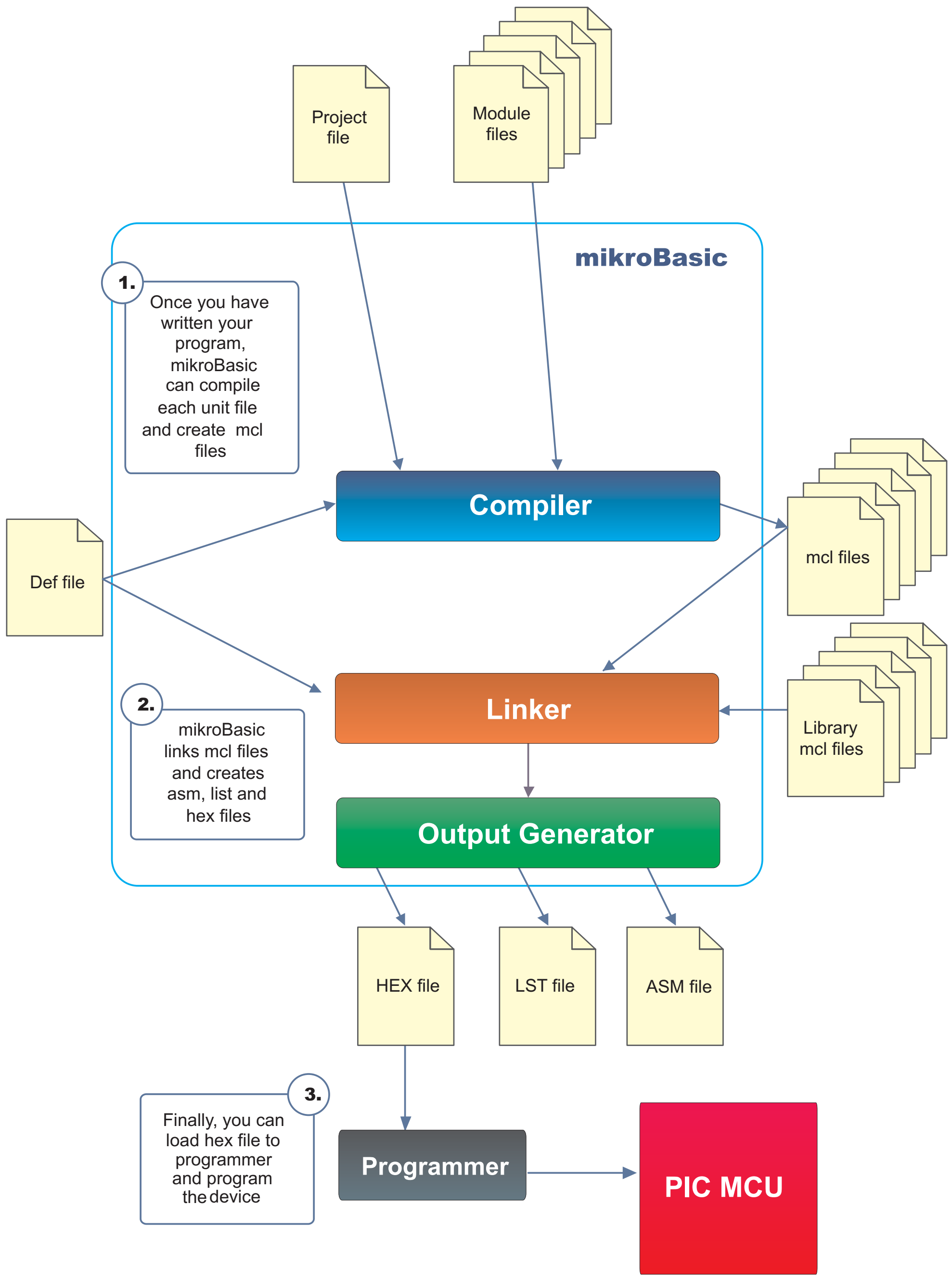
mikroBasic variables defined at the beginning of the module are global hidden variables. When you declare an identifier at the beginning of a module, you cannot use it outside the unit, but you can use it in any routine defined within the module. Refer to chapter Scope (Variable Visibility) for more details.

Main Unit File

mikroBasic application has one main module file and none or more module files. All source files have the same extension (`pbas`). Main file is identified by the keyword `program` at the beginning; other module files have the keyword `module` at the beginning.

```
program Project1
include "additional.pbas"
dim tA as word
dim tB as word

main:
  tA = sqrt(tb)
end.
```

Keyword `include` instructs the compiler which file to compile. If you want to include a module, add the keyword `include` followed by the quoted name of the file. The example above includes the module `additional.pbas` in the program file.


Project file
Module files
mikroBasic
1.
Once you have written your program, mikroBasic can compile each unit file and create mcl files
Compiler
mcl files
Def file
2.
mikroBasic links mcl files and creates asm, list and hex files
Linker
Library mcl files
Output Generator
HEX file
LST file
ASM file
3.
Finally, you can load hex file to programmer and program the device
Programmer
PIC MCU

SCOPE (IDENTIFIER VISIBILITY)

Scope, or identifier visibility, determines if identifier can be referenced in certain part of the program code. Location of identifier declaration in the code determines its scope. Identifiers with narrower scope - especially identifiers declared in functions and procedures - are sometimes called local, while identifiers with wider scope are called global.

All functions and procedures are visible in the whole project, and they are visible in any part of the program or any module. Constants not local for a procedure or function are also visible in the whole project. Local constants are visible only in procedure or function body in which they are declared.

Rules for determining the variable identifier scope are summarized below:

- If the identifier is declared in the declaration of a main module, it is visible from the point where it is declared to the end of the module.

- If the identifier is declared in the declaration of function, or procedure, its scope extends from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.

- If the identifier is declared in the implementation section of a module, but not within the block of any function or procedure, its scope extends from the point where it is declared to the end of the module. The identifier is available to any function or procedure in the module.

PIC SFR (Special Function Registers) such as TMR0, PORTA, etc, are implicitly declared as global variables of byte type. Their scope is the entire project and they are visible in any part of the program or any module.

For example, in a function declaration:

```
sub function Com(dim R as byte) as byte
dim B as char
dim K as byte
   ...
end sub
```

first line of the declaration is the function heading . B and K are local variables; their declarations apply only to the Com function block and override - in this routine only - any declarations of the same identifiers that may occur in the program module or at beginning of a module.

PROGRAM ORGANIZATION

Program elements (constants, variables and routines) need to be declared in their proper place in the code. Otherwise, compiler may not be able to comprehend the program correctly.

Organization of the main unit should have the following form:

```
program program_name                ' program name
include ...                         ' include other units

symbol ...                          ' symbols declaration
const ...                           ' constants declaration
dim ...                             ' variables declaration

sub procedure procedure_name        ' procedures declaration
    ...
end sub

sub function function_name          ' functions declaration
    ...
end sub

main:                               ' program must start with label
                                    ' main

  ...                               ' program body

end.                                ' end of program
```

Organization of other modules should have the following form:

```
module unit_name                    ' unit name
include ...                         ' include other units

symbol ...                          ' symbols declaration
const ...                           ' constants declaration
dim ...                             ' variables declaration

sub procedure procedure_name        ' procedures declaration
    ...
end sub

sub function function_name          ' functions declaration
    ...
end sub

end.                                ' end of module
```

TYPE CONVERSION

mikroBasic is capable of both implicit and explicit conversion of data types. *Implicit* conversion is the one automatically performed by compiler. On the other hand, *explicit* conversion is performed only on demand issued by user.

This means that you can, obeying a few rules, combine simple data types with any operators to create legal expressions and statements. Refer to Data Types if you are not familiar with data types supported by mikroBasic.

As stated in the chapter about operators, you cannot mix signed and unsigned data types in expressions that contain arithmetic or logical operators. You can assign signed to unsigned or vice versa only using the explicit conversion.

Implicit Conversion

- Implicit conversion takes place between byte and word, so you can combine byte and word with any operators to form legal expressions.

- Implicit conversion takes place between short, integer and longint so you can combine short, integer and longint with any operators to form legal expressions.

- Relation operators can be used without any restraints. Smart algorithm governing relation operators allows comparing any two data types.

- The compiler provides automatic type conversion when an assignment is performed, but does not allow to assign signed data type to unsigned and vice versa.

You can find more information on implicit conversion in chapters Assignment and Implicit Conversion, and Implicit Conversion and Legal Expressions.

Explicit Conversion

Explicit conversion can be executed at any point by inserting type (byte, word, short, integer, or longint) ahead of the expression to be converted. The expression must be enclosed in parentheses. You can't execute explicit conversion on the operand left of the assignment operator.

Special case is conversion between signed and unsigned. It is important to understand that explicit conversion between signed and unsigned data does not change binary representation of data; it merely allows copying of source to destination.

Example 1:

```
dim tA as byte
dim tB as byte
dim tC as byte

if tA + tB > tC then
  tA = 0
end if
```

This could be wrong, because there is an expression on the left. Compiler evaluates it, and treats it as a variable of type that matches type of tA or tB (the larger of the two); in this case - a byte.

```
tA = 250
tB = 10
tC = 20

if tA + tB > tC then
  tA = 0
end if
```

In this case, since the result of the expression is treated as byte, we get that 250 + 10 is lower than 20. Actually, the result of the expression is truncated to byte: 250 + 10 is 4, and 4 is lower than 20.

But if we wrote it like this:

```
if word(tA + tB) > tC then
  tA = 0
end if
```

.it would be correct, because we have explicitly instructed the compiler to treat tA + tB as a word. Hence, the result will equal 260 and greater than 20, returning the expected result.

Example 2:

Explicit conversion can also be used when you are sure which type you want to convert expression to. Consider the following lines:

```
dim tA as byte
dim tB as byte
dim tC as byte
dim A_ as short
dim B_ as short

tA = byte(A_)
B_ = short(tA + tB * tC)
```

It is important to understand that explicit conversion between signed and unsigned data does not change binary representation of data; it only allows copying source to destination. Thus, if A_ was -1, its binary representation would be 11111111, and A would become 255.

Even if you have ordered the explicit conversion, compiler will perform implicit if necessary.

Example 3:

You cannot execute explicit conversion on the operand left of the assignment operator:

```
word(b) = Bb   ' compiler will report an error.
```

ASSIGNMENT AND IMPLICIT CONVERSION

Overview

mikroBasic provides automatic type conversion every time an assignment is performed. But it does not allow assigning signed data to unsigned and vice versa, because there is a significant risk of losing information.

Implicit conversion takes place when assignment is performed:

between byte and word
between short, integer, and longint

Notes

Destination will store the correct value only if it can properly represent the result of expression (that is, if the result fits in destination range).

Feel free to use operands of any size under the defined rules, but keep in mind that the PIC is optimized to work with bytes. Every operation involving more complex data types (word, integer or longint) will take more run time and more memory. So for the best possible results, use as small destinations and operands as you can.

Examples

```
A = B
```

If A and B are of the same type, value of B is simply assigned to A. More interesting case is if A and B are of different types:

```
dim A as byte
dim B as word
...
B = $ff0f
A = B    ' A becomes $0f, higher byte $ff is lost
```

If A is more complex than B, then B is extended to fit the correct result:

```
dim A as word
dim B as byte
...
B = $ff
A = B    ' A becomes $00ff
```

For signed types :

```
dim B_ as integer
dim A_ as short
...
A_   = -10
B_   = A_    ' B becomes -10
```

In hex representation, this means that the higher byte is sign extended.

```
C = expression
```

Calculated value of the expression will be assigned to the destination. Part of the information may be lost if the destination cannot properly represent the result of the expression (i.e. if result can't fit in range of destination data type). Browse through examples for more details.

For example (this is correct):

```
C = A + B
```

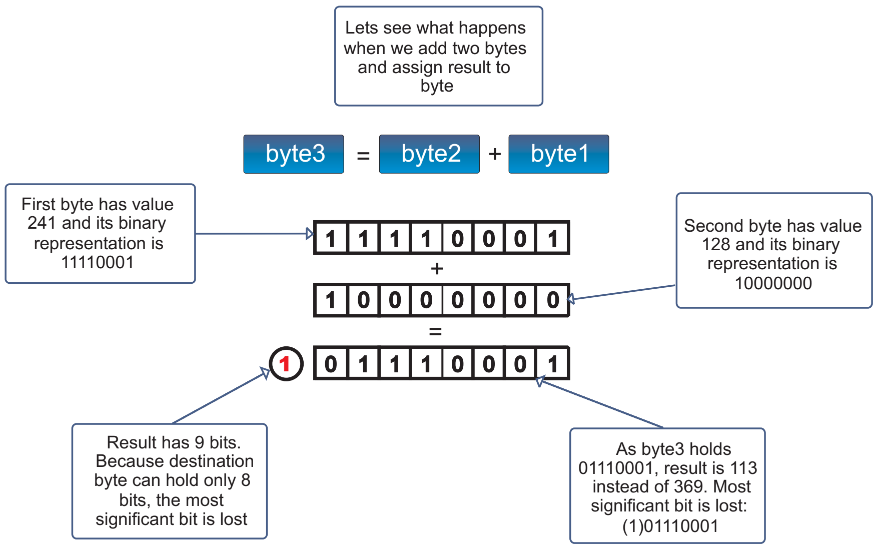
C is byte, so its range is 0 .. 255. If (A + B) fits in this range you will get the correct value in C.

```
A = 123
B = 90

C = A + B   ' C becomes 213
```

But what happens when A + B exceeds the destination range? Let's assume the following:

```
A = 241
B = 128

C = A + B  ' C becomes 113, obviously incorrect
```

See the following figure for closer explanation.


Lets see what happens when we add two bytes and assign result to byte
byte3 = byte2 + byte1
First byte has value 241 and its binary representation is 11110001
1 1 1 1 0 0 0 1
+
1 0 0 0 0 0 0 0
Second byte has value 128 and its binary representation is 10000000
=
1
0 1 1 1 0 0 0 1
Result has 9 bits. Because destination byte can hold only 8 bits, the most significant bit is lost
As byte3 holds 01110001, result is 113 instead of 369. Most significant bit is lost: (1)01110001

In order to fully understand this, we should recollect the data types.

Data type determines not only the range of values variable can hold, but also the amount of RAM space it consumes. This is fundamental in practical programming.

Let's assume that our destination variable C is a byte, consuming 8 bits of PIC RAM, spanning values 0 to 255. Now observe what really happens inside the PIC: the result should be 369, but in binary representation it equals (1)01110001. Because C is limited to 8 bits it will store the lower 8 bits while dropping the rest of the information (the most significant bit). 01110001 equals 113 in decimal representation.

```
dim  testA as byte
dim  testB as byte
dim  Cc     as word

main:
   testA = 250
   testB = 10

   Cc = testA + testB  ' this will always be correct because
                      ' range for Cc is 0..65535 and maximum result
                      ' of adding two bytes is only 255 + 255 = 510
end.
```

As already stated, destination will store the correct value only if it can properly represent the result of the expression (that is, the result fits in destination range).

```
dim testA as byte
dim testB as byte
dim     Cc as word
dim     Sa as short
dim     Sb as short
dim     Sc as short
dim    Saa as integer
dim Sbbbb as longint

main:
  testA = 250
  testB = 10
  Cc = testA * testB + testB    ' Cc becomes 2510;
  Sb = 120
  Sc = -100
  Sa = Sb + Sc      ' Sa  becomes  20;
  Sa = Sb - Sc      ' Sa is short with range -127..128,
                    ' thus, instead of 220,
                    ' Sa becomes -36, because only
                    ' lower 8 bits are registered

  Saa = (Sb * Sc) div 13
  ' Saa becomes -923

  Sbbbb = integer(Sb * Sc) * Sc
  ' Sbbbb becomes 1200000
end.
```

IMPLICIT CONVERSION AND LEGAL EXPRESSIONS

Overview

To create legal expressions, you can:

1. combine byte and word with any operators,

2. combine short, integer, and longint (note that longint does not employ *, div, mod) with any operators,

3. use Relation operators

```
        expression1 (relation operator) expression2
```

Expression1 and expression2 can be any legal expressions. Be sure to understand how implicit conversion works with relation operators.

Implicit Conversion and Relation Operators

Comparing variable or constant to variable or constant always produces correct results.

Comparing expressions requires a little more attention.

```
expression1 (relation operator) expression2
```

Expressions can be any legal expressions created with arithmetical or logical operators. Every expression no matter how complex, can be decomposed to a number of simple expressions. Simple expression is expression composed of just one operator and its operands. Operator is logical or arithmetical. Examine the rules presented below.

Rules for Comparing Expressions

1. Complex expression is decomposed to a number of simple expressions, with respect to operator precedence and overriding parenthesis.

2. Simple expression is now treated in the following manner: if operands are of the same type, operation is performed, assuming that the result is of the same type.

3. If operands are not of the same type, then less complex operand (speaking in terms of data range) is extended:

- If one operand is byte and another is word, byte is converted in word.

- If one operand is short and another is integer, short is converted to integer.

- If one operand is short and another is longint, short is converted to longint.

- If one operand is integer and another is longint, integer is converted to longint.

4. After the first expression is decomposed to simpler ones, each of these simpler ones is evaluated abiding the rules presented here.

Expression `a + b + c` is decomposed like this:

First evaluate `a + b` and get (`value of a + b`)

This gives us another simple expression
`(value of a + b) + c`

Let's assume a and b are bytes and c is word, with values:

```
a = 23
b = 34
c = 1000
```

Compiler first calculates value of `a + b` and assumes that the result is byte:
`a + b` gives 57.

As c is of word type, result of adding `a + b` is casted to word and then added to c:
`57 + c` is 1057.

Signed and unsigned numbers cannot be combined using arithmetical and logical operators. Rules presented above are not valid when assigning expression result to variable.

```
r = expression
```

Refer to chapter Assignment and Implicit Conversion for details.

Note

When adding operands of the same type and assigning value to third operand, incorrect value may be proceeded if the result exceeds range of declared data type. Similar rules apply to other arithmetical operators.

For example, if a and b are bytes, and cc is word:

```
a   = 56
b   = 200
cc  = 1000
```

a + b equals 1, because result type is assumed to be same as the operands' type (byte).

Added to cc, we get 1001, instead of the expected 1256.

Solution is to simply instruct the compiler to evaluate expression as you explicitly define. For example, you could explicitly cast the expression, like this:

```
word(a + b + c).
```

As result fits in word range, we get 1256 as expected.

For more details, refer to chapter Explicit Conversion.

Comparing variables and constants always produces the correct results regardless of the operands' type.

Example

```
if A + B > A then
```

First, compiler evaluates the expression on the left. During runtime, result is stored in a variable of type that matches the largest data type in the expression. In this case it is byte, as variables A and B are both bytes.

This is correct if the value does not exceed range 0..255, that is, if A + B is less than 255.

Let's assume Aa is of word type :

```
if Aa + B > A ...
```

First, compiler evaluates the expression on the left. The result value is treated as type that matches the largest data type in the expression. Since Aa is word and B is byte, our result will be treated as word type.

This is correct if the value does not exceed range 0..65535, i.e. if Aa + B is less than 65535.

```
  ' if tC is less than zero, tC = -tC

if  tC < 0  then
  tC = -tC
end if


  ' Stay in loop while C is not equal to variable
  ' compare_match; increment C in every cycle

while tC <> compare_match
   tC = tC + 1
wend
```

OPERATORS

There are three types of operators in mikroPascal:

Arithmetic Operators
Logical (Bitwise) Operators
Relation Operators (Comparison Operators)

Operator Precedence

Operator	Priority
not	first (highest)
*, div, mod, and, shl, shr	second
+, -, or, xor	third
=, <>, <, >, <=, >=	fourth (lowest)

In complex expressions, operators with higher precedence are evaluated before the operators with lower precedence; operators of equal precedence are evaluated according to their position in the expression starting from the left.

Example 1:

```
B and T + A
 ' (bitwise and) B and T, then add A to the result;
 ' and is performed first, because it has precedence over +.
```

Example 2:

```
A - B + D
 ' first subtract B from A, then add D to the result;
 ' - and + have the equal precedence, thus the operation on
 ' the left is performed first.
```

Example 3:

```
 ' You can use parentheses to override these precedence rules.
 ' An expression within  parentheses is evaluated first, then
 ' treated as a single operand. For example:

(A + B) * D
 ' multiply D and the sum of A and B.

A + B * D
 ' first multiply B and D and then add A to the product.
```

Rules for Creating Legal Expressions

You cannot mix signed and unsigned data types in expressions with arithmetic or logical operators. If you need to combine signed with unsigned, you will have to use explicit conversion.
Example:

```
dim  Sa as short
dim  teA as byte
dim  Bb as word
dim  Sbb as integer
dim  Scccc as longint
...
teA = teA + Sa      ' compiler will report an error
teA = teA and Sa    ' compiler will report an error

 ' But you can freely mix byte and word . .
    Bb = Bb + (teA * teA)

 ' . . and short with integer and longint;
    Scccc = Sbb * Sa + Scccc
```

You can assign signed to unsigned, or unsigned to signed only using the explicit conversion. More details can be found in chapter Implicit Conversion and Assignment Operator.

```
Sa = short(teA)
 ' this can be done

Sa = teA
 ' this can't be done, compiler will report an error
```

Relation operators can be used with all data types, regardless of the sign.
Example:

```
if Sa > teA then
    Sa = 0
end if
```

Notes for Relation Operators

Comparing variable or constant to variable or constant will always produce correct results.

Comparing expressions requires a little more attention - when compiler is calculating value of the expression to be compared, it first has to evaluate the expression. If the result of the expression exceeds the range of the largest data type in the expression, comparison will most likely be inaccurate. This can be avoided by using the explicit conversion.

More details can be found in chapter Implicit Conversion and Relation Operators.

Runtime Behavior

PIC MCUs are optimized for working with bytes. It takes less time to add two bytes than to add two words, naturally, and similar pattern is followed by all the other operators. It is a good practice to use byte or short data type whenever appropriate. Although the improvement may seem insignificant, it could prove valuable for applications which impose execution within time boundaries.

This is a recommendation which shouldn't be followed too literally - word, integer and longint are indispensable in certain situations.

Arithmetic Operators

Overview

Operator	Operation	Operand Types	Result Type
+	addition	byte, short, integer, word, longint	byte, short, integer, word, longint
-	subtraction	byte, short, integer, word, longint	byte, short, integer, word, longint
*	multiplication	byte, short, integer, word	integer, word, longint
div	division	byte, short, integer, word	byte, short, integer, word
mod	remainder	byte, short, integer, word	byte, short, integer, word

Mod and Div

`A div B` is the value of A divided by B rounded down to the nearest integer. The **mod** operator returns the remainder obtained by dividing its operands. In other words,

```
X mod Y = X - (X div Y) * Y.
```

If 0 (zero) is used explicitly as the second operand (i.e. X div 0), compiler will report an error and will not generate code. But in case of implicit division by zero: `X div Y`, where Y is 0 (zero), result will be the maximum value for the appropriate type (for example, if X and Y are words, the result will be $ffff).

Important

Destination will store the correct value only if it can properly represent the result of the expression (that is, if result fits in the destination range). More details can be found in chapter Assignment and Implicit Conversion.

