mikroElektronika
Development tools - Books - Compilers
www.mikroe.com

BASIC COMPILER FOR MICROCHIP PIC MICROCONTROLLERS

mikroBASIC

Making it simple

User's manual

mikro ICD
IN-CIRCUIT DEBUGGER

SUPPORTED from V5.0

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.

Reader's note

DISCLAIMER:

mikroBasic and this manual are owned by mikroElektronika and are protected by copyright law and international copyright treaty. Therefore, you should treat this manual like any other copyrighted material (e.g., a book). The manual and the compiler may not be copied, partially or as a whole without the written consent from the mikroElektronika. The PDF-edition of the manual can be printed for private or local use, but not for distribution. Modifying the manual or the compiler is strictly prohibited.

HIGH RISK ACTIVITIES

The mikroBasic compiler is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of the Software could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities"). mikroElektronika and its suppliers specifically disclaim any express or implied warranty of fitness for High Risk Activities.

LICENSE AGREEMENT:

By using the mikroBasic compiler, you agree to the terms of this agreement. Only one person may use licensed version of mikroBasic compiler at a time.

This manual covers mikroBasic version 5.0.0.2 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@mikroe.com. Please include the following information in your bug report:

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

CONTACT US:

mikroElektronika
Voice: + 381 (11) 30 66 377, + 381 (11) 30 66 378
Fax: + 381 (11) 30 66 379
Web: www.mikroe.com
E-mail: office@mikroe.com

PIC, PICmicro and MPLAB is a Registered trademark of Microchip company. Windows is a Registered trademark of Microsoft Corp. All other trade and/or services marks are the property of the respective owners.

mikroBASIC User's manual

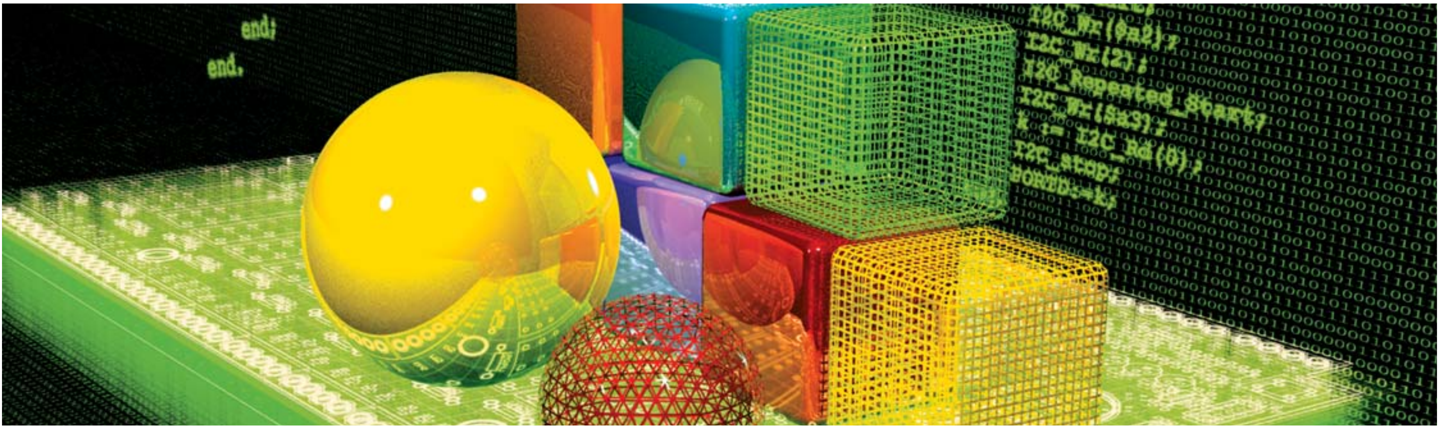
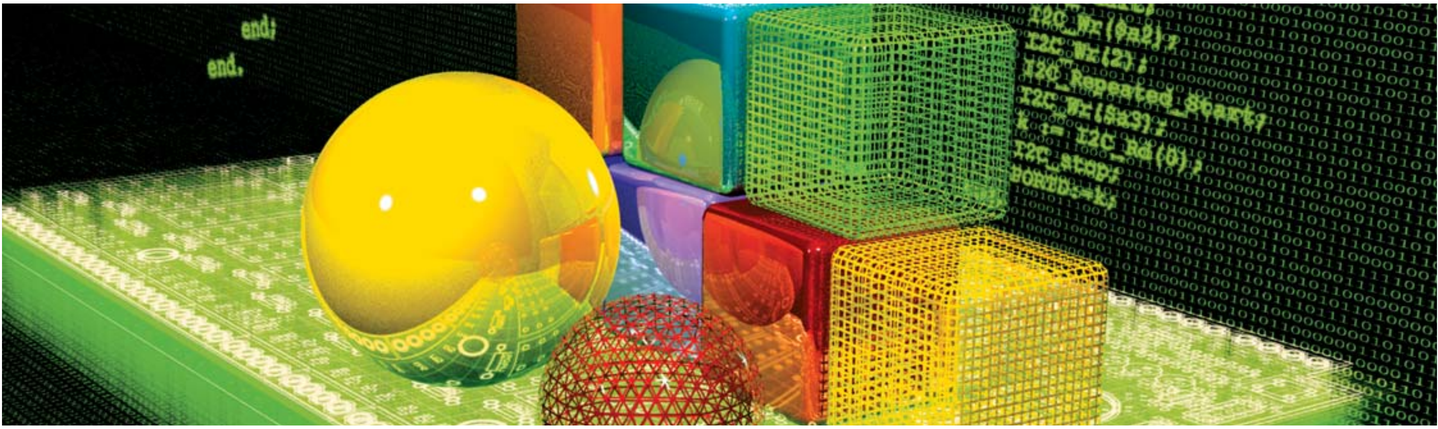