Arithmetics and Data Types

mikroBasic is more flexible compared to standard Basic as it allows both implicit and explicit type conversion. In mikroBasic, operator can take operands of different type; refer to chapter Type Conversion for more details. You cannot combine signed and unsigned data types in expressions with arithmetic operators.

Unsigned and Conversion

If number is converted from less complex to more complex data type, upper bytes are filled with zeros. If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

Signed and Conversion

If number is converted from less complex to more complex data type, upper bytes are filled with ones if sign bit equals 1 (number is negative). Upper bytes are filled with zeros if sign bit equals 0 (number is positive). If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

Example

```
program extr

dim Sa as short
dim A as byte
dim Bb as word
dim Sbb as integer
dim Scccc as longint

A = A + Sa
  ' compiler will report an error,
  ' mixing signed with unsigned;

A = A - Sa
  ' compiler will report an error,
  ' mixing signed with unsigned;

  ' But you can freely combine byte with word . .

Bb = Bb + (A * A)

  ' . . and short with integer and longint

Scccc = Sbb * Sa + Scccc

end.
```

Unary arithmetic operators

Operator	Operation	Operand Types	Result Types
+(unary)	sign identity	short, integer, longint	short, integer, longint
- (unary)	sign negation	short, integer, longint	short, integer, longint

Unary arithmetic operators can be used to change sign of variables:

```
a = 3
b = -a
 ' assign value -3 to b
```

Runtime Behavior

PIC microcontrollers are optimized to work with bytes. Refer to PIC MCU Specific.

Boolean Operators

Boolean operators are not true operators, because there is no boolean data type defined in mikroBasic.

These "operators" conform to standard Boolean logic. They cannot be used with any data type, but only to build complex conditional expression.

Operator	Operation
not	negation
and	conjunction
or	disjunction

Example

```
if (astr > 10) and (astr < 20) then
PORTB = 0xff
end if
```

Logical (Bitwise) Operators

Overview

Operator	Operation	Operand Types	Result Types
not	bitwise negation	byte, word, short, integer, long	byte, word, short, integer, long
and	bitwise conjunction	byte, word, short, integer, long	byte, word, short, integer, long
or	bitwise disjunction	byte, word, short, integer, long	byte, word, short, integer, long
xor	bitwise xor	byte, word, short, integer, long	byte, word, short, integer, long
<<	bit shift left	byte, word, short, integer, long	byte, word, short, integer, long
>>	bit shift right	byte, word, short, integer, long	byte, word, short, integer, long

<< and >>

<< : shift left the operand for a number of bit places specified in the right operand (must be positive and less then 255).

>> : shift right the operand for a number of bit places specified in the right operand (must be positive and less then 255).

For example, if you need to extract the higher byte, you can do it like this:

```
dim temp as word

main:
  TRISA = word(temp >> 8)
end.
```

Important

Destination will hold the correct value if it can properly represent the result of the expression (that is, if result fits in destination range). More details can be found in chapters Type Conversions and Assignment and implicit Conversion.

Logical Operators and Data Types

mikroBasic is more flexible compared to standard Basic as it allows both implicit and explicit type conversion. Note that you cannot mix signed and unsigned data types in expressions with logical operators.

Unsigned and Conversion

If number is converted from less complex to more complex data type, upper byte is filled with zeros;

If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

Example for unsigned and logical operators :

```
dim  teA as byte
dim  Bb as word

main:
  Bb = $F0F0
  teA = $aa
  Bb = Bb and teA    ' Bb becomes $00a0
end.
```

In this case, teA is treated as a word with upper byte equal to 0 prior to the operation.

Signed and Conversion

If number is converted from less complex data type to more complex, upper bytes are filled with ones if sign bit is 1 (number is negative); upper bytes are filled with zeros if sign bit is 0 (number is positive).

If number is converted from more complex data type to less complex, data is simply truncated (upper bytes are lost).

```
dim  Sa as short
dim  Sbb as integer

main:
  Sbb = $70FF
  Sa = -12
  Sbb = Sbb and Sa    ' Sbb becomes $70f4
end.
```

In this case, Sa is treated as an integer with the upper byte equal to $FF (this in fact is sign extending of short to integer) prior to the operation.

```
main:
  Sbb = $OFF0
  Saa = $0a
  Sbb = Sbb and Sa    ' Sbb becomes $0000
end.
```

In this case, Sa is treated as an integer with the upper byte equal to $00 (this in fact is sign extending of short to integer) prior to the operation.

Example

```
dim teA as byte
dim teB as byte
dim teC as byte

 ' The logical operators perform bitwise manipulation
 '   on the operands. For example, if the value stored in
 '   teA (in binary) is 00001111 and the value stored in
 '   teB is 10000001, the following statements..

main:
teA = $0F   ' .. assign the value 00001111 to teA.
teB = $81   ' .. assign the value 10000001 to teB.

teC = teA or teB
 ' Performs bitwise or with teA, teB and the
 '   result is assigned to teC (value 10001111)

teC = not teA
 ' Performs bitwise not with teA and the
 '   result is assigned to teC (value 11110000)

teC = teA  <<  4
 ' shift teA to the left for a number of positions
 '   specified in the operand on the right;
 '   operand on the right must be positive.
 '   In this example teC becomes $F0
 ' All bits shifted in are zeros.

teC = teA  >>  4
 ' shift teA to the right for a number of positions
 '  specified in operand on the right;
 '  operand on the right must be positive.
 '  In this example C becomes $00.

 ' New bits shifted in are zeros if operand type is
 ' byte/word sign extended for short, word, integer.
end.
```

```
 ' You cannot mix signed and unsigned data types in
 '  expressions with logical operators:

dim   Sa  as short
dim   teA as byte
dim   Bb  as word
dim   Sbb as integer
dim   Scccc as longint

main:
teA = teA + Sa      ' compiler will report an error
teA = teA and Sa    ' compiler will report an error

 ' But you can freely mix byte with word . .
Bb = Bb and ( not teA)

 ' . . and short with integer and longint.
Scccc = Sbb xor Sa or Scccc
end.
```

Runtime Behavior

PIC microcontrollers are optimized to work with bytes. Refer to PIC MCU Specific.

Relation (Comparison) Operators

Overview

Operator	Operation	Operand Types	Result Types
=	equality	All simple types	True or False
<>	inequality	All simple types	True or False
<	less-than	All simple types	True or False
>	greater-than	All simple types	True or False
<=	less-than-or-equal	All simple types	True or False
>=	greater-than-or-equal	All simple types	True or False

Relation operators (Comparison Operators) are commonly used in conditional and loop statements to control the program flow.

In general case:

Expression1 (relation operator) Expression2,

expression1 and expression2 can be any legal expression. Be familiar with how implicit conversion works with relations operators. You can compare signed and unsigned values.

Rules for Comparing Expressions

1. Complex expression is decomposed to a number of simple expressions, with respect to operator precedence and overriding parenthesis.

2. Simple expression is now treated in the following manner: if operands are of the same type, operation is performed, assuming that the result is of the same type.

3. If operands are not of the same type, then less complex operand (speaking in terms of data range) is extended:

If one operand is byte and another is word, byte is converted in word.

If one operand is short and another is integer, short is converted to integer.

If one operand is short and another is longint, short is converted to longint.

If one operand is integer and another is longint, integer is converted to longint.

4. After the first expression is decomposed to simpler ones, each of these is evaluated abiding the rules presented here.

Illustration

Expression a + b + c is decomposed like this:

First evaluate a + b and get (value of a + b)

This gives us another simple expression
(value of a + b) + c

Let's assume a and b are bytes and c is word,
with values:

```
a = 23
b = 34
c = 1000
```

Compiler first calculates value of a + b and assumes
that the result is byte: a + b gives 57.

As c is of word type, result of adding a + b is
casted to word and then added to c: 57 + c is 1057.

Signed and unsigned numbers cannot be combined using arithmetical and logical operators. Rules presented above are not valid when assigning expression result to variable.

```
r = expression
```

Refer to Assignment and Implicit Conversion for more details.

Examples

Comparing variables and constants always produces the correct results regardless of the operands' type.

```
if A > B then ...
if A > 47 then ...

if A + B > A ...
```

First, compiler evaluates the expression on the left. During the run-time, result is stored in a variable of type that matches the largest data type in the expression. In this case it is byte, as variables A and B are both bytes.

This is correct if the value does not exceed range 0..255, that is, if A + B is less then 255.

Let's assume Aa is of word type :

```
if Aa + B > A ...
```

First, compiler evaluates the expression on the left. The result value is treated as type that matches the largest data type in the expression. Since Aa is word and B is byte, our result will be treated as word type.

This is correct if the value does not exceed range 0..65535, that is, if A + B is less then 65535.

```
 ' if tC is less than zero, tC = -tC :

if  tC < 0  then
  tC = -tC
end if

 ' Stay in loop while C is not equal to variable
 ' compare_match; increment C in every cycle:

while tC <> compare_match
    tC = tC + 1
wend
```

CONDITIONAL STATEMENTS

Conditional statements control which part(s) of the program will be executed, depending on a specified criteria. There are two conditional statements in mikroBasic:

SELECT CASE statement,
IF statement.

We suggest browsing the chapters Relation Operators and Implicit Conversion and Relation Operators, if you have not done so already.

Labels and Goto

Labels represent a more clear-cut way of controlling the program flow. You can declare a label below variables declarations, but you cannot declare two labels under the same name within the same routine. Name of the label needs to be a valid identifier. Multiple label declarations in single line are not allowed.

Goto statement jumps to the specified label unconditionally, and the program execution continues normally from that point on.

Here is an example:

```
program test

dim jjj as byte

main:
  ' some instructions ...
goto myLabel
  ' some instructions...
myLabel:
  ' some instructions...
end.
```

Select Case Statement

Select Case statement is used for selecting one of several available branches in the program course. It consists of a selector variable as a switch condition, and a list of possible values. These values can be constants, numerals, or expressions.

Eventually, there can be an else statement which is executed if none of the labels corresponds to the value of the selector.

Proper declaration of case statement is:

```
select case Selector
   case Values_1
      Statements_1
   case Values_2
      Statements_2
   ...
   case Values_N
      Statements_n
end select
```

where *selector* is any variable of simple type or expression, and each *Values* is a comma-delimited sequence of expressions.

Case statement can have a final else clause:

```
select case Selector
   case Values_1
      Statements_1
   case Values_2
      Statements_2
   ...
   case Values_N
      Statements_n
   case else
      Statements_else
end select
```

As soon as the case statement is executed, at most one of the *statements statements_1* .. *statements_n* will be executed. The *Values* which matches the selector determines the statements to be executed.

If none of the *Value* items matches the selector, then the *statements_else* in the else clause (if there is one) are executed.

Examples

```
select case message_flag
    case 0
       opmode = 0
       LCD_Out(1, 1, "Test Message 0")
    case 1, 2, 3, 4
       opmode = 1
       LCD_Out(1, 1, "Test Message 1")
    case 5, 6, 7
       opmode = 2
       LCD_Out(1, 1, "Test Message 2")
end select
```

In case there are multiple matches, the first matching block will be executed.

For example, if *state2* equals -1, *msg* will be OK, not Error:

```
select case state
    case 0, state0, state1, state2
       msg = "OK"
    case -1, state0 or errorFlag, state1 or errorFlag
       msg = "Error"
    case else
       msg = "No input"
end select
```

If Statement

There are two forms of if statement:

Syntax of `if..then` statement is:

```
if expression then
  statements
end if
```

where *expression* returns a True or False value. If *expression* is True, then *statement* is executed, otherwise it's not.

Syntax of `if..then..else` statement is:

```
if expression then
  statements1
else
  statements2
end if
```

where *expression* returns a True or False value. If *expression* is True, then *statements1* are executed; otherwise *statements2* are executed. *Statements1* and *statements2* can be statements of any type.

Nested IF

Nested if statements require additional attention. General rule is that the nested conditionals are parsed starting from the innermost conditional, with each else bound to the nearest available if on its left.

```
if expression1 then
   if expression2  then
        statements1
   else
        statements2
   end if
end if
```

Compiler treats the construction like this:

```
if expression1 then
       [ if expression2 then
           statement1
       else
         statement2
        end if ]
end if
```

To force the compiler to interpret our example the other way around, we would have to write it explicitly:

```
if expression1 then
   if expression2 then
        statement1
   end if
else
  statement2
end if
```

Examples

```
if J <> 0 then
   Res = I div J
end if

if j <> 0 then
  i = i + 1
  j = 0
end if

 ...

if v = 0 then
    portb = por2
    porta = 1
    v = 1
else
   portb = por1
   porta = 2
   v = 0
end if
```

LOOPS

Loops are a specific way to control the program flow. By using loops, you can execute a sequence of statements repeatedly, with a control condition or variable to determine when the execution stops.

You can use the standard `break` and `continue` to control the flow of a `do..loop until`, `while`, or `for` statement. `Break` terminates the statement in which it occurs, while `continue` begins executing the next iteration of the sequence.

mikroBasic has three kinds of control loop instructions:

DO..LOOP UNTIL statement
WHILE statement
FOR statement

Runtime Behavior

Note that certain operations may take longer time to be executed, which can lead to undesired consequences.

If you add two variables of short type and assign the result to short, it will be faster than to add two longint and assign value to longint, naturally.

Take a look at the following code :

```
dim Sa as short
dim Sb as short
dim Saaaa as longint
dim Sbbbb as longint

for Sa = 0 to 100
      Sb = Sb + 2
next Sa

for Saaaa = 0 to 100
      Sbbbb = Sbbbb + 2
next Saaaa
end.
```

PIC will execute the first loop *considerably* faster.

For Statement

`For` statement requires you to specify the number of iterations you want the loop to go through. Syntax of `for` statement is:

```
for counter = initialValue to finalValue [step step_value]
  statement_1
  statement_2
  ...
  statement_N
next counter
```

where *counter* is variable; *initialValue* and *finalValue* are expressions compatible with *counter*; *statement_X* is any statement that does not change the value of *counter*; *step_value* is value that is added to the *counter* in each iteration. *Step_value* is optional, and defaults to 1 if not stated otherwise. Be careful when using large values for step_value, as overflow may occur.

Every statement between `for` and `next` will be executed once for each iteration.

Endless Loop

Be careful not to create endless loop by mistake. The following statement:

```
for counter = initialValue to finalValue
  statement
next counter
```

will result in an an endless loop if *finalValue* is greater than, or equal to maximum value of *counter* data type. For example, this will be an endless loop, if *counter* is of byte type:

```
for counter = 0 to 255
  nop
next counter

 ' or

for counter = 0 to 500
  nop
next counter
```

Example

Here is a simple example of a `for` loop used for emitting hex code on PORTB. Nine digits will be printed with one second delay, by incrementing the counter.

```
for i = 1 to 9
    portb = i
    delay_ms(1000)
next i
```

Do..Loop Until Statement

Syntax of `do..loop` statement is:

```
do
  statement_1
  ...
  statement_N
loop until expression
```

where *expression* returns a True or False value. `Do..loop` statement executes *statement_1 ... statement_N* continually, checking the *expression* after each iteration. Eventually, when *expression* returns True, `do..loop` statement terminates.

The sequence is executed at least once because the check takes place in the end.

Example

```
i = 0
do
    i = i + 1        ' execute these 2 statements
    PORTB = i        '     until i equals 10 (ten)
loop until i = 10
```

While Statement

Syntax of `while` statement is:

```
while expression
  statement_0
  statement_1
  ...
  statement_N
wend
```

Expression is tested first. If it returns True, all the following statements enclosed by `while` and `wend` will be executed. It will keep on executing statements until the *expression* returns False.

Eventually, as *expression* returns False, `while` will be terminated without executing statements.

`While` is similar to `do..loop until`, except the check is performed at the beginning of the loop. If *expression* returns False upon first test, *statements* will not be executed.

Example

```
while i < 90
  i = i + 1
wend

 ...

while i > 0
  i = i div 3
  PORTA = i
wend
```

ASM Statement

Sometimes it can be useful to write part of the program in assembly. ASM statement allows you to embed PIC assembly instructions into Basic code.

Note that you cannot use numerals as absolute addresses for SFR or GPR variables in assembly instructions. You may use symbolic names instead (listing will display these names as well as addresses). Also, you cannot use Labels in assembly; instead, you can use relative jumps such as goto $-1.

Declaration of asm statement is:

```
asm
  statementList
end asm
```

where *statementList* is a sequence of assembly instructions.

Be careful when embedding assembly code - mikroBasic will not check if assembly instruction changed memory locations already used by Basic variables.

Also, you cannot write comments in assembly.

Example

```
asm
  movlw 67
  movwf TMR0
end asm


asm        ' second instruction is incorrect, see above
  MOVLW 0
  MOVWF $5
  MOVWF PORTA
end asm   ' note that you cannot write comments in assembly
```

PIC MCU SPECIFIC

In order to get the most from your mikroBasic compiler, you should be familiar with certain aspects of PIC MCU. This chapter is not essential, but it can provide you a better understanding of PICs' capabilities and limitations, and their impact on the code writing.

For start, you should know that arithmetical operations such as addition and subtraction are carried out by ALU (Arithmetical Logical Unit). With PIC MCUs (series PIC16 and PIC18), ALU is optimized for working with bytes. mikroBasic is capable of handling much more complex data types, but note that these can increase the time needed for performing even simple operations.

Also, not all PIC MCU models are of equal performance. PIC16 series lacks hardware resources to multiply two bytes in HW - it is carried out by software algorithm generated by mikroBasic. On the other hand, PIC18 series has HW multiplier, and as a result, multiplication works considerably faster.

Loops are convincing examples of byte type efficiency, especially if statements repeated hundreds of times are involved. Consider the following lines:

```
for i = 1 to 100
    tA = ta + 1
next i

  ...

for ii = 1 to 100
    Aa = Aa + 1
next ii
```

where *i* and *A* are variables of byte type, and *ii* and *Aa* are variables of word type. First loop will be executed *considerably* faster.

Although memory management is completely under the compiler's control, you can explicitly assign address to variable by means of directive absolute. See Directives for more information.

NOTE : Be aware that nested function and procedure calls have limited depth - 8 for PIC16 series and 31 for PIC18 series.

mikroBASIC SPECIFIC

mikroBasic compiler was designed with reliability and comfort in mind. Thus, certain modifications of standard Basic were necessary in order to make the compiler more PIC MCU compatible.

PIC SFR (Special Function Registers) are implicitly declared as global variables of byte type. Their scope is the entire project - they are visible in any part of the program or any unit. Memory management is completely under compiler's control, so there is no need to worry about PIC memory banks and storing the variables.

Accessing to individual bits of SFR (as accessing to bit of any variable of byte type) is very simple. Use identifier followed by dot, and a pin:

```
Identifier.PIN      ' PIN is a constant value between 0..7
```

For example:

```
sub procedure check
  ifPORTB.1 = 1 then
   counter = counter + 1
  else
   INTCON.GIE = 0
  end if
end sub
```

Interrupts can be easily handled in mikroBasic by means of predefined procedure interrupt. All you need to do is include the complete procedure definition in your program. mikroBasic saves the following SFR when entering interrupt: PIC12 and PIC16 series: W, STATUS, FSR, PCLATH; PIC18 series: FSR (fast context is used to save WREG, STATUS, BSR). Upon return from interrupt routine, these registers are restored.

NOTE: For PIC18 family, interrupts must be of high priority. mikroBasic does not support low priority interrupts.

For example, when handling the interrupts from TMR0
(if no other interrupts are allowed):

```
sub procedure interrupt
  counter = counter + 1
  TMR0 = 96
  INTCON = $20
end sub
```

In case of multiple interrupts enabled, you must test which of the interrupts occurred and then proceed with the appropriate code (interrupt handling):

```
sub procedure interrupt

 if INTCON.TMR0IF = 1 then
    counter = counter + 1
    TMR0 = 96
    INTCON.TMR0IF = 0
 else
    if INTCON.RBIF = 1 then
        counter = counter + 1
        TMR0 = 96
        INTCON.RBIF = 0
    end if
  end if
end sub
```

See also:

Built-in Functions and Procedures
Library Functions and Procedures

COMPILER ERROR MESSAGES

Error Messages

Type of Error	Error No.
_SYNTAX_ERROR	100
_NOT_VALID_IDENT	101
_INVALID_STATEMENT	102
_STACK_OVERFLOW	103
_INVALID_OPERATOR	104
_IF_ELSE_ERROR	105
_VARIABLE_EXPECTED	106
_CONSTANT_EXPECTED	107
_ASSIGNMENT_EXPECTED	108
_BREAK_ERROR	109
_UNKNOWN_TYPE	110
_VARIABLE_REDECLARED	111
_VARIABLE_NOT_DECLARED	112
_MAX_LINE_NUMBER_EXCEEDED	113
_ALREADY_DECLARED *// for proc and func*	114
_TOO_MANY_PARAMS	115
_NOT_ENOUGH_PARAMS	116
_TYPE_MISMATCH	117
_FILE_NOT_FOUND	118
_NOT_ENOUGH_RAM	119
_USES_IN_BETA_V	120
_INTERNAL_ERROR	121
_NOT_ENOUGH_ROM	122
_INVALID_ARRAY_TYPE	123
_BAUD_TOO_HIGH	124
_DIVISION_BY_ZERO	125
_INCOMPATIBLE_TYPES	126
_TOO_MANY_CHARACTERS	127
_OUT_OF_RANGE	128
_USES_POSITION	129
_INVALID_ASM_COMMAND	130
_OPERATOR_NOT_APPLICABLE	131
_EXPRESSION_BY_ADDRESS	132
_IDENTIFIER_EXPECTED	133
_MOVING_ARRAYS	134

Warning Messages

Type of Error	Error No.
_CODE_AFTER_END	200
_BAUD_ERROR	201
_UPPER_BYTES_IGNORED	202
_UPPER_WORDS_IGNORED	203
_IMPLICIT_TYPECAST	204

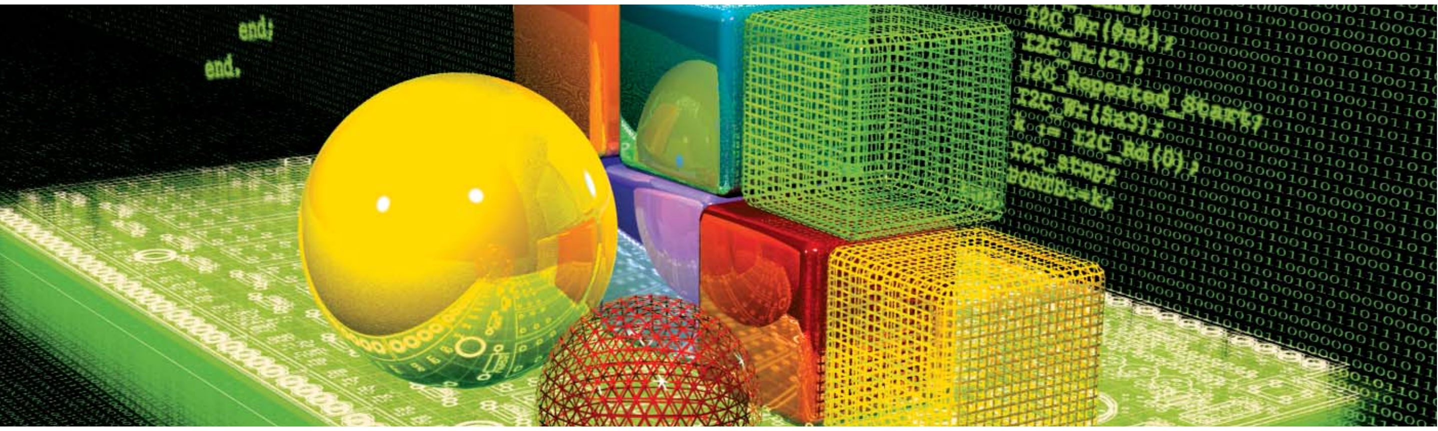
Hint Messages

Type of Error	Error No.
_VAR_NOT_USED	300
_PROC_NOT_CALLED	301

Linker Error Messages

Type of Error	Error No.
_UNKNOWN_ASM	400
_ADDRESS_CALC_ERROR	401

CHAPTER 3



Built-in and Library Routines

mikroBasic provides a number of built-in and library routines which help you develop your application faster and easier. Libraries for ADC, CAN, USART, SPI, I2C, 1-Wire, LCD, PWM, RS485, numeric formatting, bit manipulation, and many other are included along with practical, ready-to-use code examples.

BUILT-IN ROUTINES

mikroBasic compiler incorporates a set of built-in functions and procedures. They are provided to make writing programs faster and easier. You can call built-in functions and procedures in any part of the program.

Routines

```
sub procedure SetBit(dim byref REG as byte, dim BIT as byte)
sub procedure ClearBit(dim byref REG as byte, dim BIT as byte)
sub function  TestBit(dim byref REG as byte, dim BIT as byte) as byte

sub function  Lo(dim arg as byte..longint) as byte
sub function  Hi(dim arg as word..longint) as byte
sub function  Higher(dim arg as longint)   as byte
sub function  Highest(dim arg as longint)  as byte

sub procedure Inc(byref arg as byte..longint)
sub procedure Dec(byref arg as byte..longint)

sub procedure Delay_us(const COUNT as word)
sub procedure Delay_ms(const COUNT as word)
sub procedure Delay_Cyc(dim Cycles_div_by_10 as byte)

sub function  Length(dim text as string) as byte
```

Routines SetBit, ClearBit and TestBit are used for bit manipulation. Any SFR (Special Function Register) or variable of byte type can pass as valid variable parameter, but constants should be in range [0..7].

Routines Lo, Hi, Higher and Highest extract one byte from the specified parameter. Check the examples for details.

Routines Inc and Dec increment and decrement their argument respectively

Routines Delay_us and Delay_ms create a software delay in duration of COUNT microseconds or milliseconds, respectively.

Routine Delay_Cyc creates a delay based on MCU clock. Delay lasts for (10 times the input parameter) in MCU cycles. Input parameter needs to be in range 3 .. 255.

Function Length returns string length.

Examples

```
SetBit(PORTB,2)
 ' set PORTB bit RB2 to value 1

ClearBit(PORTC,7)
 ' clear PORTC bit RC7

TestBit(PORTA,2)
 ' returns 1 if PORTA bit RA2 is 1, and 0 if RA2 is 0


Lo(A)
 ' returns lower byte of variable A
 ' byte 0, assuming that word/integer comprises bytes 1 and 0,
 ' and longint comprises bytes 3, 2, 1, and 0

Hi(Aa)
 ' returns higher byte of variable Aa
 ' byte 1, assuming that word/integer comprises bytes 1 and 0,
 ' and longint comprises bytes 3, 2, 1, and 0

Higher(Aaaa)
 ' returns byte next to the highest byte of variable Aaaa
 ' byte 2, assuming that longint comprises bytes 3, 2, 1, 0

Highest(Aaaa)
 ' returns the highest byte of variable Aaaa
 ' byte 3, assuming that longint comprises bytes 3, 2, 1, 0


Inc(Aaaa)
 ' increments variable Aaaa by 1

Dec(Aaaa)
 ' decrements variable Aaaa by 1


Delay_us(100)
 ' creates software delay equal to 100 microseconds.

Delay_ms(1000)
 ' creates software delay equal to 1000 milliseconds = 1s.

Delay_Cyc(100)
 ' creates delay equal to 1000 MCU cycles.

Length(Text)
 ' returns string length as byte
```

LIBRARY ROUTINES

Library procedures and functions represent a set of routines. This collection of functions and procedures is provided for simplifying the initialization and use of PIC MCU and its hardware modules (ADC, I2C, USART, SPI, PWM), driver for LCD, drivers for internal and external CAN modules, flexible 485 protocol, numeric formatting routines...

Currently included libraries:

1wire
ADC
CAN
CANSPI
Compact Flash
Flash Memory
EEPROM
I2C
LCD (4-bit interface)
LCD8 (8-bit interface)
Graphic LCD
PWM
RS485
SPI
USART

Software I2C
Software SPI
Software UART

Sound
Manchester Code
Numeric Formatting Routines
Utilities

1-Wire Library

1-wire library provides routines for communicating via 1-wire bus, for example with DS1820 digital thermometer.

Note that oscillator frequency Fosc needs to be at least 4MHz in order to use the routines with Dallas digital thermometers.

Routines

```
function  OW_Reset(dim byref PORT as byte, dim PIN as byte) as byte
function  OW_Read(dim byref PORT as byte, dim PIN as byte) as byte
procedure OW_Write(dim byref PORT as byte, dim PIN, par as byte)
```

```
function  OW_Reset(dim byref PORT as byte, dim PIN as byte) as byte
```

Issues 1-wire reset signal for DS1820.
Parameters PORT and pin specify the location of DS1820; return value of the function is 0 if DS1820 is present, and 1 if it is not present.

```
function  OW_Read(dim byref PORT as byte, dim PIN as byte) as byte
```

Reads one byte via 1-wire bus.

```
procedure OW_Write(dim byref PORT as byte, dim PIN, par as byte)
```

Writes one byte (parameter par) via 1-wire bus.

Example

The following code demonstrates use of 1-wire library procedures and functions. The example reads the temperature using DS1820 connected to PORTA, pin 5. Be sure to set the Fosc appropriately in your project.

```
program onewire_test

dim i    as byte
dim j1   as byte
dim j2   as byte
dim por1 as byte
dim por2 as byte
dim text as char[20]

main:
   text    = "Temperature:"
   PORTB   = 0                         ' initialize PORTB to 0
   PORTA   = 255                       ' initialize PORTA to 255
   TRISB   = 0                         ' PORTB is output
   TRISA   = 255                       ' PORTA is input
   LCD_Init(PORTB)
   LCD_Cmd(LCD_CURSOR_OFF)
   LCD_Out(1, 1, text)

   do
       OW_Reset(PORTA,5)               ' 1-wire reset signal
       OW_Write(PORTA,5,$CC)           ' issue command to DS1820
       OW_Write(PORTA,5,$44)           ' issue command to DS1820
       Delay_ms(120)
       i = OW_Reset(PORTA,5)
       OW_Write(PORTA,5,$CC)           ' issue command to DS1820
       OW_Write(PORTA,5,$BE)           ' issue command to DS1820
       Delay_ms(1000)
       j1 = OW_Read(PORTA,5)           ' get result
       j2 = OW_Read(PORTA,5)           ' get result
       j1 = j1 >> 1                    ' assuming the temp. >= 0C
       ByteToStr(j1, text)             ' convert j1 to text
       LCD_Out(2, 8, text)             ' print text
       LCD_Chr(2, 10, 223)             ' degree character (°)
       LCD_Chr(2, 11,"C")
       Delay_ms(500)

   loop until false                    ' endless loop
end.
```

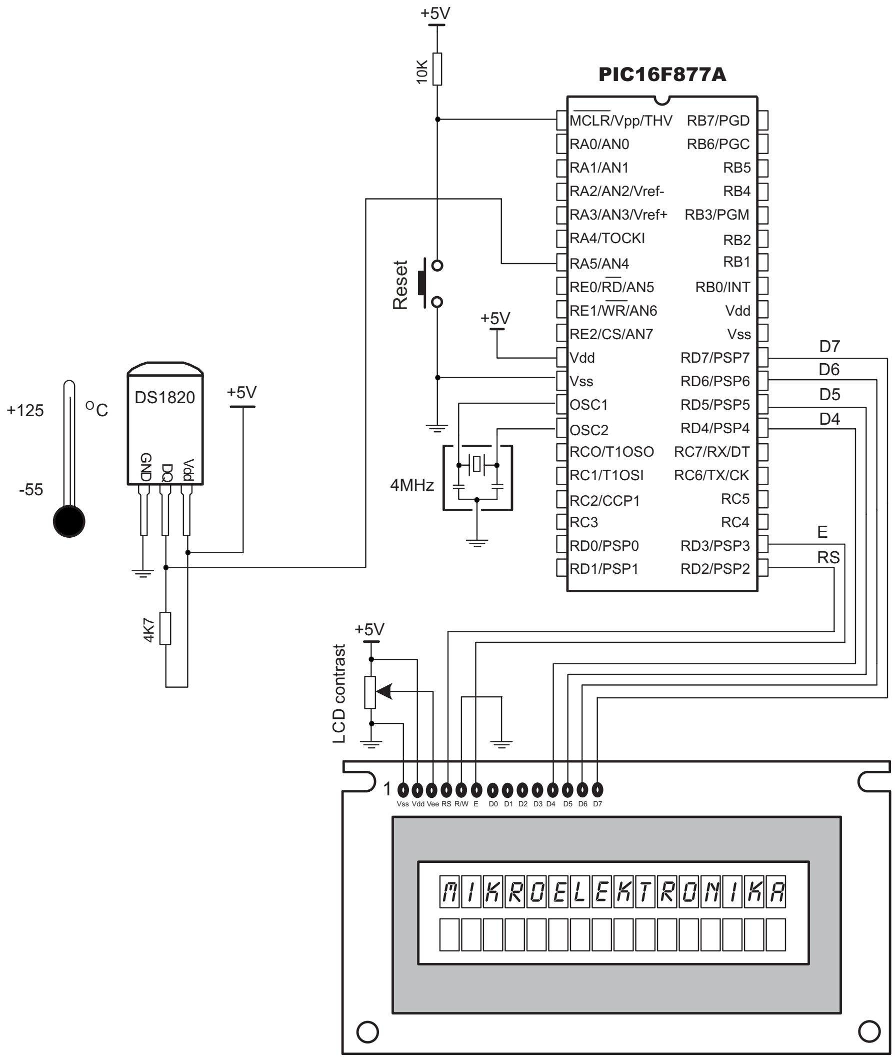


Figure (example of DS1820 on PORTA, pin 5)

ADC Library

ADC (Analog to Digital Converter) module is available with a number of PIC MCU models. Library function `ADC_read` is included to provide you comfortable work with the module.

The function is currently unsupported by the following PIC MCU models: P18F2331, P18F2431, P18F4331, and P18F4431.

Routines

You can use the library function to initialize internal AD converter, select channel, and get the result of conversion:

```
sub function ADC_Read(dim Channel as byte) as word
```

It initializes ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12 Tad). RC sources typically have Tad 4uS (A/D conversion time per bit).

Parameter *Channel* determines which channel will be sampled. Refer to the device data sheet for information on device channels.

Important

Before using the function above, be sure to configure the appropriate TRISA bits to designate the pins as input. Also, configure the desired pin as analog input, and set Vref (voltage reference value).

Example

The following code demonstrates use of library function ADC_read. Example reads Channel 2 and stores value in variable temp_res.

```
program ADC_Test
dim temp_res as word

main:
  ADCON1  = $80              ' configure analog inputs and Vref
  TRISA   = $FF              ' PORTA is input
  TRISB   = $3F              ' pins RB7, RB6 are output
  TRISD   = $0               ' PORTD is output
  while true
    temp_res = ADC_read(2)
                             ' now you can use temp_res ...
    PORTD = temp_res         ' send lower 8 bits to PORTD
    PORTB = word(temp_res >> 2)
        ' send two most significant bits to PORTB
  wend
end.
```

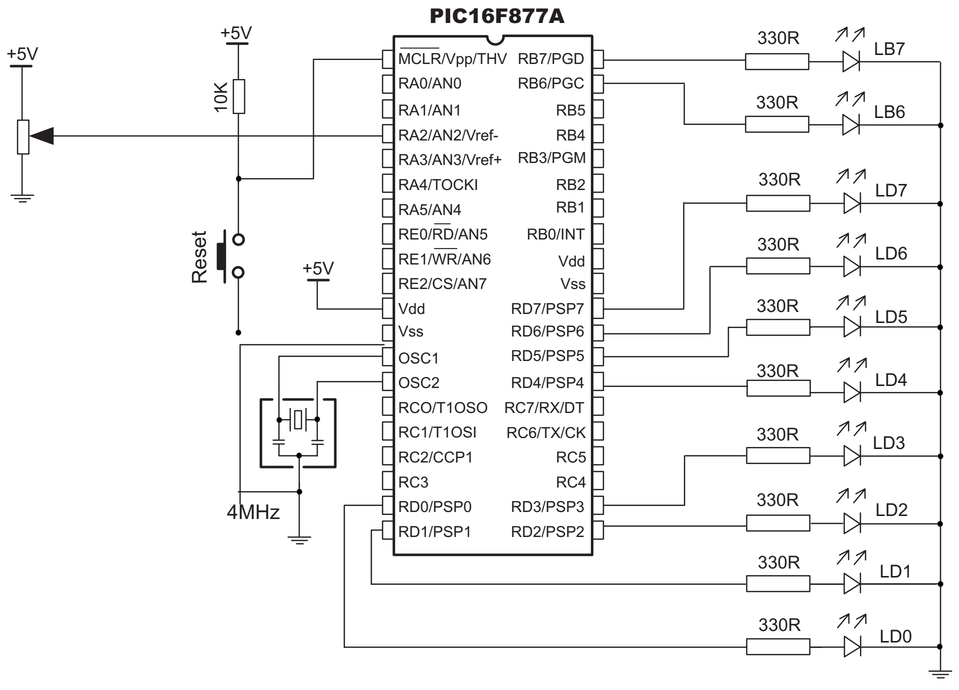


Figure (ADC HW connection)

CAN Library

CAN (Controller Area Network) module is available with a number of PIC MCU models. mikroBasic includes a set of library routines to provide you comfortable work with the module.

CAN routines are currently supported by PIC MCU models P18XXX8. Microcontroller must be connected to CAN tranceiver (MCP2551 or similar) which is connected to CAN bus.

The Controller Area Network module is a serial interface, useful for communicating with other peripherals or microcontrollers. Details about CAN can be found in appropriate literature and on mikroElektronika Web site.

Following routines can be considered a driver for CAN module on PIC MCUs.

```
sub procedure CANSetOperationMode(dim mode as byte, dim WAIT as byte)

sub function  CANGetOperationMode as byte

sub procedure CANInitialize(dim SJW as byte, dim BRP as byte, dim PHSEG1 as byte,
                            dim PHSEG2 as byte, dim PROPSEG as byte,dim CAN_CONFIG_FLAGS as byte)

sub procedure CANSetBaudRate(dim SJW as byte, dim BRP as byte, dim PHSEG1 as byte,
                            dim PHSEG2 as byte, dim PROPSEG as byte,dim CAN_CONFIG_FLAGS as byte)

sub procedure CANSetMask(dim CAN_MASK as byte, dim val as longint, dim CAN_CONFIG_FLAGS as byte)

sub procedure CANSetFilter(dim CAN_FILTER as byte, dim val as longint,
                        dim CAN_CONFIG_FLAGS as byte)

sub function  RegsToCANID(dim byref ptr as byte, dim CAN_CONFIG_FLAGS as byte) as longint

sub procedure CANIDToRegs(dim byref ptr as byte, dim val as longint, dim CAN_CONFIG_FLAGS as
byte)

sub function  CANwrite(dim id as longint, dim byref Data as byte[8], dim DataLen as byte,
                     dim CAN_TX_MSG_FLAGS as byte) as byte

sub function  CANread(dim byref id as longint, dim byref Data as byte[8],
                    dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte
```

CANSetOperationMode

Prototype: `sub procedure CANSetOperationMode(dim mode as byte, dim WAIT as byte)`

Parameters: mode - Operation mode code can take any of predefined constant values (see the constants below)
WAIT - Should have value TRUE(255) or FALSE(0)

Effects: CAN is set to requested mode

Overview: Given mode byte is copied to CANSTAT

Note: If WAIT is true, this is a blocking call. It won't return until requested mode is set.

If WAIT is false, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use CANGetOperationMode() to verify correct operation mode before performing mode specific operation.

CANGetOperationMode

Prototype: `sub function CANGetOperationMode as byte`

Parameters: None

Output: Current operational mode of CAN module is returned

CANInitialize

Prototype: `sub procedure CANInitialize(dim SJW as byte, dim BRP as byte, dim PHSEG1 as byte, dim PHSEG2 as byte, dim PROPSEG as byte,dim CAN_CONFIG_FLAGS as byte)`

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters: SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS value is formed from constants (see below)

Effects: CAN bit rate is set. All masks registers are set to '0' to allow all messages.

Filter registers are set according to flag value:

```
If (CAN_CONFIG_FLAGS and CAN_CONFIG_VALID_XTD_MSG) <> 0
    Set all filters to XTD_MSG
Else if (config and CONFIG_VALID_STD_MSG) <> 0
    Set all filters to STD_MSG
Else
    Set half of the filters to STD, and the rest to XTD_MSG.
```

Side Effects: All pending transmissions are aborted.

CANSetBaudRate

Prototype:
```
sub procedure CANSetBaudRate(dim SJW as byte, dim BRP as byte, dim PHSEG1 as
byte,dim PHSEG2 as byte,dim PROPSEG as byte,dim CAN_CONFIG_FLAGS as byte)
```

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters: SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS - Value formed from constants (see section below)

Effects: CAN bit rate is set as per given values.

Overview: Given values are bit adjusted to fit in 18XXX8. BRGCONx registers and copied.

CANSetMask

Prototype:
```
sub procedure CANSetMask(dim CAN_MASK as byte, dim val as longint, dim
CAN_CONFIG_FLAGS as byte)
```

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_MASK - One of predefined constant value
val - Actual mask register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer mask registers.

CANSetFilter

Prototype:

```
sub procedure CANSetFilter(dim CAN_FILTER as byte, dim val as longint,
dim CAN_CONFIG_FLAGS as byte)
```

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_FILTER - One of predefined constant values
val - Actual filter register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer filter registers.

RegsTOCANID and CANIDToRegs

Prototypes:

```
sub function  RegsToCANID(dim byref ptr as byte, dim CAN_CONFIG_FLAGS as
byte) as longint

sub procedure CANIDToRegs(dim byref ptr as byte, dim val as longint,
CAN_CONFIG_FLAGS as byte)
```

Effects: These two routines are used by other routines (internal purpose only)

CANWrite

Prototype:	`sub function CANwrite(dim id as longint, dim byref Data as byte[8], dim DataLen as byte, dim CAN_TX_MSG_FLAGS as byte) as byte`
Precondition:	CAN must be in Normal mode.
Parameters:	id - CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended). Data - array of bytes up to 8 bytes in length DataLen - Data length from 1 thru 8. CAN_TX_MSG_FLAGS - Value formed from constants (see section below)
Effects:	If at least one empty transmit buffer is found, given message is queued for the transmission. If none found, FALSE value is returned.

CANRead

Prototype:	`sub function  CANread(dim byref id as longint, dim byref Data as byte[8], dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte`
Precondition:	CAN must be in mode in which receiving is possible.
Parameters:	id - CAN message identifier Data - array of bytes up to 8 bytes in length DataLen - Data length from 1 thru 8. CAN_TX_MSG_FLAGS - Value formed from constants (see below)
Effects:	If at least one full receive buffer is found, it is extracted and returned. If none found, FALSE value is returned.

CAN Library Constants

You need to be familiar with constants that are provided for use with CAN library routines. See how to form values (from constants) that will be passed to or from routines in the example at the end of the chapter. All of the constants are predefined in CAN library.

CAN_OP_MODE

These constant values define CAN module operation mode. CANSetOperationMode() routine requires this code. These values must be used by itself, i.e. they cannot be ANDed to form multiple values.

```
const CAN_MODE_BITS   = $E0  ' Use these to access opmode bits
const CAN_MODE_NORMAL = 0
const CAN_MODE_SLEEP  = $20
const CAN_MODE_LOOP   = $40
const CAN_MODE_LISTEN = $60
const CAN_MODE_CONFIG = $80
```

CAN_TX_MSG_FLAGS

These constant values define flags related to transmission of a CAN message. There could be more than one this flag ANDed together to form multiple flags.

```
const CAN_TX_PRIORITY_BITS  = $03
const CAN_TX_PRIORITY_0  = $FC          ' XXXXXX00
const CAN_TX_PRIORITY_1  = $FD          ' XXXXXX01
const CAN_TX_PRIORITY_2  = $FE          ' XXXXXX10
const CAN_TX_PRIORITY_3  = $FF          ' XXXXXX11

const CAN_TX_FRAME_BIT = $08
const CAN_TX_STD_FRAME = $FF            ' XXXXX1XX
const CAN_TX_XTD_FRAME = $F7            ' XXXXX0XX

const CAN_TX_RTR_BIT = $40
const CAN_TX_NO_RTR_FRAME = $FF         ' X1XXXXXX
const CAN_TX_RTR_FRAME = $BF            ' X0XXXXXX
```

CAN_RX_MSG_FLAGS

These constant values define flags related to reception of a CAN message. There could be more than one this flag ANDed together to form multiple flags. If a particular bit is set; corresponding meaning is TRUE or else it will be FALSE.

e.g.

```
if (MsgFlag and CAN_RX_OVERFLOW) <> 0 then

  ' Receiver overflow has occurred.
  ' We have lost our previous message.
  ...

const CAN_RX_FILTER_BITS = $07   ' Use these to access filter bits
const CAN_RX_FILTER_1 = $00
const CAN_RX_FILTER_2 = $01
const CAN_RX_FILTER_3 = $02
const CAN_RX_FILTER_4 = $03
const CAN_RX_FILTER_5 = $04
const CAN_RX_FILTER_6 = $05
const CAN_RX_OVERFLOW = $08      ' Set if Overflowed else cleared
const CAN_RX_INVALID_MSG = $10   ' Set if invalid else cleared
const CAN_RX_XTD_FRAME = $20     ' Set if XTD message else cleared
const CAN_RX_RTR_FRAME = $40     ' Set if RTR message else cleared
const CAN_RX_DBL_BUFFERED = $80  ' Set if this message was
                                 '   hardware double-buffered
```

CAN_MASK

These constant values define mask codes. Routine CANSetMask() requires this code as one of its arguments. These enumerations must be used by itself i.e. it cannot be ANDed to form multiple values.

```
const CAN_MASK_B1 = 0
const CAN_MASK_B2 = 1
```

CAN_FILTER

These constant values define filter codes. Routine CANSetFilter() requires this code as one of its arguments. These enumerations must be used by itself i.e. it cannot be ANDed to form multiple values.

```
const CAN_FILTER_B1_F1 = 0
const CAN_FILTER_B1_F2 = 1
const CAN_FILTER_B2_F1 = 2
const CAN_FILTER_B2_F2 = 3
const CAN_FILTER_B2_F3 = 4
const CAN_FILTER_B2_F4 = 5
```

CAN_CONFIG_FLAGS

These constant values define flags related to configuring CAN module. Routines CANInitialize() and CANSetBaudRate() use these codes. One or more these values may be ANDed to form multiple flags.