Table of Contents

CHAPTER 4: mikroBasic Libraries 99

CHAPTER 1

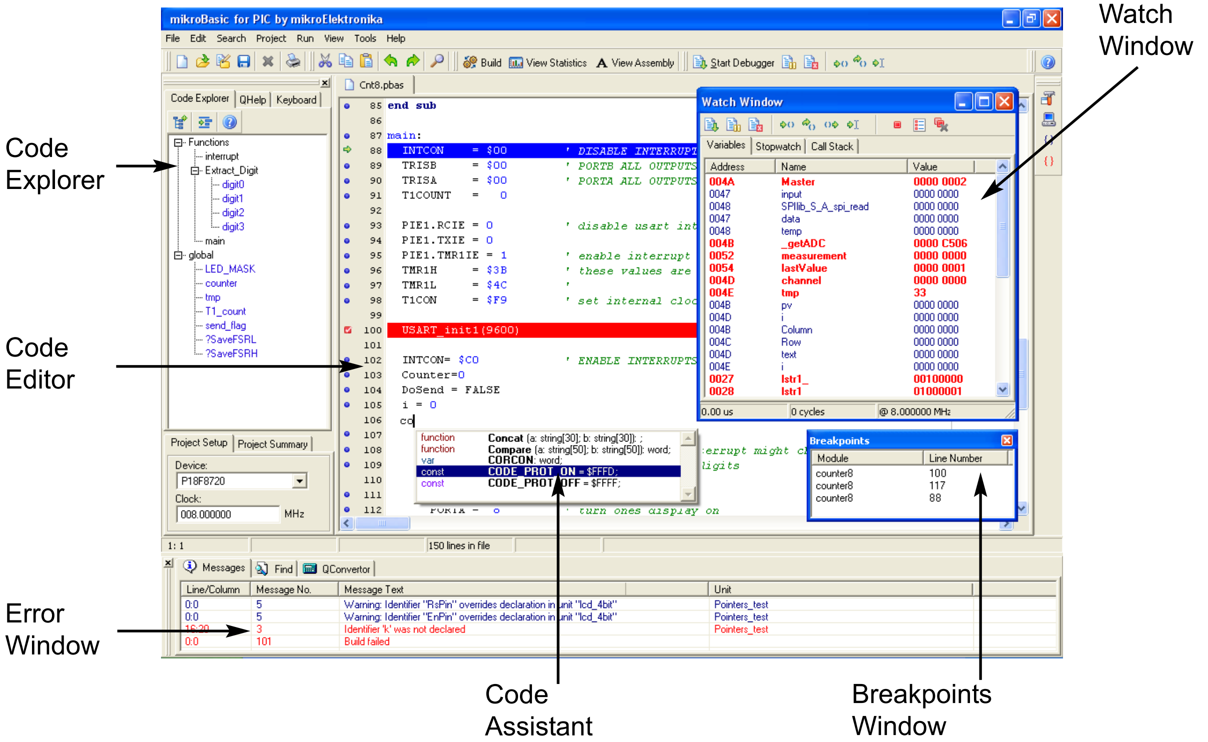


mikroBasic IDE

QUICK OVERVIEW

mikroBasic is a powerful, feature rich development tool for PIC microcontrollers. It is designed to provide the customer with the easiest possible solution for developing applications for embedded systems, without compromising performance or control.