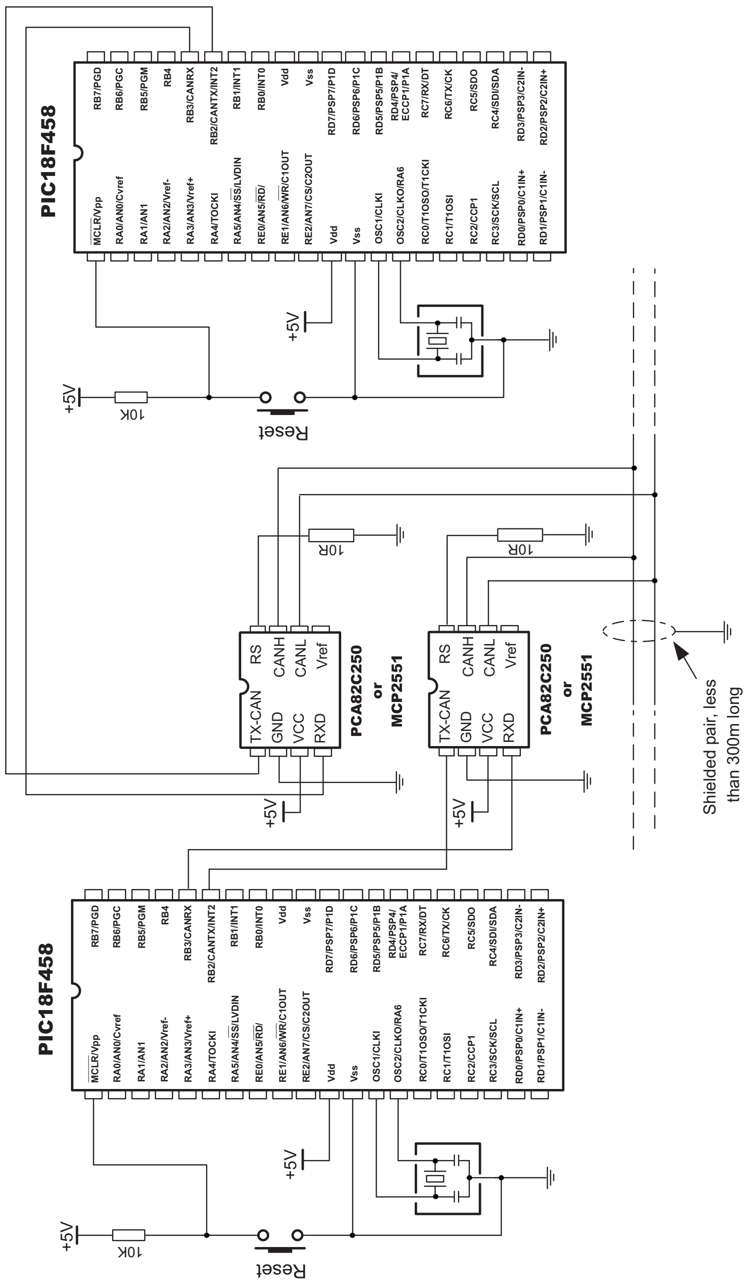
```
const CAN_CONFIG_DEFAULT = $FF          ' 11111111

const CAN_CONFIG_PHSEG2_PRG_BIT = $01
const CAN_CONFIG_PHSEG2_PRG_ON = $FF    ' XXXXXXX1
const CAN_CONFIG_PHSEG2_PRG_OFF = $FE   ' XXXXXXX0

const CAN_CONFIG_LINE_FILTER_BIT = $02
const CAN_CONFIG_LINE_FILTER_ON = $FF   ' XXXXXX1X
const CAN_CONFIG_LINE_FILTER_OFF = $FD  ' XXXXXX0X

const CAN_CONFIG_SAMPLE_BIT = $04
const CAN_CONFIG_SAMPLE_ONCE = $FF      ' XXXXX1XX
const CAN_CONFIG_SAMPLE_THRICE = $FB    ' XXXXX0XX

const CAN_CONFIG_MSG_TYPE_BIT = $08
const CAN_CONFIG_STD_MSG = $FF          ' XXXX1XXX
const CAN_CONFIG_XTD_MSG = $F7          ' XXXX0XXX

const CAN_CONFIG_DBL_BUFFER_BIT = $10
const CAN_CONFIG_DBL_BUFFER_ON = $FF    ' XXX1XXXX
const CAN_CONFIG_DBL_BUFFER_OFF = $EF   ' XXX0XXXX

const CAN_CONFIG_MSG_BITS = $60
const CAN_CONFIG_ALL_MSG = $FF          ' X11XXXXX
const CAN_CONFIG_VALID_XTD_MSG = $DF    ' X10XXXXX
const CAN_CONFIG_VALID_STD_MSG = $BF    ' X01XXXXX
const CAN_CONFIG_ALL_VALID_MSG = $9F    ' X00XXXXX
```

Example

This code demonstrates use of CAN library routines:

```
program CAN

dim aa  as byte
dim aa1 as byte
dim lenn as byte
dim aa2 as byte
dim data as byte[8]
dim id as longint
dim zr as byte
dim cont as byte
dim oldstate as byte

sub function TestTaster as byte
  result = true
  if Button(PORTB, 0, 1, 0) then
       oldstate = 255
  end if
  if oldstate and Button(PORTB, 0, 1, 1) then
      result = false
      oldstate = 0
  end if
end sub

main:
  TRISB.0 = 1            ' designate pin RB0 as input
  PORTC  = 0
  TRISC  = 0
  PORTD  = 0
  TRISD  = 0
  aa     = 0
  aa1    = 0
  aa2    = 0

  aa1 =  CAN_TX_PRIORITY_0 and             ' form value to be used
          CAN_TX_XTD_FRAME and             ' with CANSendMessage
          CAN_TX_NO_RTR_FRAME

  aa =   CAN_CONFIG_SAMPLE_THRICE and      ' form value to be used
          CAN_CONFIG_PHSEG2_PRG_ON and     ' with CANInitialize
          CAN_CONFIG_STD_MSG and
          CAN_CONFIG_DBL_BUFFER_ON and
          CAN_CONFIG_VALID_XTD_MSG and
          CAN_CONFIG_LINE_FILTER_OFF

  ' continues..
' ..continued

 cont = true               ' upon signal change on RB0 pin
 while cont                '      from logical 0 to 1
                           '      proceed with program
   cont = TestTaster       '      execution
 wend

 data[0] = 0
 CANInitialize( 1,1,3,3,1,aa)                ' initialize CAN
 CANSetOperationMode(CAN_MODE_CONFIG,TRUE)   ' set CONFIG mode
 ID = -1

 CANSetMask(CAN_MASK_B1,ID,CAN_CONFIG_XTD_MSG)
  ' set all mask1 bits to ones
 CANSetMask(CAN_MASK_B2,ID,CAN_CONFIG_XTD_MSG)
  ' set all mask2 bits to ones
 CANSetFilter(CAN_FILTER_B1_F1,3,CAN_CONFIG_XTD_MSG)
  ' set id of filter B1_F1 to 3
 CANSetOperationMode(CAN_MODE_NORMAL,TRUE)
  ' set NORMAL mode

 portd = $FF
 id = 12111
 CANWrite(id,data,1,aa1)          ' send message via CAN

 while true
   oldstate = 0
   zr = CANRead(id , Data , lenn, aa2)
   if (id = 3) and zr then
     portd = $AA
     portc = data[0]              ' output data at portC
     data[0] = data[0]+1
     id = 12111
     CANWrite(id,data,1,aa1)      ' send incremented data back
     if lenn = 2 then             ' if msg contains two data bytes
       portd = data[1]            '   output second byte at portd
     end if
   end if
 wend

end.
```

Example of interfacing CAN transceiver with MCU and bus



CANSPI Library

SPI (Serial Peripheral Interface) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with external CAN modules (such as MCP2515 or MCP2510) via SPI.

CANSPI routines are supported by any PIC MCU model that has SPI interface on portc. Also, CS pin of MCP2510 or MCP2515 must be connected to RC0 pin. Example of HW connection is given at the end of the chapter.

The Controller Area Network module is a serial interface, useful for communicating with other peripherals or microcontrollers. Details about CAN can be found in appropriate literature and on mikroElektronika Web site. MCP2515 or MCP2510 are modules that enable any chip with SPI interface to communicate over CAN bus.

Following routines should be considered a driver for CANSPI (CAN via SPI module) on PIC MCUs.

```
sub procedure CANSPISetOperationMode(dim mode as byte, dim WAIT as byte)

sub function  CANSPIGetOperationMode as byte

sub procedure CANSPIInitialize(dim SJW as byte, dim BRP as byte, dim PHSEG1 as byte,
                           dim PHSEG2 as byte, dim PROPSEG as byte,dim CAN_CONFIG_FLAGS as byte)

sub procedure CANSPISetBaudRate(dim SJW as byte, dim BRP as byte, dim PHSEG1 as byte,
                           dim PHSEG2 as byte, dim PROPSEG as byte,dim CAN_CONFIG_FLAGS as byte)

sub procedure CANSPISetMask(dim CAN_MASK as byte, dim val as longint, dim CAN_CONFIG_FLAGS as
byte)

sub procedure CANSPISetFilter(dim CAN_FILTER as byte, dim val as longint,
                        dim CAN_CONFIG_FLAGS as byte)

sub function  RegsToCANSPIID(dim byref ptr as byte, dim CAN_CONFIG_FLAGS as byte) as longint

sub procedure CANSPIIDToRegs(dim byref ptr as byte, dim val as longint,
                        dim CAN_CONFIG_FLAGS as byte)

sub function  CANSPIwrite(dim id as longint, dim byref Data as byte[8], dim DataLen as byte,
                      dim CAN_TX_MSG_FLAGS as byte) as byte

sub function  CANSPIread(dim byref id as longint, dim byref Data as byte[8],
                     dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte
```

CANSPISetOperationMode

Prototype:	`sub procedure CANSPISetOperationMode(dim mode as byte, dim WAIT as byte)`
Parameters:	mode - Operation mode code can take any of predefined constant values (see the constants below) WAIT - Should have value TRUE(255) or FALSE(0)
Effects:	CAN is set to requested mode
Overview:	Given mode byte is copied to CANSTAT
Note:	If WAIT is true, this is a blocking call. It won't return until requested mode is set. If WAIT is false, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use CANSPIGetOperationMode() to verify correct operation mode before performing mode specific operation.

CANSPIGetOperationMode

Prototype:	`sub function CANSPIGetOperationMode as byte`
Parameters:	None
Output:	Current operational mode of CAN module is returned

CANSPIInitialize

Prototype:

```
sub procedure CANSPIInitialize(dim SJW as byte, dim BRP as byte, dim PHSEG1 as
byte, dim PHSEG2 as byte, dim PROPSEG as byte,dim CAN_CONFIG_FLAGS as byte)
```

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters: SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS value is formed from constants (see below)

Effects: CAN bit rate is set. All masks registers are set to '0' to allow all messages.

Filter registers are set according to flag value:

```
If (CAN_CONFIG_FLAGS and CAN_CONFIG_VALID_XTD_MSG) <> 0
    Set all filters to XTD_MSG
Else if (config and CONFIG_VALID_STD_MSG) <> 0
    Set all filters to STD_MSG
Else
    Set half of the filters to STD, and the rest to XTD_MSG.
```

Side Effects: All pending transmissions are aborted.

CANSPISetBaudRate

Prototype:
```
sub procedure CANSPISetBaudRate(dim SJW as byte, dim BRP as byte, dim PHSEG1 as
byte, dim PHSEG2 as byte, dim PROPSEG as byte,dim AN_CONFIG_FLAGS as byte)
```

Precondition: CAN must be in Configuration mode or else these values will be ignored.

Parameters: SJW value as defined in 18XXX8 datasheet (must be between 1 thru 4)
BRP value as defined in 18XXX8 datasheet (must be between 1 thru 64)
PHSEG1 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PHSEG2 value as defined in 18XXX8 datasheet (must be between 1 thru 8)
PROPSEG value as defined in 18XXX8 datasheet (must be between 1 thru 8)
CAN_CONFIG_FLAGS - Value formed from constants (see section below)

Effects: CAN bit rate is set as per given values.

Overview: Given values are bit adjusted to fit in 18XXX8. BRGCONx registers and copied.

CANSPISetMask

Prototype:
```
sub procedure CANSPISetMask(dim CAN_MASK as byte, dim val as longint, dim
CAN_CONFIG_FLAGS as byte)
```

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_MASK - One of predefined constant value
val - Actual mask register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer mask registers.

CANSPISetFilter

Prototype:

```
sub procedure CANSPISetFilter(dim CAN_FILTER as byte, dim val as longint,
dim CAN_CONFIG_FLAGS as byte)
```

Precondition: CAN must be in Configuration mode. If not, all values will be ignored.

Parameters: CAN_FILTER - One of predefined constant values
val - Actual filter register value.
CAN_CONFIG_FLAGS - Type of message to filter, either
CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG

Effects: Given value is bit adjusted to appropriate buffer filter registers.

RegsTOCANSPIID and CANSPIIDToRegs

Prototypes:

```
sub function  RegsToCANSPIID(dim byref ptr as byte, dim CAN_CONFIG_FLAGS as
byte) as longint

sub procedure CANSPIIDToRegs(dim byref ptr as byte, dim val as longint, dim
CAN_CONFIG_FLAGS as byte)
```

Effects: These two routines are used by other routines (internal purpose only).

CANSPIWrite

Prototype:
```
sub function  CANSPIwrite(dim id as longint, dim byref Data as byte[8], dim
DataLen as byte, dim CAN_TX_MSG_FLAGS as byte) as byte
```

Precondition: CAN must be in Normal mode.

Parameters: id - CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended).
Data - array of bytes up to 8 bytes in length
DataLen - Data length from 1 thru 8.
CAN_TX_MSG_FLAGS - Value formed from constants (see section below)

Effects: If at least one empty transmit buffer is found, given message is queued for the transmission. If none found, FALSE value is returned.

CANSPIRead

Prototype:
```
sub function  CANSPIread(dim byref id as longint, dim byref Data as byte[8],
dim byref DataLen as byte, dim byref CAN_RX_MSG_FLAGS as byte) as byte
```

Precondition: CAN must be in mode in which receiving is possible.

Parameters: id - CAN message identifier
Data - array of bytes up to 8 bytes in length
DataLen - Data length from 1 thru 8.
CAN_TX_MSG_FLAGS - Value formed from constants (see below)

Effects: If at least one full receive buffer is found, it is extracted and returned.
If none found, FALSE value is returned.

CAN Library Constants

You need to be familiar with constants that are provided for use with CANSPI library routines. See how to form values (from constants) that will be passed to or from routines in the example at the end of the chapter. All of the constants are predefined in CAN library.

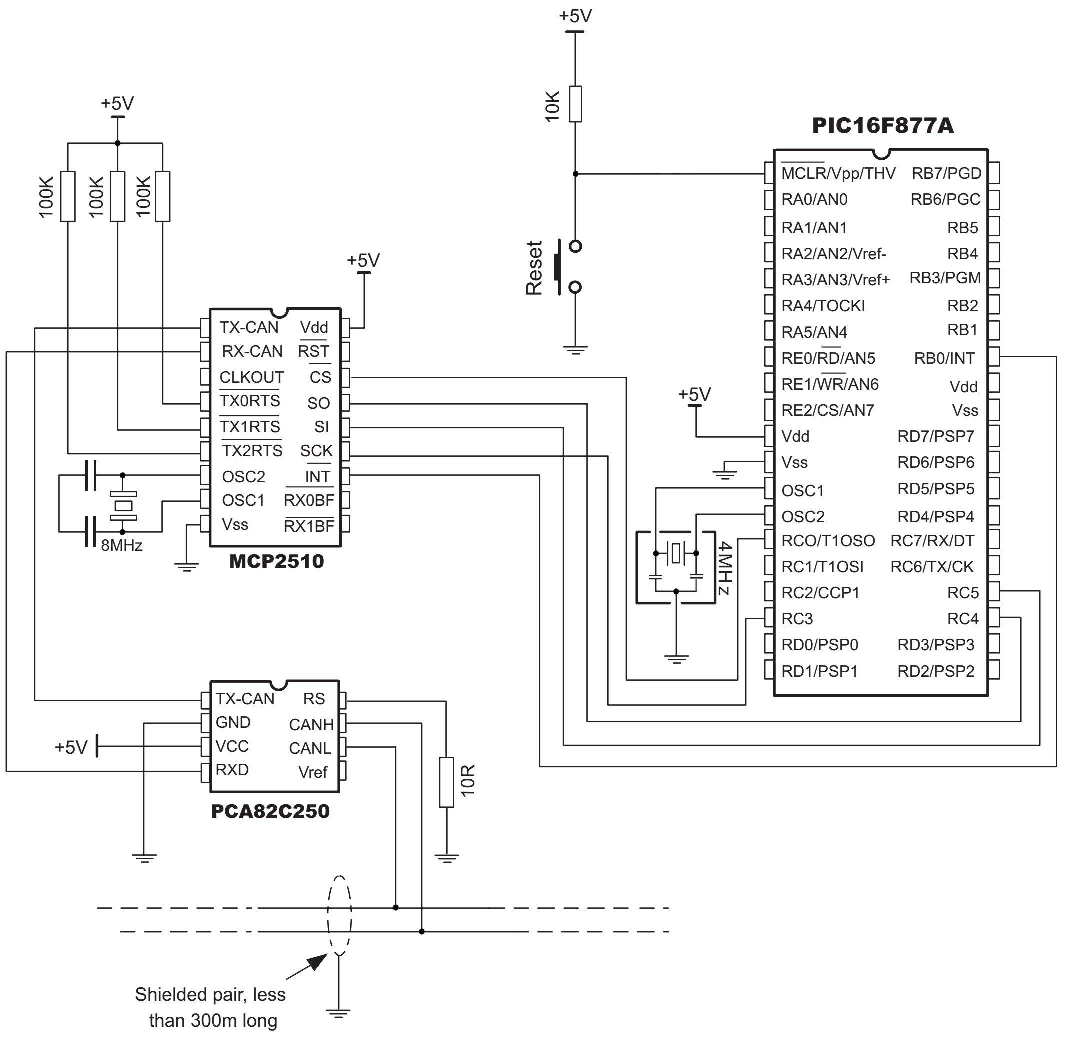
For the complete list of constants refer to page 119.

Example

This code demonstrates use of CANSPI library routines.

```
program CANSPI
dim  aa as byte
dim  aa1 as byte
dim  lenn as byte
dim  aa2 as byte
dim  data as byte[8]
dim  id as longint
dim  zr as byte

main:
  TRISB = 0
  SPI_init           ' must be performed before any other activity
  TRISC.2 = 0        ' this pin is connected to Reset pin of MCP2510
  portc.2 = 0        ' keep MCP2510 in reset state
  PORTC.0 = 1        ' make sure that MCP2510 is not selected
  TRISC.0 = 0        ' make RC0 output
  PORTD  = 0
  TRISD  = 0         ' designate portd as output
  aa = 0
  aa1 = 0
  aa2 = 0
  aa = CAN_CONFIG_SAMPLE_THRICE and
         CAN_CONFIG_PHSEG2_PRG_ON and
         CAN_CONFIG_STD_MSG and
         CAN_CONFIG_DBL_BUFFER_ON and
         CAN_CONFIG_VALID_XTD_MSG        ' prepare flags for
                                         '   CANSPIinitialize
  PORTC.2 = 1                            ' activate MCP2510 chip


  ' continues..
```

```
    ' ..continued

    aa1 = CAN_TX_PRIORITY_BITS and
           CAN_TX_FRAME_BIT and
           CAN_TX_RTR_BIT
    ' prepare flags for CANSPIwrite function

    CANSPIInitialize( 1,2,3,3,1,aa)       ' initialize MCP2510

    CANSPISetOperationMode(CAN_MODE_CONFIG,true)
    ' set configuration mode
    ID = -1

    CANSPISetMask(CAN_MASK_B1,id,CAN_CONFIG_XTD_MSG)
    ' bring all mask1 bits to ones

    CANSPISetMask(CAN_MASK_B2,0,CAN_CONFIG_XTD_MSG)
    ' bring all mask2 bits to ones

    CANSPISetFilter(CAN_FILTER_B1_F1,12111,CAN_CONFIG_XTD_MSG)
    ' set filter_b1_f1 id to 12111

    CANSPISetOperationMode(CAN_MODE_NORMAL,true)
    ' get back to Normal mode

    while true
      zr = CANSPIRead(id , Data , len, aa2)
      if (id = 12111) and zr then
         portd = $AA
         portB = data[0]
         data[0] = data[0]+1
         id = 3
         delay_ms(10)
         CANSPIWrite(id,data,1,aa1)
        if lenn = 2 then
          portd = data[1]
        end if
      end if
    wend

  end.
```



Example of interfacing CAN transceiver MCP2551 and MCP2510 with MCU and bus

Compact Flash Library

Compact Flash Library provides routines for accessing data on Compact Flash card (abbrev. CF further in text). CF cards are widely used memory elements, commonly found in digital cameras. Great capacity (8MB ~ 2GB, and more) and excellent access time of typically few microseconds make them very attractive for microcontroller applications.

Following routines can be used for CF with FAT16, and FAT32 file system. Note that routines for file handling can be used only with FAT16 file system.

File accessing routines can write file up to 128KB in size. File names must be exactly 8 characters long and written in uppercase. User must ensure different names for each file, as CF routines will not check for possible match.

In CF card, data is divided into sectors, one sector usually comprising 512 bytes (few older models have sectors of 256B). Read and write operations are not performed directly, but successively through 512B buffer. Before write operation, make sure you don't overwrite boot or FAT sector as it could make your card on PC or digital cam unreadable. Drive mapping tools, such as Winhex, can be of a great assistance.

Following routines implement data and file access to Compact Flash:

```
sub procedure CF_INIT_PORT(dim byref CtrlPort as byte, dim byref DataPort as byte)
sub function  CF_DETECT(dim byref CtrlPort as byte) as byte
sub procedure CF_WRITE_INIT(dim byref CtrlPort as byte, dim byref DataPort as byte,
                            dim Adr as longint, dim SectCnt as byte)
sub procedure CF_WRITE_BYTE(dim byref CtrlPort as byte, dim byref DataPort as byte,
                            dim BData as byte)
sub procedure CF_WRITE_WORD(dim byref CtrlPort as byte, dim byref DataPort as byte,
                            dim WData as word)
sub procedure CF_READ_INIT(dim byref CtrlPort as byte, dim byref DataPort as byte,
                             dim Adr as longint, dim SectCnt as byte)
sub function  CF_READ_BYTE(dim byref CtrlPort as byte, dim byref DataPort as byte) as byte
sub function  CF_READ_WORD(dim byref CtrlPort as byte, dim byref DataPort as byte) as word
sub procedure CF_SET_REG_ADR(dim byref CtrlPort as byte, dim adr as byte)


sub procedure CF_File_Write_Init(dim byref CtrlPort as byte, dim byref DataPort as byte)
sub procedure CF_File_Write_Byte(dim byref CtrlPort as byte, dim byref DataPort as byte,
                                 dim Bdata as byte)
sub procedure CF_File_Write_Complete(dim byref CtrlPort as byte, dim byref DataPort as byte,
                                     dim byref Filename as char[9])
```

CF_INIT_PORT

Prototype: `sub procedure CF_INIT_PORT(dim byref CtrlPort as byte, dim byref DataPort as byte)`

Precondition: None.

Parameters: *CtrlPort* is control port, *DataPort* is data port to which CF is attached.

Effects: Initializes ports appropriately.

CF_DETECT

Prototype: `sub function  CF_DETECT(dim byref CtrlPort as byte) as byte`

Precondition: *CtrlPort* must be initialized (call CF_INIT_PORT first).

Effects: Check for presence of CF.

Output: Returns TRUE if CF is present, otherwise returns FALSE.

CF_WRITE_INIT

Prototype: `sub procedure CF_WRITE_INIT(dim byref CtrlPort as byte, dim byref DataPort as byte, dim Adr as longint, dim SectCnt as byte)`

Precondition: Ports must be initialized.

Parameters: *CtrlPort* - control port , *DataPort* - data port , *Adr* - specifies sector address from where data will be written, *SectCnt* - parameter is total number of sectors prepared for write.

Effects: Initializes CF card for write operation.

CF_WRITE_BYTE

Prototype:	`sub procedure CF_WRITE_BYTE(dim byref CtrlPort as byte, dim byref DataPort as byte, dim BData as byte)`
Precondition:	Ports must be initialized, CF must be initialized for write operation (see CF_WRITE_INIT).
Parameters:	*CtrlPort* - control port , *DataPort* - data port , *dat* - is data byte written to CF.
Effects:	Write 1 byte to CF. This procedure has effect if writing is previously initialized, and all 512 bytes are transferred to a buffer.

CF_WRITE_WORD

Prototype:	`sub procedure CF_WRITE_WORD(dim byref CtrlPort as byte, dim byref DataPort as byte, dim WData as word)`
Precondition:	Ports must be initialized, CF must be initialized for write operation (see CF_WRITE_INIT).
Parameters:	*CtrlPort* - control port , *DataPort* - data port , *dat* - is data word written to CF.
Effects:	Writes 1 word to CF. This procedure has effect if writing is previously initialized, and all 512 bytes are transferred to a buffer.

CF_READ_INIT

Prototype:	`sub procedure CF_READ_INIT(dim byref CtrlPort as byte, dim byref DataPort as byte, dim Adr as longint, dim SectCnt as byte)`
Precondition:	Ports must be initialized.
Parameters:	*CtrlPort* - control port , *DataPort* - data port , *Adr* - specifies sector address from where data will be read, *SectCnt* - parameter is total number of sectors prepared for read operations.
Effects:	This procedure initializes CF card for write operation.

CF_READ_BYTE

Prototype: `sub function  CF_READ_BYTE(dim byref CtrlPort as byte, dim byref DataPort as byte) as byte`

Precondition: Ports must be initialized, CF must be initialized for read operation (see CF_READ_INIT).

Parameters: *CtrlPort* - control port , *DataPort* - data port.

Effects: Read 1 byte from CF.

CF_READ_WORD

Prototype: `sub function  CF_READ_WORD(dim byref CtrlPort as byte, dim byref DataPort as byte) as word`

Precondition: Ports must be initialized, CF must be initialized for read operation (see CF_READ_INIT).

Parameters: *CtrlPort* - control port , *DataPort* - data port.

Effects: Read 1 word from CF.

CF_SET_REG_ADR

Prototype: `sub procedure CF_SET_REG_ADR(dim byref CtrlPort as byte, dim adr as byte)`

Effects: This procedure is for internal use only.

CF_FILE_WRITE_INIT

Prototype:
```
procedure CF_File_Write_Init(dim byref CtrlPort as byte,
                                                dim byref DataPort as byte)
```

Precondition: Ports must be initialized, CF must be initialized for read operation (see CF_READ_INIT).

Parameters: CtrlPort - control port, DataPort - data port.

Effects: This procedure initializes CF card for file writing operation (FAT16 only).

CF_FILE_WRITE_BYTE

Prototype:
```
procedure CF_File_Write_Byte(dim byref CtrlPort as byte,
                                   dim byref DataPort as byte, dim Bdata as byte)
```

Precondition: Ports must be initialized, CF must be initialized for write operation (see CF_File_Write_Init).

Parameters: CtrlPort - control port, DataPort - data port, Bdata - data byte to be written.

Effects: This procedure adds one byte (<*Bdata*>) to file.

CF_FILE_WRITE_BYTE

Prototype:
```
procedure CF_File_Write_Complete(dim byref CtrlPort as byte,
            dim byref DataPort as byte, dim byref Filename as char[9])
```

Parameters: CtrlPort - control port, DataPort - data port, Filename (**must be in uppercase and must have exactly 8 characters**).

Effects: Upon all data has be written to file, use this procedure to close the file and make it readable by Windows.

Example

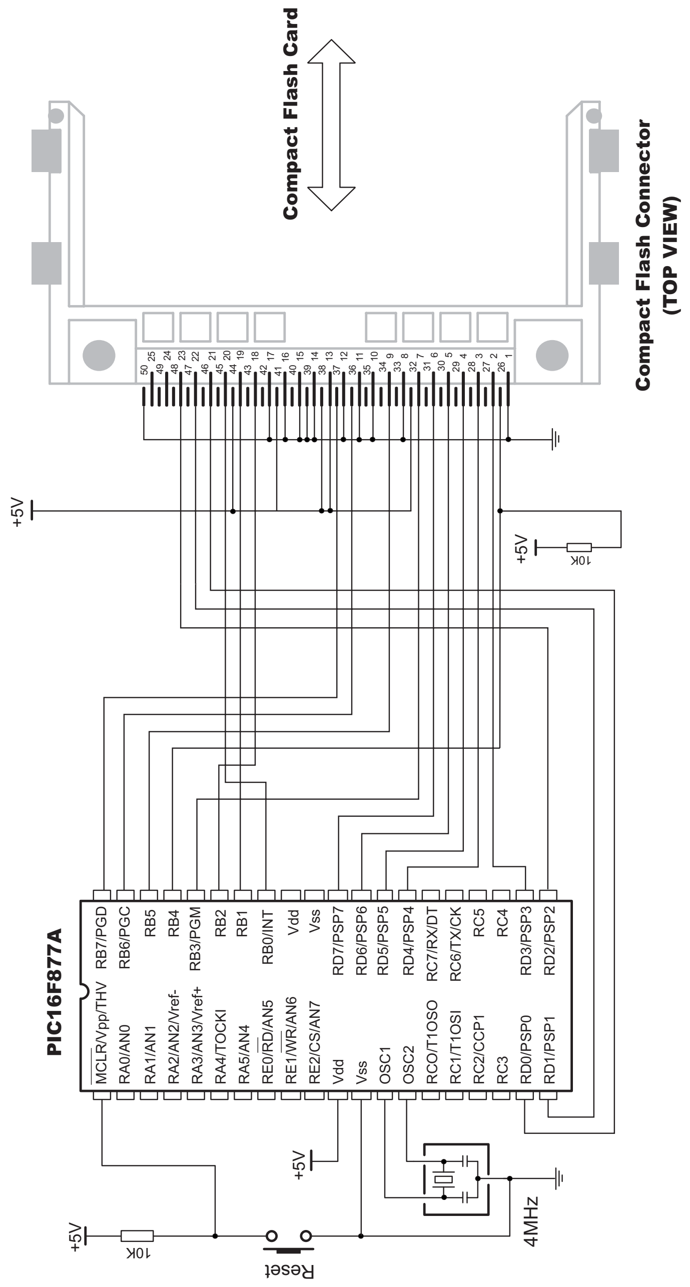
This code demonstrates use of CF library procedures and functions.

```
program CompactFlash

dim i as word
dim temp as longint
dim k as longint


main:
  TRISC = 0                      ' designate portc as output
  CF_INIT_PORT(PORTB,PORTD)      ' initialize ports

  do
    nop
  loop until CF_DETECT(PORTB) = true
  ' wait until CF card is inserted


  Delay_ms(500)

  CF_WRITE_INIT(PORTB, PORTD, 590, 1)
    ' Initialize write at sector address 590
    ' of  1 sector (512 bytes)

  for i = 0 to 511               ' write 512 bytes to sector (590)
    CF_WRITE_BYTE(PORTB,PORTD,i+11)
  next i

  PORTC = $FF
  Delay_ms(1000)

  CF_READ_INIT(PORTB, PORTD, 590, 1)
    ' Initialize write at sector address 590
    '   of  1 sector (512 bytes)

  for i = 0 to 511               ' read 512 bytes from sector (590)
       PORTC = CF_READ_BYTE(PORTB, PORTD)
         ' read byte and display on portc
       Delay_ms(1000)
  next i

end.
```

Example

```
program CompactFlash_File

dim i1 as word
dim index as byte
dim Fname as string[9]

sub procedure Init
 TRISC = 0    ' designate portc as output
 CF_Init_Port(PORTB,PORTD)           ' initialize ports
do
   nop
loop until CF_DETECT(PORTB) = true ' wait until CF card is inserted
Delay_ms(50)                    ' wait for until the card stabilizes
end sub

main:

  index = 0                     ' index of file to be written
  while index < 5
     portc = 0
     Init
     portc = index
     CF_File_Write_Init(PORTB,PORTD) ' initialization for writing
                                     ' to new file
     i1 = 0
     while i1 < 50000
       CF_File_Write_Byte(PORTB,PORTD,48+index)   ' writes 50000
                                                   ' bytes to file
       inc(i1)
     wend
     Fname = "RILEPROX" ' must be 8 character long in upper case
     fname[8] = 48 + index ' ensure that files have different name
     CF_File_Write_Complete(PORTB,PORTD, Fname)   ' close the file
     Inc(index)
   wend
   PORTC = $FF

end.
```

Figure: Pin diagram of Compact Flash memory card



EEPROM Library

EEPROM data memory is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide you comfortable work with EEPROM.

Routines

Basically, there are two operations that can be performed on EEPROM data memory.

```
function  EEprom_Read(dim Address as byte) as byte
procedure EEprom_Write(dim Address as byte, dim Data as byte)
```

Library function EEprom_Read reads data from specified *Address*, while the library procedure EEprom_Write writes *Data* to specified *Address*.

Note

Parameter *Address* is of byte type, which means it can address only 256 locations. For PIC18 MCU models with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately.

Example

```
program EEPROMtest
dim i as byte
dim j as byte

main:
  TRISB = 0
  for i = 0 to 20
   EEprom_write(i,i+6)
  next i

  for i = 0 to 20
     PORTB = EEprom_read(i)
     for j = 0 to 200
       delay_us(500)
     next j
  next i
end.
```

I2C Library

I2C (Inter Integrated Circuit) full master MSSP (Master Synchronous Serial Port) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to support the master I2C mode.

Important

Note that these functions support module on PORTC, and won't work with modules on other ports. Examples for PIC MCUs with module on other ports can be found in your mikroBasic installation folder, subfolder 'examples'.

Routines

I2C interface is serial interface used for communicating with peripheral or other microcontroller devices. All functions and procedures bellow are intended for PIC MCUs with MSSP module. By using these, you can configure and use PIC MCU as master in I2C communication.

```
sub procedure I2C_Init(const clock as longint)
sub function  I2C_Is_Idle as byte
sub function  I2C_Start as byte
sub procedure I2C_Repeated_Start
sub function  I2C_Wr(dim Data as byte) as byte
sub function  I2C_Rd(dim Ack as byte) as byte
sub procedure I2C_Stop as byte
```

```
sub procedure I2C_Init(const clock as longint)
```

Parameter *clock* is a desired I2C clock (refer to device data sheet for correct values in respect with Fosc).

Example:

```
I2C_init(100000)
```

After configuring the I2C master mode, you have the following functions and procedures at your disposal:

```
sub function I2C_Start as byte
```

Determines if I2C bus is free and issues START condition; if there is no error, function returns 0.

```
sub procedure I2C_Repeated_Start
```

Performs repeated start condition.

```
sub function I2C_Wr(dim Data as byte) as byte
```

After you have issued a start or repeated start you can send data byte via I2C bus; this function also returns 0 if there is no errors.

```
sub function I2C_Rd(dim Ack as byte) as byte
```

Receives 1 byte from the slave; and sends not acknowledge signal if parameter Ack is 0 in all other cases it sends acknowledge.

```
sub procedure I2C_Stop as byte
```

Issues STOP condition.

Example

The following code demonstrates use of I2C Library procedures and functions. PIC MCU is connected (SCL,SDA pins) to 24c02 EEPROM. Program sends data to EEPROM (data is written at address 2). Then, we read data via I2C from EEPROM and send its value to PORTD, to check if the cycle was successful. See the following figure on how to interface 24c02 to PIC.

```
' Example of communication with 24c02 EEPROM

program BasicI2c

dim  EE_adr as byte
dim  EE_data as byte
dim  jj as word

main:
     I2C_init(100000)       ' initialize full master mode
     TRISD = 0              ' designate portd as output
     PORTD = $ff            ' initialize portd
     I2C_Start              ' issue I2C start signal
     I2C_Wr($a2)            ' send byte via I2C(command to 24cO2)
     EE_adr  = 2
     I2C_Wr(EE_adr)         ' send byte(address for EEPROM)
     EE_data = $aa
     I2C_Wr(EE_data)        ' send data to be written
     I2C_Stop               ' issue I2C stop signal

     for jj = 0 to 65500    ' pause while EEPROM writes data
         nop
     next i

     I2C_Start              ' issue I2C start signal
     I2C_Wr($a2)            ' send byte via I2C
     EE_adr = 2
     I2C_Wr(EE_adr)         ' send byte(address for EEPROM)
     I2C_Repeated_Start     ' issue I2Csignal repeated start
     I2C_Wr($a3)            ' send byte(request data from EEPROM)
     EE_data = I2C_rd(1)    ' Read the data
     I2C_Stop               ' issue I2C stop signal
     PORTD = EE_data        ' show data on PORTD

noend:                      ' endless loop
     goto noend
end.
```

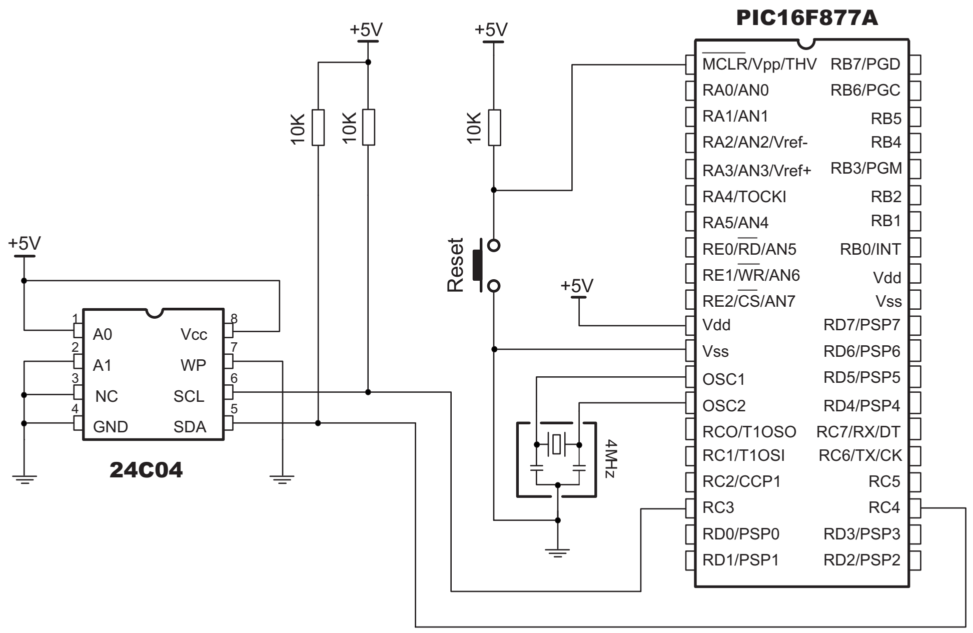


Figure: I2C interfacing EEPROM 24C04 to PIC MCU

LCD Library

mikroBasic provides a set of library procedures and functions for communicating with commonly used 4-bit interface LCD (with Hitachi HD44780 controller). Figure showing HW connection of PIC and LCD is given at the bottom of the page (if you need different pin settings, refer to LCD_Config routine).

Note

Be sure to designate port with LCD as output, before using any of the following library routines.

Routines

```
sub procedure LCD_Config(dim byref Port as byte, const RS,
                         const EN, const WR, const D7,
                         const D6, const D5, const D4)
```

Initializes LCD at <*Port*> with pin settings you specify:
parameters <*RS*>, <*EN*>, <*WR*>, <*D7*> .. <*D4*> need to be a combination of values 0..7 (e.g. 3,6,0,7,2,1,4).

```
sub procedure LCD_Init(dim byref Port as byte)
```

Initializes LCD at <*Port*> with default pin settings (check the figures at the end of the chapter).

```
sub procedure LCD_Out(dim Row as byte, dim Column as byte,
                      dim byref Text as char[255])
```

Prints <*Text*> (string variable) at specified row and column on LCD. Both string variables and string constants can be passed.

```
sub procedure LCD_Out_CP(dim byref Text as char[255])
```

Prints <*Text*> (string variable) at current cursor position. Both string variables and string constants can be passed.

```
sub procedure LCD_Chr(dim Row as byte, dim Column as byte,
                      dim Character as byte)
```

Prints <*Character*> at specified row and column on LCD.

```
sub procedure LCD_Chr_CP(dim Character as byte)
```

Prints <*Character*> at current cursor position.

```
sub procedure LCD_Cmd(dim Command as byte)
```

Sends command <*Command*> to LCD. Refer to the following list of available commands.

LCD Commands

Command	**Purpose**
LCD_First_Row	Moves cursor to 1st row
LCD_Second_Row	Moves cursor to 2nd row
LCD_Third_Row	Moves cursor to 3rd row
LCD_Fourth_Row	Moves cursor to 4th row
LCD_Clear	Clears display
LCD_Return_Home	Returns cursor to home position, returns a shifted display to original position. Display data RAM is unaffected
LCD_Cursor_Off	Turns off cursor
LCD_Underline_On	Underline cursor on
LCD_Blink_Cursor_On	Blink cursor on
LCD_Move_Cursor_Left	Move cursor left without changing display data RAM
LCD_Move_Cursor_Right	Move cursor right without changing display data RAM
LCD_Turn_On	Turn LCD display on
LCD_Turn_Off	Turn LCD display off
LCD_Shift_Left	Shift display left without changing display data RAM
LCD_Shift_Right	Shift display right without changing display data RAM

Example

Here are several examples of LCD routine calls:

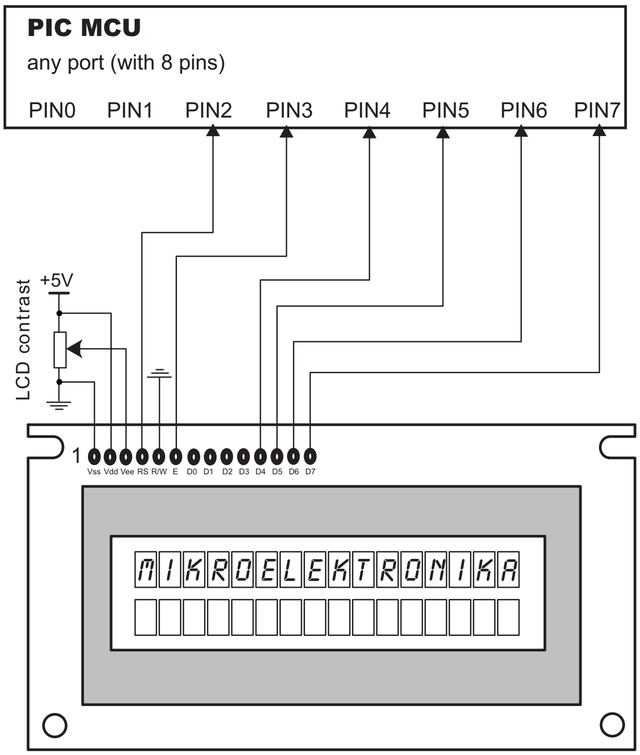
```
LCD_Config(PORTD,0,1,2,6,5,4,3)
 ' Initializes LCD on PORTD with custom pin settings
 ' (4-bit interface).

LCD_Init(PORTB)
 ' Initializes LCD on PORTB with default pin settings
 ' (4-bit interface).

LCD_Out(1,1,txt)
 ' Prints string variable <txt> on LCD (1st row, 1st column).

LCD_Out_CP(txt)
 ' Prints string variable <txt> at current cursor position.

LCD_Char(1,1,"e")
 ' Prints character "e" on LCD (1st row, 1st column).

LCD_Char_CP("f")
 ' Prints character "f" at current cursor position.

LCD_Cmd(LCD_Clear)
 ' Sends command LCD_Clear to LCD (clears LCD display).
```

Example

Use LCD_Init for default pin settings (see the figure below).

```
program LCD_default_test

main:
  TRISB = 0                                 ' PORTB is output
  LCD_Init(PORTB)                           ' Initialize LCD at PORTB
  LCD_Cmd(LCD_CURSOR_OFF)                   ' Turn off cursor
  LCD_Out(1, 1, "mikroelektronika")         ' Print Text at LCD
end.
```

PIC		LCD
PIN7	⟷	D7
PIN6	⟷	D6
PIN5	⟷	D5
PIN4	⟷	D4
PIN3	⟷	E
PIN2	⟷	RS
PIN1		
PIN0		

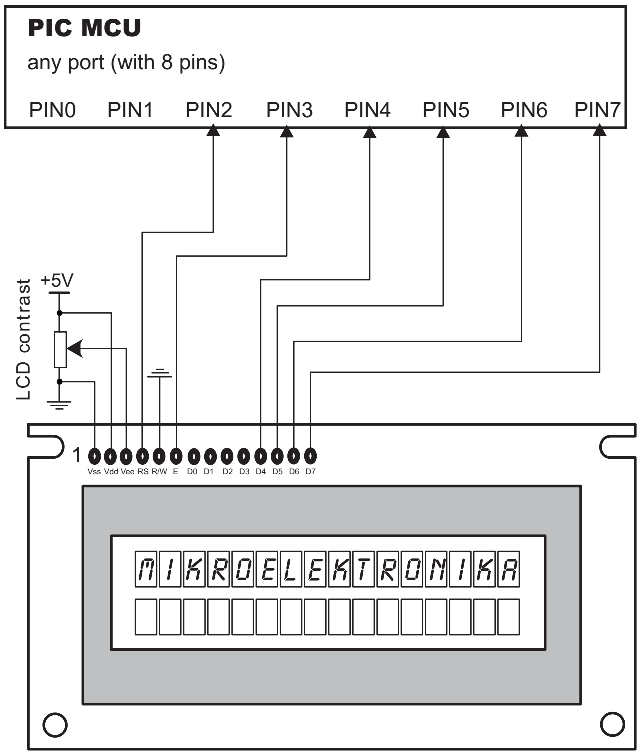


LCD HW connection by default initialization (using LCD_Init). If you need different pin settings, refer to LCD_Config routine.