Highly advanced IDE, broad set of hardware libraries, comprehensive documentation, and plenty of ready to run examples should be more than enough to get you started in programming microcontrollers.



```
end sub

main:
  INTCON     = $00           ' DISABLE INTERRUPT
  TRISB      = $00           ' PORTB ALL OUTPUTS
  TRISA      = $00           ' PORTA ALL OUTPUTS
  T1COUNT    =   0

  PIE1.RCIE = 0              ' disable usart int
  PIE1.TXIE = 0
  PIE1.TMR1IE = 1            ' enable interrupt
  TMR1H      = $3B           ' these values are
  TMR1L      = $4C           '
  T1CON      = $F9           ' set internal cloc

  USART_init1(9600)

  INTCON= $C0                ' ENABLE INTERRUPTS
  Counter=0
  DoSend = FALSE
  i = 0
 co
```

function	Concat (a: string[30]; b: string[30]): ;
function	Compare (a: string[50]; b: string[50]): word;
var	CORCON: word;
const	CODE_PROT_ON = $FFFD;
const	CODE_PROT_OFF = $FFFF;

Address	Name	Value
004A	Master	0000 0002
0047	input	0000 0000
0048	SPIlib_S_A_spi_read	0000 0000
0047	data	0000 0000
0048	temp	0000 0000
004B	_getADC	0000 C506
0052	measurement	0000 0000
0054	lastValue	0000 0001
004D	channel	0000 0000
004E	tmp	33
004B	pv	0000 0000
004D	i	0000 0000
004B	Column	0000 0000
004C	Row	0000 0000
004D	text	0000 0000
004E	i	0000 0000
0027	lstr1_	00100000
0028	lstr1	01000001

0.00 us | 0 cycles | @ 8.000000 MHz

Module	Line Number
counter8	100
counter8	117
counter8	88

Line/Column	Message No.	Message Text	Unit
0:0	5	Warning: Identifier "RsPin" overrides declaration in unit "lcd_4bit"	Pointers_test
0:0	5	Warning: Identifier "EnPin" overrides declaration in unit "lcd_4bit"	Pointers_test
16:20	3	Identifier 'k' was not declared	Pointers_test
0:0	101	Build failed	

mikroBasic allows you to quickly develop and deploy complex applications:

- Write your BASIC source code using the highly advanced Code Editor

- Use the included mikroBasic libraries to dramatically speed up the development: data acquisition, memory, displays, conversions, communications…

- Monitor your program structure, variables, and functions in the Code Explorer. Generate commented, human-readable assembly, and standard HEX compatible with all programmers.

- Inspect program flow and debug executable logic with the integrated Debugger. Get detailed reports and graphs on code statistics, assembly listing, calling tree…

- We have provided plenty of examples for you to expand, develop, and use as building bricks in your projects.