Alternatively, you can use LCD_Config for custom pin settings. For example:

```
program LCD_custom_test


main:
  TRISD = 0                             ' PORTD is output

  ' Initialize LCD at portd with custom pin settings (figure)
  LCD_Config(PORTD,1,2,0,3,5,4,6)
  LCD_Cmd(LCD_CURSOR_OFF)               ' Turn off cursor
  LCD_Out(1, 1, "mikroelektronika")     ' Print text at LCD
end.
```

PIC		LCD
PIN7	⟷	D7
PIN6	⟷	D6
PIN5	⟷	D5
PIN4	⟷	D4
PIN3	⟷	E
PIN2	⟷	RS
PIN1		
PIN0		



LCD HW custom connection (using LCD_Config, see the example above).

LCD8 Library (8-bit interface LCD)

mikroPascal provides a set of library procedures and functions for communicating with commonly used 8-bit interface LCD (with Hitachi HD44780 controller). Figure showing HW connection of PIC and LCD is given on the following page (if you need different pin settings, refer to LCD8_Config routine).

Note

Be sure to designate port with LCD as output, before using any of the following library routines.

Routines

```
sub procedure LCD8_Config(dim byref portCtrl as byte,
        dim byref portData as byte, const RS, const EN, const WR,
        const D7, const D6, const D5, const D4, const D3, const D2,
        const D1, const D0)
```

Initializes LCD at <*portCtrl*> and <*portData*> with pin settings you specify: parameters <RS>, <EN>, <WR> need to be in range 0..7;
parameters <D7> .. <D0> need to be a combination of values 0..7
(e.g. 3,6,5,0,7,2,1,4).

```
sub procedure LCD8_Init(dim byref portCtrl as byte,
                        dim byref portData as byte)
```

Initializes LCD at <*portCtrl*> and <*portData*> with default pin settings (check the figures at the end of the chapter).

```
sub procedure LCD8_Out(dim Row as byte, dim Column as byte,
                      dim byref Text as char[255])
```

Prints <*Text*> (string variable) at specified row and column on LCD. Both string variables and string constants can be passed.

```
sub procedure LCD8_Out_CP(dim byref Text as char[255])
```

Prints <*Text*> (string variable) at current cursor position. Both string variables and string constants can be passed.

```
sub procedure LCD8_Chr(dim Row as byte, dim Column as byte,
                       dim Character as byte)
```

Prints <*Character*> at specified row and column on LCD.

```
sub procedure LCD8_Chr_CP(dim Character as byte)
```

Prints <*Character*> at current cursor position.

```
sub procedure LCD8_Cmd(dim Command as byte)
```

Sends command <*Command*> to LCD. Refer to page 150 for the complete list of available LCD commands.

Example

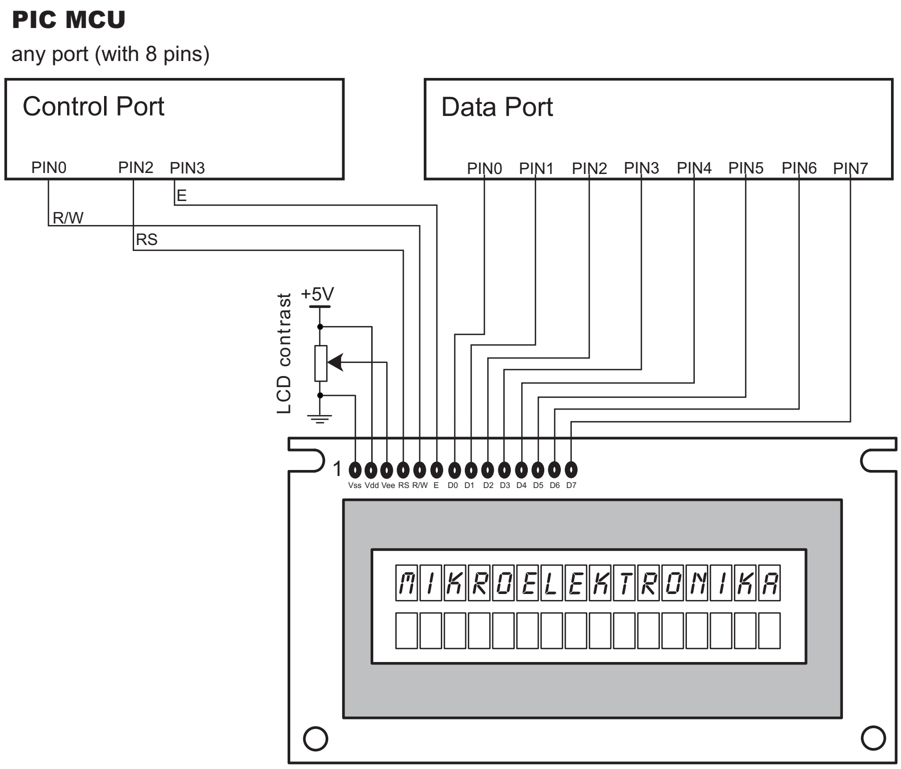
Here are several examples of LCD8 routine calls:

```
LCD8_Config(PORTC,PORTD,0,1,2,6,5,4,3,7,1,2,0)
 ' Initializes LCD on PORTC and PORTD with custom pin settings
 ' (8-bit interface).

LCD8_Init(PORTB,PORTC)
 ' Initializes LCD on PORTB and PORTC with default pin settings
 ' (8-bit interface).

LCD8_Out(1,1,txt)
 ' Prints string variable <txt> on LCD (1st row, 1st column).

LCD8_Out_CP(txt)
 ' Prints string variable <txt> at current cursor position.

LCD8_Char(1,1,"e")
 ' Prints character "e" on LCD (1st row, 1st column).

LCD8_Char_CP("f")
 ' Prints character "f" at current cursor position.

LCD8_Cmd(LCD_Clear)
 ' Sends command LCD_Clear to LCD (clears LCD display).
```

Example

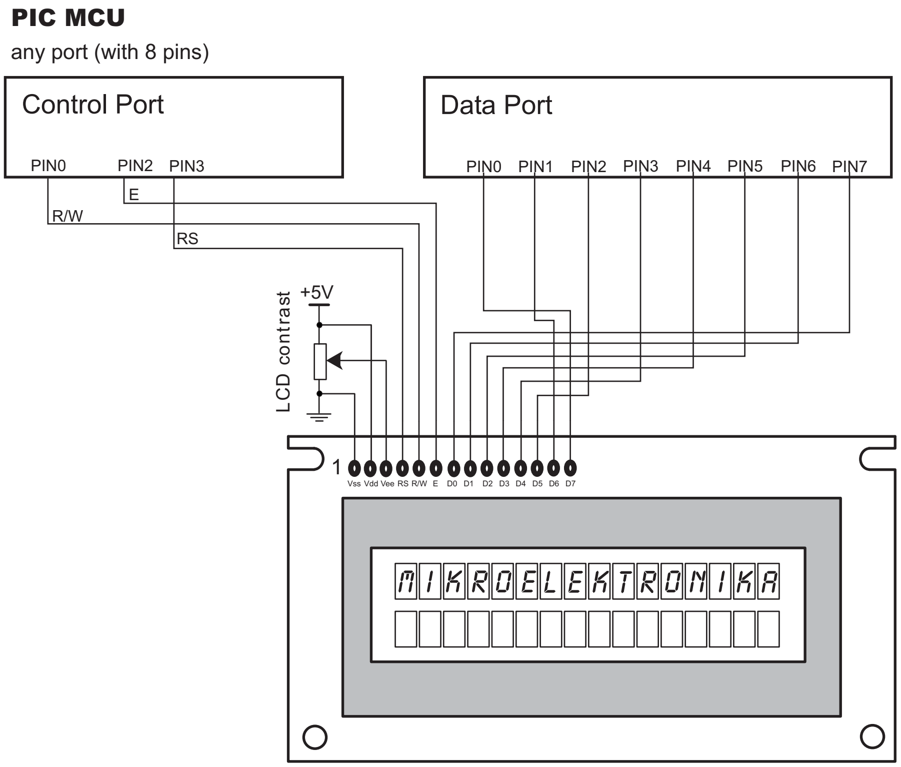
Use LCD8_Init for default pin settings (see the figure below).

```
program LCD8_test

dim Text as char[17]

main:
  TRISB = 0                                         ' Portb is output
  TRISC = 0                                         ' Portc is output
  LCD8_Init(portb, portc)          ' Initialize LCD at portb and portc
  LCD8_Cmd(LCD_CURSOR_OFF)                          ' Turn off cursor
  Text = "mikroElektronika"
  LCD8_Out(1, 1, Text)                              ' Print text at LCD
end.
```



Alternatively, you can use LCD8_Config to set custom pin settings. For example:

```
program LCD8_custom_test

dim Text as char[17]

main:
  TRISB = 0                                              ' Portb is output
  TRISD = 0                                              ' Portd is output
  ' Initialize LCD at portb and portd
  LCD8_Config(PORTB,PORTD,2,3,0,7,6,5,4,3,2,1,0)
  LCD8_Cmd(LCD_CURSOR_OFF)                               ' Turn off cursor
  Text = "mikroElektronika"
  LCD8_Out(1, 1, Text)                                   ' Print text at LCD
end.
```



Graphic LCD Library

mikroBasic provides a set of library procedures and functions for drawing and writing on Graphical LCD. Also it is possible to convert bitmap (use menu option Tools > BMP2LCD) to constant array and display it on GLCD. These routines work with commonly used GLCD 128x64, and work only with the PIC18 family.

Note

Be sure to designate ports with GLCD as output, before using any of the following library procedures or functions.

Routines

```
sub procedure GLCD_Init(dim Ctrl_Port as byte,
                        dim Data_Port as byte)
```

Initializes LCD at <*Ctrl_Port*> and <*Data_Port*>.

```
sub procedure GLCD_Config(dim byref Ctrl_Port as byte,
               dim byref Data_Port as byte, dim Reset as byte,
               dim Enable as byte, dim RS as byte, dim RW as byte,
               dim CS1 as byte, dim CS2 as byte)
```

Initializes LCD at <*Ctrl_Port*> and <*Data_Port*> with custom pin settings. For example: `GLCD_Config(portb, portc, 1,7,4,6,0,2).`

```
sub procedure GLCD_Put_Ins(dim ins as byte)
```

Sends instruction <*ins*> to GLCD. Available instructions include:

```
 X_ADRESS     = $B8        ' Adress base for Page 0
 Y_ADRESS     = $40        ' Adress base for Y0
 START_LINE   = $C0        ' Adress base for line 0
 DISPLAY_ON   = $3F        ' Turn display on
 DISPLAY_OFF  = $3E        ' Turn display off
```

```
sub procedure GLCD_Put_Data(dim data as byte)
```

Sends data byte to GLCD.

```
sub procedure GLCD_Put_Data2(dim data as byte,
                             dim side as byte)
```

Sends data byte to GLCD at specified <*side*>.

```
sub procedure GLCD_Select_Side(dim LCDSide as byte)
```

Selects the side of the GLCD:

```
 ' const RIGHT =  0
 ' const LEFT  =  1
```

```
sub function GLCD_Data_Read as byte
```

Reads data from GLCD.

```
sub procedure GLCD_Set_Dot(dim x as byte, dim y as byte)
```

Draws a dot on the GLCD.

```
sub procedure GLCD_Clear_Dot(dim x as byte, dim y as byte)
```

Clears a dot on the GLCD.

```
sub procedure GLCD_Circle(dim CenterX as integer,
                   dim CenterY as integer, dim Radius as integer)
```

Draws a circle on the GLCD, centered at <*CenterX, CenterY*> with <*Radius*>.

```
sub procedure GLCD_Line(dim x1 as integer, dim y1 as integer,
                        dim x2 as integer, dim y2 as integer)
```

Draws a line from (x1,y1) to (x2,y2).

```
sub procedure GLCD_Invert(dim Xaxis as byte, dim Yaxis as byte)
```

Procedure inverts display (changes dot state on/off) in the specified area, X pixels wide starting from 0 position, 8 pixels high. Parameter X spans 0..127, parameter Y spans 0..7 (8 text lines).

```
sub procedure GLCD_Goto_XY(dim x as byte, dim y as byte)
```

Sets cursor to dot (x,y). Procedure is used in combination with GLCD_Put_Data, GLCD_Put_Data2, and GLCD_Put_Char.

```
sub procedure GLCD_Put_Char(dim Character as byte)
```

Prints <*Character*> at cursor position.

```
sub procedure GLCD_Clear_Screen
```

Clears the GLCD screen.

```
sub procedure GLCD_Put_Text(dim x_pos as word, dim y_pos as word,
                  dim byref text as char[25], dim invert as byte)
```

Prints <*text*> at specified position; y_pos spans 0..7.

```
sub procedure GLCD_Rectangle(dim X1 as byte, dim Y1 as byte,
                            dim X2 as byte, dim Y2 as byte)
```

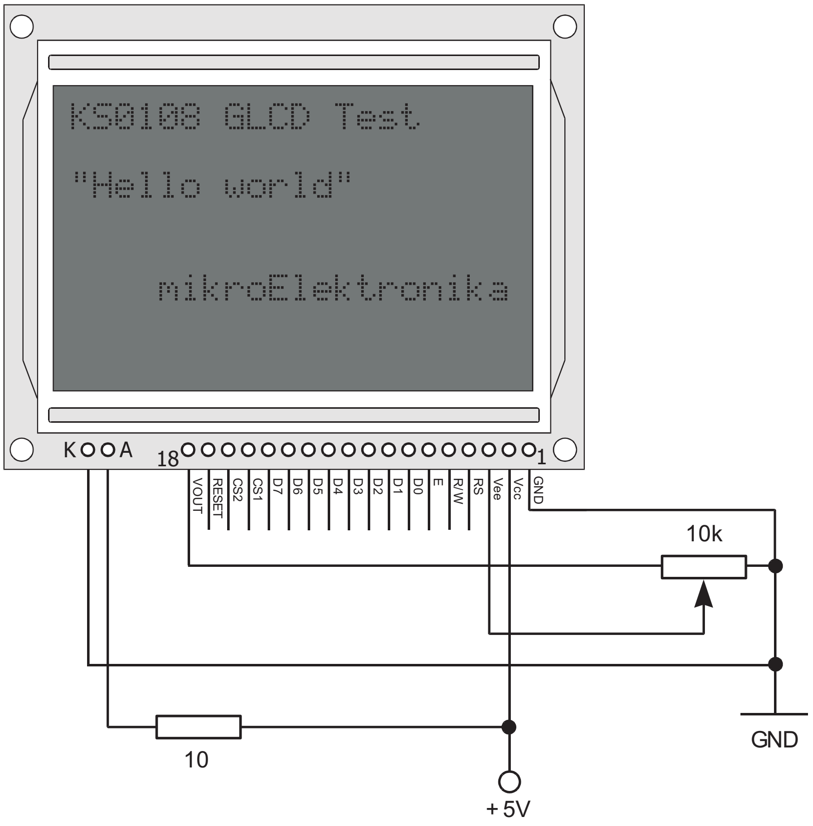
Draws a rectangle on the GLCD. (x1,y1) sets the upper left corner, (x2,y2) sets the lower right corner.

```
sub procedure GLCD_Set_Font(dim font_index as byte)
```

Sets font for GLCD. Parameter <*font_index*> spans from 1 to 4, and determines which font will be used: 1: 5x8 dots, 2: 5x7 dots, 3: 3x6 dots, 4: 8x8 dots.

Example

```
program GLCDDemo

include "GLCD_128x64.pbas"

dim text as string[25]
dim j as byte
dim k as byte

main:
  PORTC = 0
  PORTB = 0
  PORTD = 0
  TRISC = 0
  TRISD = 0
  TRISB = 0

  GLCD_LCD_Init(PORTC, PORTD)   ' default settings
  GLCD_Set_Font(FONT_NORMAL1)

  while true
     ' Draw Circles
      GLCD_Clear_Screen
      text = "Circle"
      GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT)
      GLCD_Circle(63,31,20)
      Delay_Ms(4000)

     ' Draw rectangles
      GLCD_Clear_Screen
      text = "Rectangle"
      GLCD_Put_Text(0, 7, text, NONINVERTED_TEXT)
      GLCD_Rectangle(10, 0, 30, 35)
      Delay_Ms(4000)
      GLCD_Clear_Screen

     ' Draw Lines
      GLCD_Clear_Screen
      text = "Line"
      GLCD_Put_Text(55, 7, text, NONINVERTED_TEXT)
      GLCD_Line(0, 0, 127, 50)
      GLCD_Line(0, 63, 50, 0)
      Delay_Ms(5000)

  { continued.. }
{ ..continued }

      ' Fonts DEMO
       GLCD_Clear_Screen
       text = "Fonts DEMO"
       GLCD_Set_Font(FONT_TINY)
       GLCD_Put_Text(0, 4, text, NONINVERTED_TEXT)
       GLCD_Put_Text(0, 5, text, INVERTED_TEXT)
       GLCD_Set_Font(FONT_BIG)
       GLCD_Put_Text(0, 6, text, NONINVERTED_TEXT)
       GLCD_Put_Text(0, 7, text, INVERTED_TEXT)
       Delay_ms(5000)
     wend
end.
```



PWM Library

CCP (Capture/ Compare/ PWM) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with PWM (Pulse Width Modulation).

Note

Note that these routines support module on PORTC pin RC2, and won't work with modules on other ports. Also, mikroBasic doesn't support enhanced PWM modules. Examples for PIC MCUs with module on other ports can be found in your mikroBasic installation folder, subfolder 'examples'.

Routines

```
sub procedure PWM_Init(const PWM_Freq)
sub procedure PWM_Change_Duty(dim New_Duty as byte)
sub procedure PWM_start
sub procedure PWM_stop
```

```
procedure PWM_Init(const PWM_Freq);
```

Initializes the PWM module. It starts with (duty ratio) 0%.
Parameter *PWM_Freq* is a desired PWM frequency (refer to device data sheet for correct values in respect with Fosc).

Example: `PWM_Init(5000);`

```
sub procedure PWM_Change_Duty(dim New_Duty as byte)
```

Parameter *New_Duty* (duty ratio) takes values from 0 to 255, where 0 is 0% duty ratio, 127 is 50% duty ratio, and 255 is 100% duty ratio. Other values for specific duty ratio can be calculated as (Percent*255)/100.

```
sub procedure PWM_start
```

Starts PWM.

```
sub procedure PWM_stop
```

Stops PWM.

Example

This code demonstrates use of PWM library procedures and functions. If pin RC2 is connected to LED diode, light emitted will depend of PWM duty ratio and this change can be noticed.

```
program PWMtest

dim j as byte

main:
      j = 0
      PORTC = $FF
      PWM_init(5000)          ' initializes PWM module, freq = 5kHz
      PWM_start               ' starts PWM
      while true
          delay_ms(100)
          j = j + 1
          PWM_change_duty(j)   ' changes duty ratio
      wend
end.
```

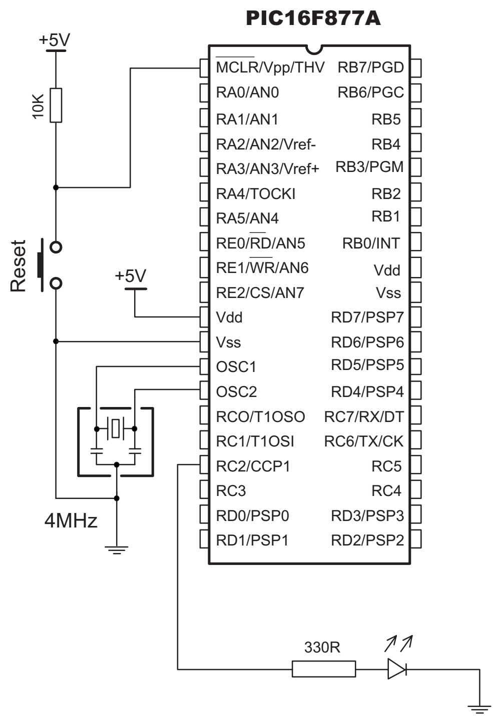


Figure: PWM demonstration

RS485 Library

RS485 is a multipoint communication which allows multiple devices to be connected to a single signal cable. mikroBasic provides a set of library routines to provide you comfortable work with RS485 system using Master/Slave architecture.

Master and Slave devices interchange packets of information, each of these packets containing synchronization bytes, CRC byte, address byte, and the data. In Master/Slave architecture, Slave can never initiate communication. Each Slave has its unique address and receives only the packets containing that particular address. It is programmer's responsibility to ensure that only one device transmits data via 485 bus at a time.

Address 50 is a common address for all Slave devices: packets containing address 50 will be received by all Slaves. The only exceptions are Slaves with addresses 150 and 169, which require their particular address to be specified in the packet.

Note

RS485 routines require USART module on port C. Pins of USART need to be attached to RS485 interface transceiver, such as LTC485 or similar. Pins of transceiver (Receiver Output Enable and Driver Outputs Enable) should be connected to port C, pin 2 (see the figure at end of the chapter).

Routines

Following routines implement flexible protocol for RS485 system with Master/Slave architecture:

```
sub procedure RS485master_init

sub procedure RS485slave_init(dim address as byte)

sub procedure RS485master_read(dim byref data as byte[5])

sub procedure RS485master_write(dim byref data as byte[2],
                              dim datalen as byte, dim address as byte)

sub procedure RS485slave_read(dim byref data as byte[5])

sub procedure RS485slave_write(dim byref data as byte[2],
                             dim datalen as byte)
```

RS485master_init

Prototype: `sub procedure RS485master_init`

Precondition: USART needs to be initialized (USART_init)

Parameters: None

Effects: Initializes MCU as Master in RS485 communication

RS485slave_init

Prototype: `sub procedure RS485slave_init(dim address as byte)`

Precondition: USART needs to be initialized (USART_init)

Parameters: Slave address can take any value between 0 and 255, except 50 which is a common address for all slaves

Effects: Initializes MCU as Slave in RS485 communication

RS485master_read

Prototype: `sub procedure RS485master_read(dim byref data as byte[5])`

Precondition: MCU must be initialized as Master to assign an address to MCU.

Parameters: `dim byref data as byte[5]`

Effects: Master receives any message sent by Slaves. As messages are multi-byte, this procedure must be called for each byte received. Upon receiving a message, buffer is filled with the following values:

data[0..2] is data; data[3] is the number of received bytes (1..3); data[4] is set to 255 (TRUE) when message is received; data[5] is set to 255 (TRUE) if an error has occurred; data[6] is the address of the Slave which sent the message

Procedure automatically sets data[4] and data[5] upon every received message.These flags need to be cleared repeatedly from the program.

RS485master_write

Prototype:
```
sub procedure RS485master_write(dim byref data as byte[2],
                                dim datalen as byte, dim address as byte)
```

Precondition: MCU must be initialized as Master in 485 communication. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Parameters: `dim byref data as byte[2], dim datalen as byte`

Effects: Sends number of bytes (1 < datalen <= 3) from buffer via 485.

RS485slave_read

Prototype: `sub procedure RS485slave_read(dim byref data as byte[5])`

Precondition: MCU must be initialized as Slave in 485 communication.

Parameters: `dim byref data as byte[5]`

Effects: Only messages that appropriately address Slaves will be received. As messages are multi-byte, this procedure must be called for each byte received (see the example at the end of the chapter). Upon receiving a message, buffer is filled with the following values:

data[0..2] is data; data[3] is number of bytes received (1..3) ; data[4] is set to 255(TRUE) when message is received; data[5] is set to 255(TRUE) if an error has occurred; rest of the buffer is undefined

Procedure automatically sets data[4] and data[5] upon every received message. These flags need to be cleared repeatedly from the program.

RS485slave_write

Prototype: `sub procedure RS485slave_write(dim byref data as byte[2],`
`dim datalen as byte)`

Precondition: MCU must be initialized as Slave in 485 communication.

Parameters: `dim byref data as byte[2], dim datalen as byte`

Effects: Sends number of bytes (1 < datalen <= 3) from buffer via 485

Example

```
program pr485

dim dat as byte[8]          ' buffer for receiving/sending messages
dim i as byte
dim j as byte

sub procedure interrupt
 if TestBit(RCSTA,OERR) = 1 then
   portd = $81
 end if

 RS485slave_receive(dat)    ' every byte is received by
end sub                     '     RS485slave_read(dat);
                            ' upon receiving a msg with no errors
main:                       '     data[4] is set to 255
  trisb = 0
  trisd = 0
  USART_init(9600)          ' initialize usart module
  RS485slave_init(160)      ' init. MCU as Slave with address 160
  SetBit(PIE1,RCIE)         ' enable interrupt
  SetBit(INTCON,PEIE)       '     on byte received
  ClearBit(PIE2,TXIE)       '     via USART (RS485)
  SetBit(INTCON,GIE)
  portb = 0
  portd = 0                 ' ensure that message received flag is 0
  dat[4] = 0                ' ensure that error flag is 0
  dat[5] = 0
  while true
    if dat[5] then
     portd = $aa            ' if there is error, set portd to $aa
    end if
    if dat[4] then          ' if message received
    dat[4] = 0              '     clear message received flag
      j = dat[3]            '     number of data bytes received
      for i = 1 to j
        portb = dat[i-1]    '     output received data bytes
      next i
      dat[0] = dat[0] + 1         '     increment received dat[0]
      RS485slave_send(dat,1)      '     send it back to Master
    end if
  wend

end.
```

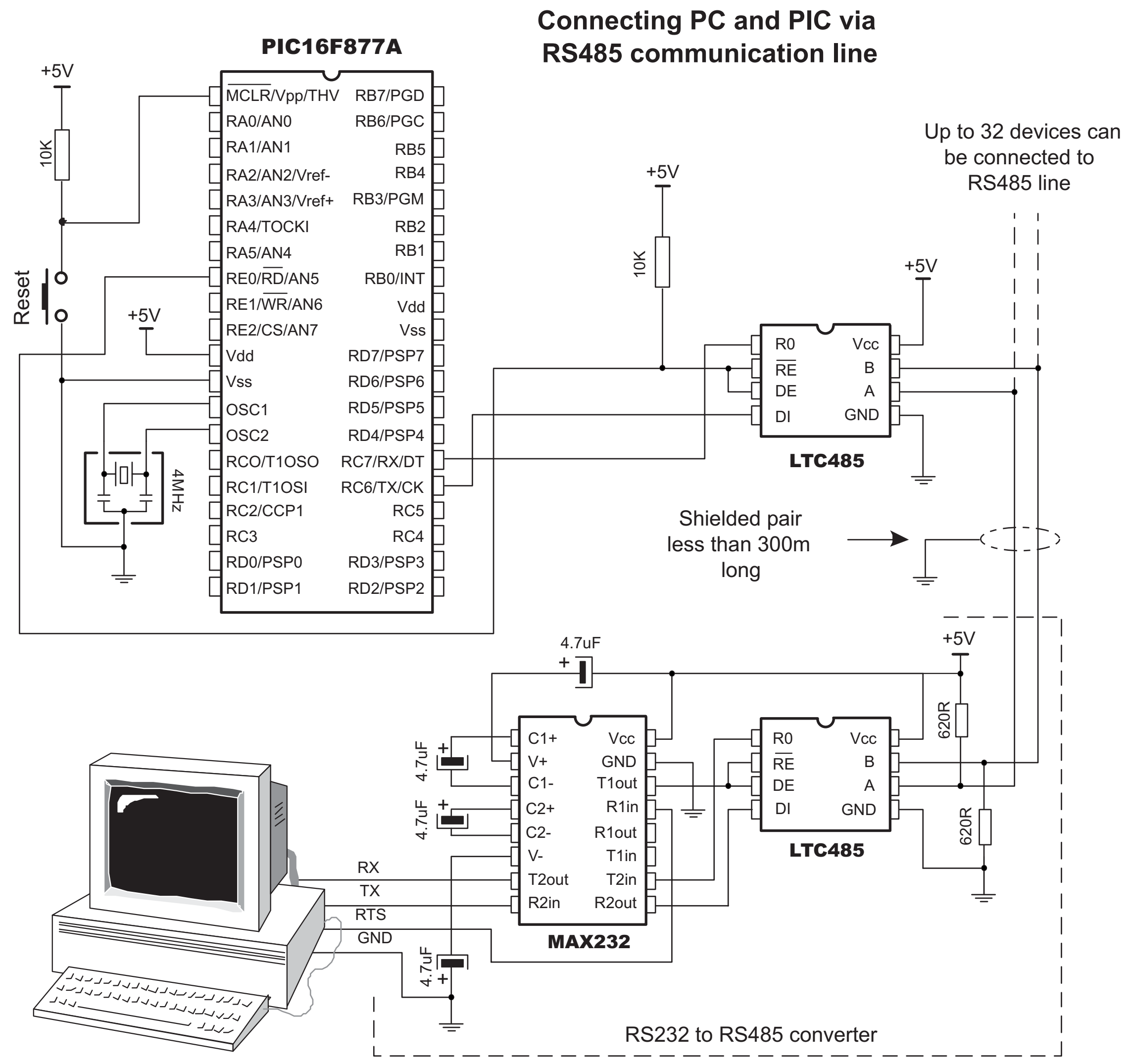


Figure: Example of interfacing PC to PIC MCU via RS485 bus

SPI Library

SPI (Serial Peripheral Interface) module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide initialization of slave mode and comfortable work with the master mode.

You can easily communicate with other devices via SPI - A/D converters, D/A converters, MAX7219, LTC1290 etc. You need PIC MCU with hardware integrated SPI (for example, PIC16F877). Then, simply use the following functions and procedures.

Note

Note that these functions support module on PORTB or PORTC, and won't work with modules on other ports. Examples for PIC MCUs with module on other ports can be found in your mikroBasic installation folder, subfolder 'examples'.

Routines

```
sub procedure SPI_init
sub procedure SPI_write(dim Data as byte)
sub function  SPI_read(dim Buffer as byte) as byte
sub procedure SPI_Init_advanced(dim Master as byte,
dim Data_Sample as byte, dim Clock_Idle as byte,dim Low_To_High as byte)
```

Initialization

You can use procedure `SPI_init` without parameters and get the default result:

Master mode, clock Fosc/4, clock idle state low, data transmitted on low to high edge, input data sampled at the middle of interval;

For advanced settings, configure and initialize SPI using the procedure:

```
sub procedure SPI_Init_advanced(dim Master as byte,
                   dim Data_Sample as byte, dim Clock_Idle as byte,
                   dim Low_To_High as byte)
```

Example:

```
SPI_init(Master_OSC_div4, Data_SAMPLE_MIDDLE,LK_Idle_LOW,LOW_2_HIGH)
```

This will set SPI to master mode, clock = Fosc/4, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge.

Parameters

Parameter *mast_slav* determines the work mode for SPI; can have the following values:

Value	**Meaning**
Master_OSC_div4	Master clock=Fosc/4
Master_OSC_div16	Master clock=Fosc/16
Master_OSC_div64	Master clock=Fosc/64
Master_TMR2	Master clock source TMR2
Slave_SS_ENABLE	Master slave select enabled
Slave_SS_DIS	Master slave select disabled

Parameter *Data_sample* determines when data is sampled. It can have the following values:

Value	**Meaning**
Data_SAMPLE_MIDDLE	Input data sampled in middle of interval
Data_SAMPLE_END	Input data sampled at end of interval

Parameter *clk_idl* determines idle state for clock; can have the following values:

Value	**Meaning**
CLK_Idle_HIGH	Clock idle HIGH
CLK_Idle_LOW	Clock idle LOW

Parameter *lth_htl* determines transmit edge for data. It can have the following values:

Value	**Meaning**
LOW_2_HIGH	Data transmit on low to high edge
HIGH_2_LOW	Data transmit on high to low edge

Note

In order to keep this working, you shouldn't override the settings made by the procedures spi_init or spi_init_ordinary as it uses some of the PIC MCU resources.

Pins RC3, RC4, RC5 are configured as needed (don't change TRISC settings for these pins - procedure will set them automatically).

Read and Write

The following routines are provided for comfortable use of master mode :

```
sub procedure SPI_write(dim Data as byte)
```

Write byte b to SSPBUF, and immediately starts the transmission.

```
sub function SPI_read(dim Buffer as byte)
```

Provide clock by sending data (byte b) and read the received data at the end of the period.

Example

The folowing code demonstrates how to use SPI library procedures and functions. Same example along with m7219.pbas file is given in folder ../mikroBasic/examples. Assumed HW configuration is: max7219 (chip select pin) is connected to RC1, and SDO, SDI, SCK pins are connected to corresponding pins of max7219.

```
program SPI

include "m7219.pbas"

dim i as byte

main:
     SPI_init                        ' standard configuration
     TRISC = TRISC and $Fd
     max7219_init                    ' initialize max7219
     PORTC.1 = 0                     ' select max7219
     SPI_write(1)                    ' send address (1) to max7219
     SPI_write(7)                    ' send data (7) to max7219
     PORTC.1 = 0                     ' deselect max7219s
end.
```

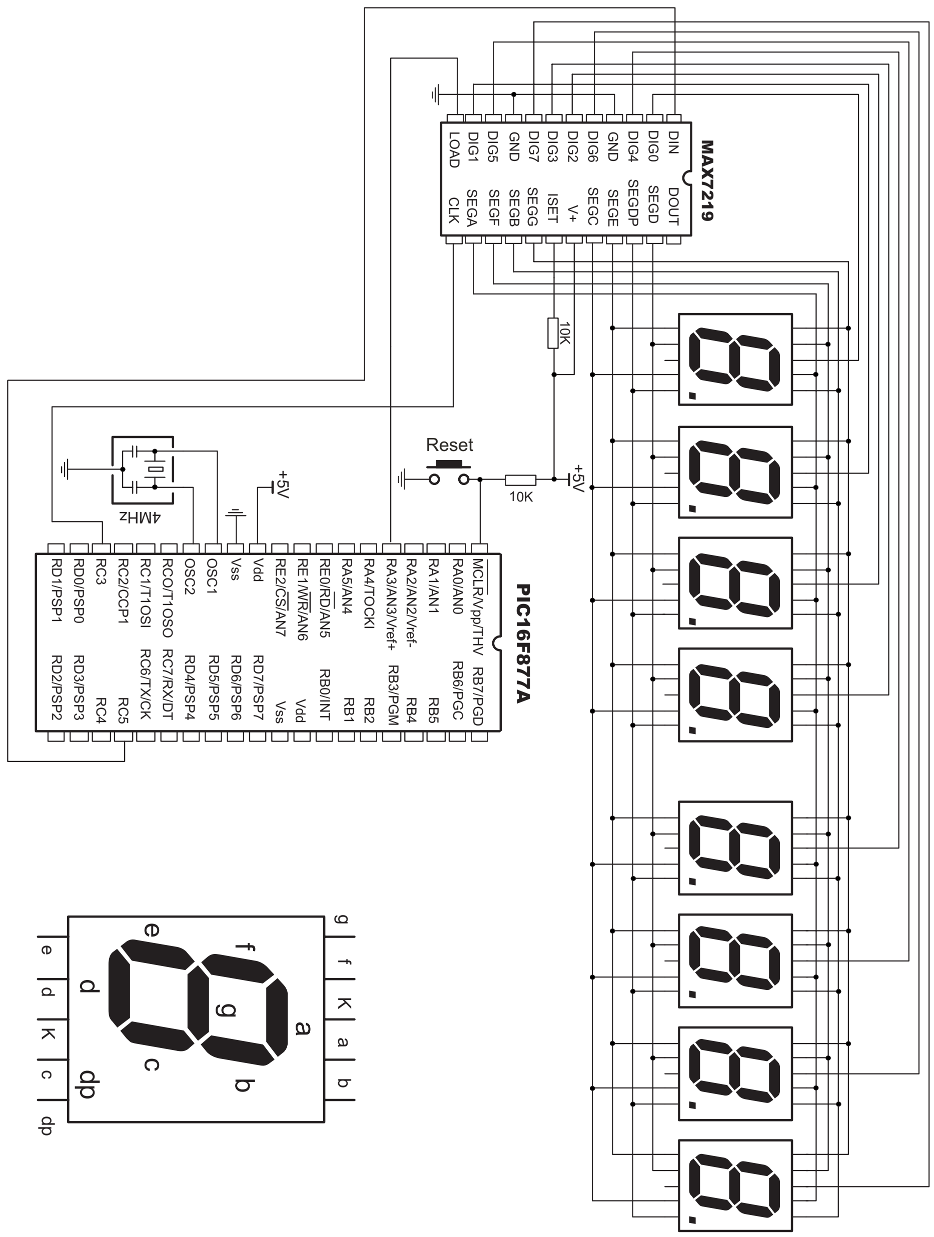


Figure: Example of interfacing MAX7219 with PIC MCU via SPI

USART Library

USART (Universal Synchronous Asynchronous Receiver Transmitter) hardware module is available with a number of PIC MCU models. Set of library procedures and functions is listed below to provide comfortable work with the Asynchronous (full duplex) mode.

You can easily communicate with other devices via RS232 protocol (for example with PC, see the figure at the end of this chapter - RS232 HW connection). You need a PIC MCU with hardware integrated USART (for example, PIC16F877). Then, simply use the functions and procedures described below.

Note

Note that these functions and procedures support module on PORTB, PORTC or PORTG, and won't work with modules on other ports. Examples for PIC MCUs with module on other ports can be found in your mikroBasic installation folder, subfolder 'examples'.

Routines

```
sub procedure USART_Init(const Baud_Rate)
sub function  USART_Data_Ready as byte
sub function  USART_Read as byte
sub procedure USART_Write(dim Data as byte)
```

Certain PIC MCU models with two USART modules, such as P18F8520, require you to specify the module you want to use. Simply append the number 1 or 2 to procedure or function name - for example, `USART_Write2(dim Data as byte)`.

```
sub procedure USART_Init(const Baud_Rate)
```

Parameter *Baud_rate* is the desired baud rate;

Example:

```
USART_init(2400)
```

This will initialize PIC MCU USART hardware and establish the communication at baud rate of 2400.

Refer to the device data sheet for baud rates allowed for specific Fosc. If you specify the unsupported baud rate, compiler will report an error.

In order to keep this working, you should not override settings made by the procedure USART_init as it uses some of the PIC MCU resources. (For example: pins RC7, RC6 configured as input, output respectively; do not change TRISC settings for this pins - procedure will set them automatically). Check the figure on the following page.

Following routines can be used after the communication has been established:

```
sub function USART_Data_Ready as byte
```

Returns 1 if data is ready; returns 0 if there is no data.

```
sub function USART_Read as byte
```

Receive a byte; if byte is not received return 0.

```
sub procedure USART_Write(dim Data as byte)
```

Transmit a byte.

Example

The following code demonstrates how to use USART library procedures and functions. When PIC MCU receives data via rs232 it immediately sends the same data back. If PIC MCU is connected to the PC (see figure below), you can test it using mikroBasic terminal for RS232 communication, menu choice Tools > Terminal.

Example

```
program RS232com

dim  Received_byte as byte

main:
  USART_init(2400)                          ' initialize USART module
  while true
    if USART_data_ready = 1 then            ' if data is received
       Received_byte = USART_read           '    read received data,
       USART_write(Received_byte)           '    send data via USART
    end if
  wend
end.
```

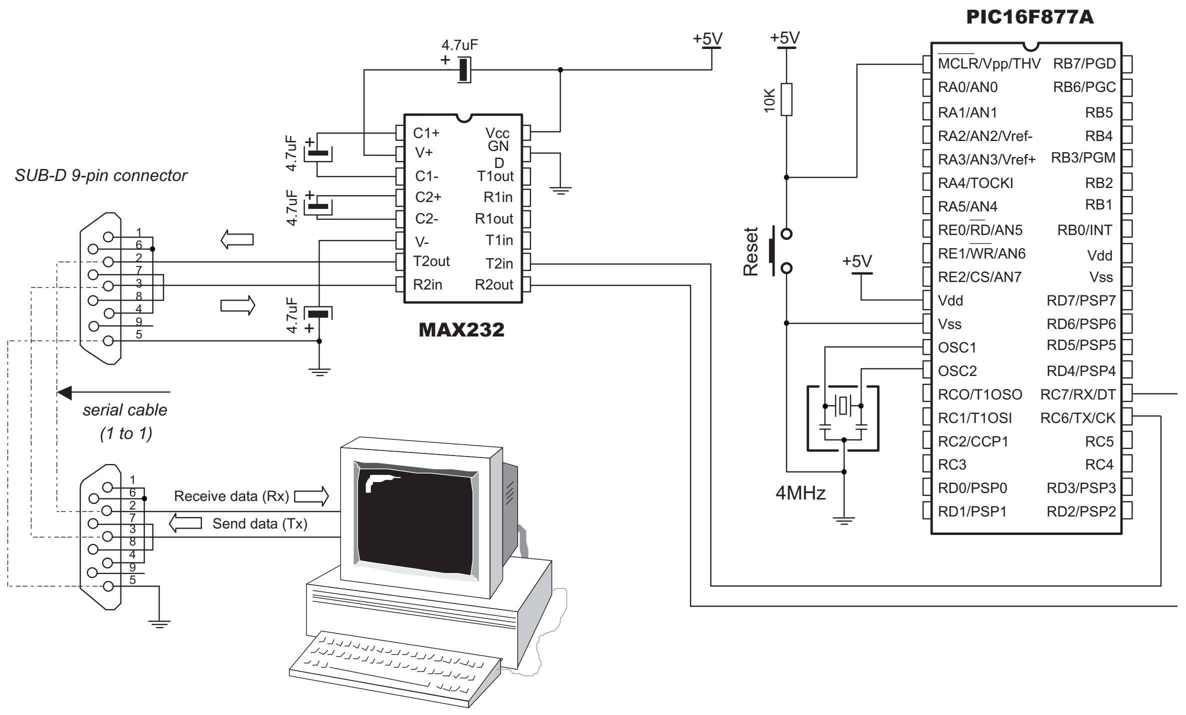


Figure: RS232 HW connection

Software I2C

mikroBasic provides routines which implement software I2C. These routines are hardware independent and can be used with any MCU. Software I2C enables you to use MCU as Master in I2C communication. Multi-master mode is not supported. Note that these functions and procedures implement time-based activities, so the interrupts must be disabled when using them.

I2C interface is serial interface used for communicating with peripheral or other microcontroller devices. Routines below are intended for PIC MCUs with MSSP module. By using these, you can configure and use PIC MCU as master in I2C communication.

Routines

```
sub procedure Soft_I2C_Config(dim byref Port as byte, const SDA,
                              const SCL, const clock)
```

Parameter <Port> specifies port of MCU on which SDA and SCL pins will be located; parameters <SCL> and <SDA> need to be in range 0..7 and cannot point at the same pin.

```
sub procedure Soft_I2C_Start
```

Issues START condition.

```
sub function Soft_I2C_Write(dim Data as byte) as byte
```

After you have issued a start or repeated start you can send data byte via I2C bus; this function also returns 0 if there are no errors.

```
sub function Soft_I2C_Read(dim Ack as byte) as byte
```

Receive 1 byte from the slave; and sends not acknowledge signal if parameter Ack is 0 in all other cases it sends acknowledge.

```
sub procedure Soft_I2C_Stop
```

Issues STOP condition.

Example

This code demonstrates use of software I2C routines. PIC MCU is connected (SCL,SDA pins) to 24c02 EEPROM. Program sends data to EEPROM (data is written at address 2). Then, we read data via I2C from EEPROM and send its value to PORTC, to check if the cycle was successful.

```
program soft_I2C_test

dim EE_adr as byte
dim EE_data as byte
dim jj as word

main:
    Soft_I2C_config(PORTD,3,4)    ' initialize full master mode
    TRISC = 0                     ' portc is output
    PORTC = $ff                   ' initialize portc
    Soft_I2C_Start                ' I2C start signal
    Soft_I2C_Write($a2)           ' send byte via I2C
    EE_adr  = 2
    Soft_I2C_Write(EE_adr)        ' send byte(address for EEPROM)
    EE_data = $aa
    Soft_I2C_Write(EE_data)       ' send data (data to be written)
    Soft_I2C_Stop                 ' I2C stop signal

    for jj = 0 to 65500           ' pause while EEPROM writes data
        nop
    next jj

    Soft_I2C_Start                ' issue I2C start signal
    Soft_I2C_Write($a2)           ' send byte via I2C
    EE_adr = 2
    Soft_I2C_Write(EE_adr)        ' send byte (address for EEPROM)
    Soft_I2C_Start                ' I2C signal repeated start
    Soft_I2C_Write($a3)           ' send byte (request data)
    EE_data = Soft_I2C_Read(0)    ' read the data
    Soft_I2C_Stop                 ' I2C_stop signal
    PORTC = EE_data               ' show data on PORTD
noend:                            ' endless loop
    goto noend
end.
```

Software SPI

mikroBasic provides routines which implement software SPI. These routines are hardware independent and can be used with any MCU.

Note

Note that these functions and procedures implement time-based activities, so the interrupts need to be disabled when using them.

Routines

```
sub procedure Soft_SPI_Config(dim byref Port as byte,
                              const SDI, const SD0, const SCK)
sub procedure Soft_SPI_Init(dim byref Port as byte)
sub procedure Soft_SPI_Write(dim Data as byte)
sub function  Soft_SPI_Read(dim Buffer as byte) as byte
```

Configure and initialize SPI using the procedure `Soft_SPI_Config`. Example:

```
Soft_SPI_Config(PORTB,1,2,3)
```

This will set SPI to master mode, clock = 50kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge. SDI pin is RB1, SDO pin is RB2 and SCK pin is RB3.

Parameter <Port> specifies port of MCU on which SDI,SDO and SCK pins will be located; parameters <SDI>, <SDO> and <SCK> need to be in range 0..7 and cannot point at the same pin;

In order to keep this working, you shouldn't override the settings made by the procedures soft_spi_config as it uses some of the PIC MCU resources. Specified pins SDI,SDO and SCK are configured as needed (don't change TRISX settings for these pins - procedure will set them automatically).

The following functions are provided for comfortable use of master mode:

```
sub procedure Soft_SPI_Write(dim Data as byte)
```

Immediately transmit byte Data.

```
sub function Soft_SPI_Read(dim Buffer as byte) as byte
```

Provide clock by sending data (byte Buffer) and return the received data.

Example

This code demonstrates how to use Software SPI procedures and functions. Assumed HW configuration is: max7219 (chip select pin) is connected to RD1, and SDO, SDI, SCK pins are connected to corresponding pins of max7219.

```
program Soft_SPI_test

include "m7219.pbas"

dim i as byte

main:
     Soft_SPI_Config(portd,4,5,3)       ' standard configuration
     TRISC = TRISC and $Fd
     max7219_init                       ' initialize max7219
     PORTD.1 = 0                        ' select max7219
     Soft_SPI_Write(1)                  ' send address to max7219
     Soft_SPI_Write(7)                  ' send data to max7219
     PORTD.1 = 0                        ' deselect max7219
end.
```

Software UART

mikroBasic provides routines which implement software UART. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via RS232 protocol. Simply use the functions and procedures described below.

Note

Note that these functions and procedures implement time-based activities, so the interrupts need to be disabled when using them.

Routines

```
sub procedure Soft_UART_Init(dim byref Port as byte,
                             const RX, const TX, const Baud_Rate)
sub function Soft_UART_Read(dim byref Msg_received as byte) as byte
sub procedure Soft_UART_Write(dim Data as byte)
```

Parameter <Port> specifies port of MCU on which RX and TX pins are located (RX and TX have to be on the same port, obviously); parameters <RX> and <TX> need to be in range 0..7 and cannot point the same pin; <Baud_Rate> is the desired baud rate.

Example:

```
Soft_UART_Init(portb, 1, 2, 9600)
```

This will initialize software UART and establish the communication at baud rate of 9600. Maximum baud rate depends on PIC MCU clock and working conditions.

In order to keep this working, you should not override settings made by the procedure `Soft_UART_Init` as it uses some of PIC resources. (the example above configures pins RB1 and as input; do not change TRISB settings for these pins - procedure will set them automatically).

Following functions can be used after the communication has been established:

```
sub function Soft_UART_Read(dim byref Msg_received as byte) as byte
```

Function returns a received byte; parameter <Msg_received> will take true if transfer was succesful. `Soft_UART_Read` is a non-blocking function call, so you should test <Msg_received> manually (check the example below).

```
sub procedure Soft_UART_Write(dim Data as byte)
```

Procedure transmits a byte.

Example

This code demonstrates how to use software UART procedures and functions. When PIC MCU receives data via RS232 it immediately sends the same data back. If PIC MCU is connected to the PC, you can test it using the mikroBasic terminal for RS232 communication, menu choice Tools > Terminal.

Be aware that during transmission, software UART is incapable of receiving data - data transfer protocol must be set in such a way to prevent loss of information.

```
program soft_UART_test

dim Received_byte as byte
dim Rec_ok as byte

main:
  Soft_UART_init(PORTB,1,2,2400)          ' initialize software UART
  while true
    do
       ' read received data, loop until Rec_ok
       Received_byte = Soft_UART_read(Rec_ok)

       ' send data via UART
       Soft_UART_write(Received_byte)
  wend
end.
```

Flash Memory Library

This library provides routines for accessing microcontroller Flash memory. Note that routines differ for PIC16 and PIC18 families.

Routines

For PIC18:

```
procedure Flash_Write(dim Address as longint,
                      dim byref Data as byte[64])
function Flash_Read(dim Address as longint) as byte
```

For PIC16:

```
procedure Flash_Write(dim Address as word, dim Data as word)
function Flash_Read(dim Address as word) as word
```

Procedure `FlashWrite` writes chunk of data to Flash memory (for PIC18, data needs to exactly 64 bytes in size).

Procedure `FlashRead` reads data from the specified <*Address*>.

Important

Keep in mind that this function erases target memory before writing <*Data*> to it. This means that if write was unsuccessful, your previous data will be lost.

Example

Demonstration of Flash Memory Library for PIC18:

```
program flash_pic18

const FLASH_ERROR   = $FF
const FLASH_OK      = $AA

dim toRead as byte
dim i as byte
dim toWrite as byte[64]

main:
  TRISB = 0                                 ' PORTB is output
  for i = 0 to 63                           ' initialize array
    toWrite[i] = i
  next i

  ' write contents of the array to the address 0x0D00
  Flash_Write($0D00, toWrite)

  ' verify write
  PORTB  = 0                                ' turn off PORTB
  toRead = FLASH_ERROR                      ' initialize error state

  for i = 0 to 63

    ' read 64 consecutive locations starting from 0x0D00
    toRead = Flash_Read($0D00 + i)

    if toRead <> toWrite[i] then            ' stop on first error
      PORTB = FLASH_ERROR                   ' indicate error
      Delay_ms(500)

    else
        PORTB = FLASH_OK                    ' indicate no error
    end if

  next i

end.
```

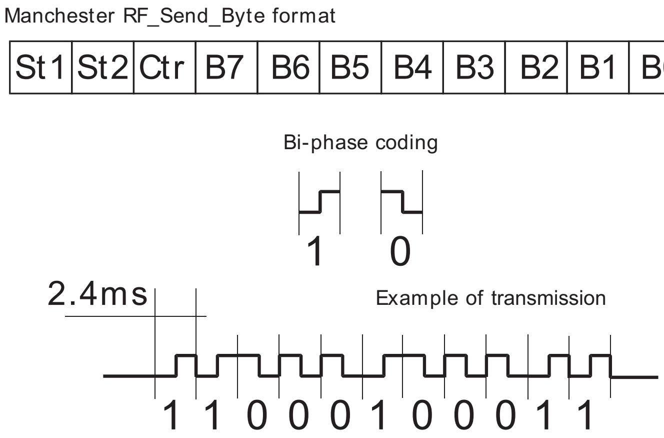
Demonstration of Flash Memory Library for PIC16:

```
program flash_pic16_test

const FLASH_ERROR = $FF
const FLASH_OK    = $AA

dim toRead as word
dim i as word

main:
  TRISB = 0                                 ' PORTB is output
  for i = 0 to 63

    ' write the value of i starting from the address 0x0A00
    Flash_Write(i + $0A00, i)

  next i

  ' verify write
  PORTB  = 0                                ' turn off PORTB
  toRead = FLASH_ERROR                      ' initialize error state

  for i = 0 to 63

    ' Read 64 consecutive locations starting from 0x0A00
    toRead = Flash_Read($0A00 + i)

    if toRead <> i then                     ' Stop on first error

      ' i contains the address of the erroneous location
      i = i + $0A00
      PORTB = FLASH_ERROR                   ' indicate error
      Delay_ms(500)

    else
      PORTB = FLASH_OK                      ' indicate no error
    end if

  next i

end.
```

Manchester Code Library

mikroBasic provides a set of library procedures and functions for handling Manchester coded signal.

Manchester code is a code in which data and clock signals are combined to form a single self-synchronizing data stream; each encoded bit contains a transition at the midpoint of a bit period, the direction of transition determines whether the bit is a "0" or a "1"; second half is the true bit value and the first half is the complement of the true bit value (as shown in the figure below).



Note

Manchester receive routines are blocking calls (`Man_Receive_Config`, `Man_Receive_Init`, `Man_Receive`). This means that PIC will wait until the task is performed (e.g. byte is received, synchronization achieved, etc.)

Routines for receiving are limited to a baud rate scope from 340 ~ 560 bps.

Routines

```
sub procedure Man_Receive_Config(dim byref port as byte, dim rxpin as byte)
sub procedure Man_Receive_Init(dim byref port as byte)
sub function  Man_Receive(dim byref error as byte) as byte
sub procedure Man_Send_Config(dim byref port as byte, dim txpin as byte)
sub procedure Man_Send_Init(dim byref port as byte)
sub procedure Man_Send(dim data as byte)
```

```
sub procedure Man_Receive_Config(dim byref port as byte, dim rxpin as byte)
```

This procedure needs to be called in order to receive signal by procedure `Man_Receive`. You need to specify the port and rxpin of input signal. In case of multiple errors on reception, you should call `Man_Receive_Init` once again to enable synchronization.

```
sub procedure Man_Receive_Init(dim byref port as byte)
```

Procedure works same as `Man_Receive_Config`, with default pin setting (pin 6).

```
sub function  Man_Receive(dim byref error as byte) as byte
```

Function extracts one byte from signal. If format does not match the expected, <*error*> flag will be set True.

```
sub procedure Man_Send_Config(dim byref port as byte, dim txpin as byte)
```

Procedure needs to be called in order to send signals via procedure `Man_Send`. Procedure specifies <*port*> and <*txpin*> for outgoing signal (const baud rate).

```
sub procedure Man_Send_Init(dim byref port as byte)
```

Procedure works same as `Man_Send_Config`, but with default pin setting (pin 0).

```
sub procedure Man_Send(dim data as byte)
```

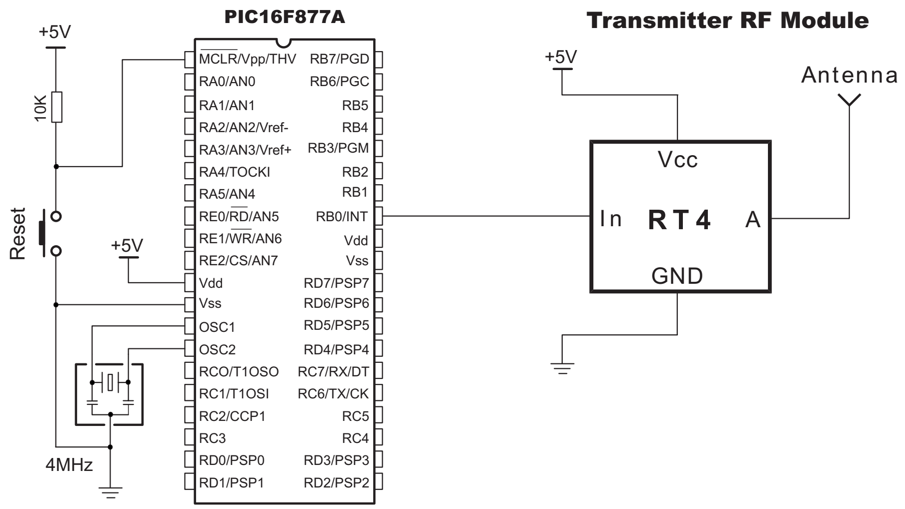
This procedure sends one <*data*> byte.

Example Following code receives message in Manchester code:

```
program RRX

dim ErrorFlag as byte
dim ErrorCount as byte
dim IdleCount as byte
dim temp as byte
dim LetterCount as byte

main:
  errorCount = 0
  TRISC      = 0                         ' errorFlag indicator
  PORTC      = 0
  Man_Receive_Init(PORTD)                ' Synchronize receiver
  LCD_Init(PORTB)                        ' Initialize LCD on PORTB

  while true
    do                                   ' endless loop
      IdleCount = 0                      ' Reset idle counter
      temp = Man_Receive(ErrorFlag)      ' Attempt byte receive
      if errorFlag then
        inc(errorCount)
      else
        PORTC = 0
      end if
      if errorCount >  20 then           ' If too many errorFlags
                    '   try to synchronize the receiver again
          errorCount = 0
          PORTC = $AA                    ' Indicate errorFlag
          Man_Receive_Init(PORTD)        ' Synchronize receiver
        end if
      inc(IdleCount)
      if IdleCount > 18 then

                  ' If nothing is received after some time
                  '   try to synchronize again

          IdleCount = 0
          Man_Receive_Init(PORTD)        ' Synchronize receiver
        end if
      loop until temp = $0B              ' End of message marker


  ' continues..
' ..continued

    ' If no errorFlag then write the message

      if errorFlag = false then
        LCD_Cmd(LCD_CLEAR)
        LetterCount = 0
        while LetterCount < 17        ' Message is 16 chars long
            inc(LetterCount)
            temp = Man_Receive(errorFlag)
            if errorFlag = false then
                LCD_Chr_CP(temp)
            else
                inc(errorCount)
                nop
            end if
        wend
       temp = Man_Receive(errorFlag)
       if temp <> $0E then
           inc(errorCount)
       end if
    end if
  wend
end.
```

Following code sends message in Manchester code:

```
program RF_TX

dim i as byte
dim s1 as string[20]

main:
  PORTB  = 0                     ' Initialize port
  TRISB  = %00001110
  ClearBit(INTCON, GIE)          ' Disable interrupts
  Man_Send_Init(PORTB)           ' Initialize manchester sender
  while TRUE
      Man_Send($0B)              ' Send start marker
      Delay_ms(100)              ' Wait for a while
      s1 = "mikroElektronika"
      for i = 1 to Length(s1)
         Man_Send(s1[i])         ' Send char
         Delay_ms(90)
      next i
      Man_Send($0E)              ' Send end marker
      Delay_ms(1000)
  wend
end.
```



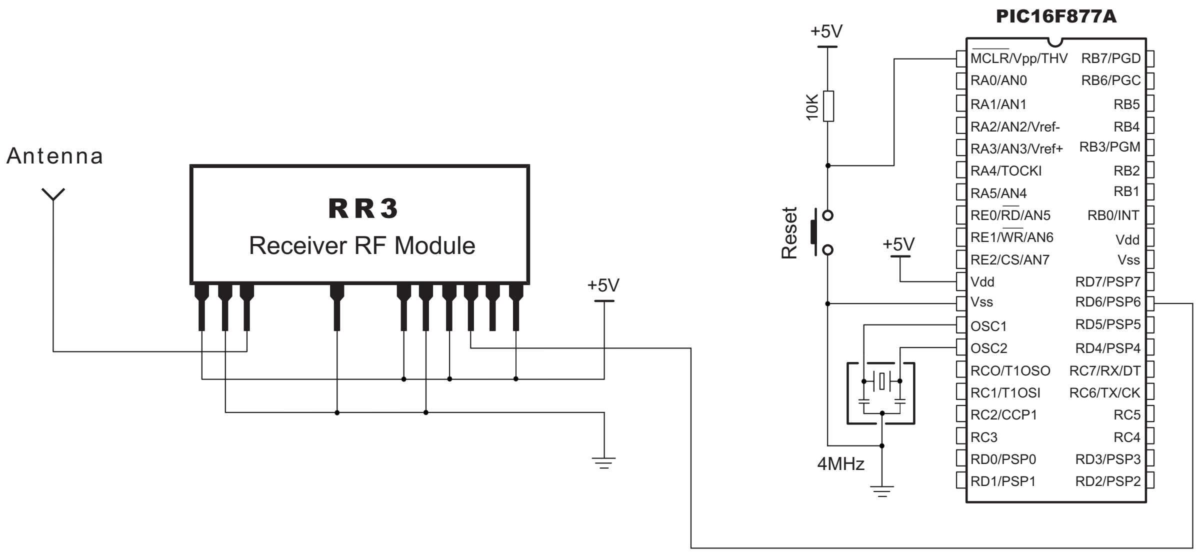


Figure: simple Transmitter and Receiver connection.

Numeric Formatting Routines

Numeric formatting routines convert byte, short, word, and integer to string, and can also convert decimal values to BCD and vice versa.

Routines

You can get text representation of numerical value by passing it to one of the routines listed below:

```
sub procedure ByteToStr(dim input as byte,  dim byref txt as char[6])
sub procedure WordToStr(dim input as word,  dim byref txt as char[6])
sub procedure ShortToStr(dim input as short, dim byref txt as char[6])
sub procedure IntToStr(dim input as integer, dim byref txt as char[6])
```

Parameter *input* represents numerical value of that should be converted to string; parameter *txt* is passed by address and it contains the result of conversion. All four procedures behave in similar fashion for appropriate input data type. (Parameter *txt* has to be of sufficient size to fit the converted string.)

Following routines convert decimal values to BCD (Binary Coded Decimal) and vice versa:

```
sub function Bcd2Dec(dim bcd_num as byte) as byte
sub function Dec2Bcd(dim dec_num as byte) as byte
sub function Bcd2Dec16(dim bcd_num as word) as word
sub function Dec2Bcd16(dim dec_num as word) as word
```

For instance, function Bcd2Dec converts 8-bit BCD numeral *bcd_num* to its decimal equivalent and returns the result as byte. Simple example:

```
..
dim a as byte
dim b as byte
begin
  a = 140
  b = Bcd2Dec(a)    ' b equals 224 now
end.
```

The following code demonstrates use of library procedure ShortToStr. Example prints the converted value to LCD display.

Example

This code demonstrates use of library procedure ShortToStr. Example prints the converted value to LCD display.

```
program num_format_test

dim txt as char[20]
dim i as short

main:
  PORTB = 0                        ' initial value for portb
  TRISB = 0                        ' designate portb as output
  LCD_Init(PORTB)                  ' initialize LCD on portb
  LCD_Cmd(LCD_CLEAR)               ' send command 'clear display'
  LCD_Cmd(LCD_CURSOR_OFF)          ' send command 'cursor off'
  txt = "mikroElektronika"         ' assign text

  LCD_Out(1,1,txt)                 ' print txt, 1st row, 1st col
  Delay_ms(1000)

  txt = "testing.."                ' write string to txt
  LCD_Out(2,1,txt)
  Delay_ms(1000)
                                   ' print txt, 2nd row, 1st col
  LCD_Cmd(LCD_CLEAR)
  for i = 127 to -111 step -1
     ShortToStr(i,txt)             ' convert variable i to string
     LCD_Out(2,1,txt)              ' print i (string value)
     Delay_ms(100)
     LCD_Cmd(LCD_CLEAR)
  next i
   LCD_Out(1,1,"The End")
end.
```

Trigonometry Library

Trigonometric functions take an angle (in degrees) as parameter of type word and return sine and cosine multiplied by 1000 and rounded up (as integer).

Routines

Functions implemented in the library are:

```
sub function sinE3(dim Angle as word) as integer
sub function cosE3(dim Angle as word) as integer
```

Functions take a word-type number which represents angle in degrees and return the sine of <*Angle*> as integer, multiplied by 1000 (1E3) and rounded up to nearest integer:

result = round_up(sin(Angle)*1000)

Thus, the range of the return values for these functions is from -1000 to 1000.

For example:

```
dim angle as word
dim result as integer

angle  = 45;
result = sinE3(angle)    ' result is 707
```

Note

Parameter <*Angle*> cannot be negative.

These functions are implemented as lookup tables. The maximum error obtained is ±1.

Example

The example demonstrates use of library functions sinE3 and cosE3. Example prints the deg, sine and cosine angle values on LCD display. The angle parameter can be altered by pushbuttons on PORTC.0 and PORTC.1.

```
program TestTrigon

dim angle as word
dim txtNum as char[6]
dim res as integer

main:
  TRISB = 0
  TRISC = $FF
  LCD_Init(PORTB)
  LCD_Cmd(LCD_CURSOR_OFF)

  angle = 45
  LCD_Out(1,1,"deg")
  LCD_Out(1,6,"sin")
  LCD_Out(1,12,"cos")

  while True
    LCD_Out(2,1,"  ")
    if (Button(PORTC, 0, 1, 1)=True) and (angle < 1000) then
      inc(angle)
    end if
    if (Button(PORTC, 1, 1, 1)=True) and (angle > 0) then
      dec(angle)
    end if

    WordToStr(angle,txtNum)  ' convert angle to text
    LCD_Out(2,1,txtNum)
    res = sinE3(angle)
    IntToStr(res, txtNum)    ' convert 1000*sin(angle) to text
    LCD_Out(2,6,txtNum)

    res = cosE3(angle)
    IntToStr(res, txtNum)    ' convert 1000*cos(angle) to text
    LCD_Out(2,12,txtNum)
    Delay_ms(100)
  wend
end.
```

Sound Library

mikroBasic provides a sound library which allows you to use sound signalization in your applications.

Routines

```
sub procedure Sound_Init(dim byref Port as byte, Pin as byte)
sub procedure Sound_Play(dim Period as byte,
                         dim Num_Of_Periods as word)
```

Procedure `Sound_Init` initializes sound engine at specified <*Port*> and <*Pin*>. Parameter <*Pin*> needs to be within range 0..7.

Procedure `Sound_Play` plays the sound at the specified port pin. <*Period_div_10*> is a sound period given in MCU cycles divided by ten, and generated sound lasts for a specified number of periods (<*Num_of_Periods*>).

For example, if you want to play sound of 1KHz: T = 1/f = 1ms = 1000 cycles @ 4MHz. This gives us our first parameter: 1000/10 = 100. We'll play 150 periods like this:

```
Sound_Play(100, 150)
```

Example

This is a simple demonstration of how to use sound library for playing tones on a piezo speaker. The code can be used with any MCU that has PORTB and ADC on PORTA. Sound frequencies in this example are generated by reading the value from ADC and using the lower byte of the result as base for T (f = 1/T).

```
program SoundADC

dim adcValue as byte

begin
  PORTB   = 0                         ' Clear PORTB
  TRISB   = 0                         ' PORTB is output
  INTCON  = 0                         ' Disable all interrupts
  ADCON1  = $82                       ' Configure VDD as Vref,
                                      '    and analog channels
  TRISA   = $FF                       ' PORTA is input

  Sound_Init(PORTB,2)                 ' Initialize sound on PORTB.RB2
  while true
      adcValue = ADC_Read(2)          ' Get lower byte from ADC
      Sound_Play(adcValue, 200)       ' Play the sound
    wend
end.
```

Utilities

mikroBasic provides a set of procedures and functions for faster development of your applications.

Routines

```
sub function Button(dim byref PORT as byte, dim PIN as byte,
                    dim Time as byte, dim Astate as byte) as byte
```

The Button function eliminates the influence of contact flickering due to the pressing of a button (debouncing).

Parameters *PORT* and *PIN* specify the location of the button; parameter *Time* represents the minimum time interval that pin must be in active state in order to return one; parameter *Astate* can be only zero or one, and it specifies if button is active on logical zero or logical one.

Example

This code demonstrates use of library function Button. Example reads the state on PORTB, pin 0, to which the button is connected. On transition from logical 1 to logical 0 which corresponds to release of a button, value on PORTD is inverted.

```
program test

dim byref oldstate as byte

main:
  PORTD = 255
  TRISD = 0
  TRISB = 255
  while true
    if Button(PORTB, 0, 1, 1) then
          oldstate = 255
    end if

    if  oldstate and  Button(PORTB, 0, 1, 0) then
         portD = 0
         oldstate = 0
    end if
  wend

end.
```

Contact us:

If you are experiencing problems with any of our products or you just want additional information, please let us know.

Technical Support for compiler

If you are experiencing any trouble with mikroBasic, please do not hesitate to contact us - it is in our mutual interest to solve these issues.

Discount for schools and universities

MikroElektronika offers a special discount for educational institutions. If you would like to purchase mikroBasic for purely educational purposes, please contact us.

Problems with transport or delivery

If you want to report a delay in delivery or any other problem concerning distribution of our products, please use the link given below.

Would you like to become mikroElektronika's distributor?

We in mikroElektronika are looking forward to new partnerships. If you would like to help us by becoming distributor of our products, please let us know.

Other

If you have any other question, comment or a business proposal, please contact us:

MikroElektronika magazine
Admirala Geprata 1B
11000 Belgrade
EUROPE

Phone: + 381 (11) 30 66 377, + 381 (11) 30 66 378
Fax: + 381 (11) 30 66 379
E-mail: *office@mikroelektronika.co.yu*
Web: *www.mikroelektronika.co.yu*