CODE EDITOR

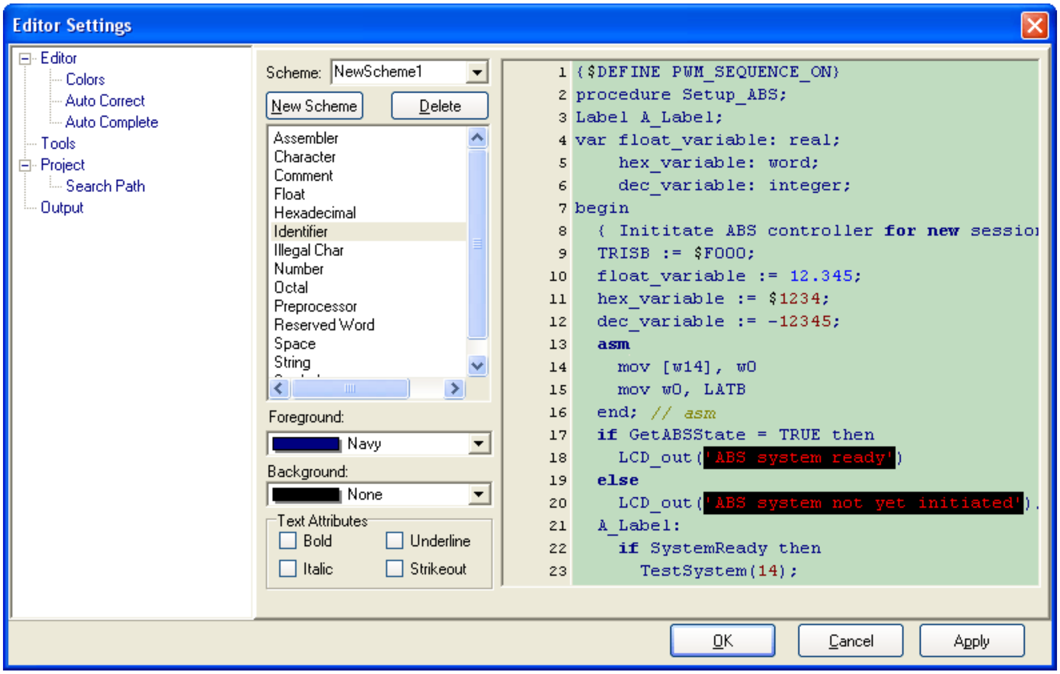
The Code Editor is advanced text editor fashioned to satisfy the needs of professionals. General code editing is same as working with any standard text-editor, including familiar Copy, Paste, and Undo actions, common for Windows environment.

Advanced Editor features include:

- Adjustable Syntax Highlighting
- Code Assistant
- Parameter Assistant
- Code Templates
- Auto Correct for common typos
- Bookmarks and Goto Line

You can customize these options from Editor Settings dialog. To access the settings, click Tools > Options from the drop-down menu, or click the Tools icon.

Tools Icon.



Scheme: NewScheme1

Foreground: Navy

Background: None

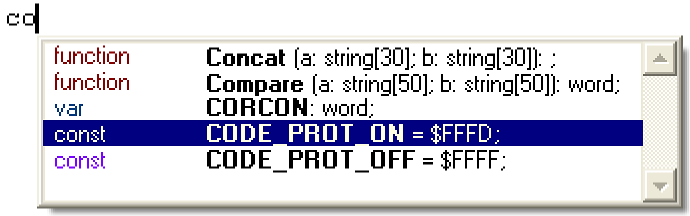
☐ Bold ☐ Underline

☐ Italic ☐ Strikeout

```
{$DEFINE PWM_SEQUENCE_ON}
procedure Setup_ABS;
Label A_Label;
var float_variable: real;
    hex_variable: word;
    dec_variable: integer;
begin
  { Inititate ABS controller for new sessio
  TRISB := $F000;
  float_variable := 12.345;
  hex_variable := $1234;
  dec_variable := -12345;
  asm
    mov [w14], w0
    mov w0, LATB
  end; // asm
  if GetABSState = TRUE then
    LCD_out('ABS system ready')
  else
    LCD_out('ABS system not yet initiated')
  A_Label:
    if SystemReady then
      TestSystem(14);
```

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 the keyboard arrows and Enter.



```
co
function    Concat (a: string[30]; b: string[30]): ;
function    Compare (a: string[50]; b: string[50]): word;
var         CORCON: word;
const       CODE_PROT_ON = $FFFD;
const       CODE_PROT_OFF = $FFFF;
```

Parameter Assistant [CTRL+SHIFT+SPACE]

The 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.

```
dim channel as byte
ADC_Read(
```

Code Template [CTR+J]

You can insert the Code Template by typing the name of the template (for instance, *whileb*), 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 the drop-down menu, or click the Tools Icon from the Settings Toolbar, and then select the Auto Complete Tab. Here you can enter the appropriate keyword, description, and code of your template.

Auto Correct

The Auto Correct feature corrects some common typing mistakes. To access the list of recognized typos, select Tools > Options from the 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.

Comment/Uncomment

The Code Editor allows you to comment or uncomment selected block of code by a simple click of a mouse, using the Comment/Uncomment icons from the Code Toolbar.

Bookmarks

Bookmarks make navigation through large code easier.

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

Goto Line

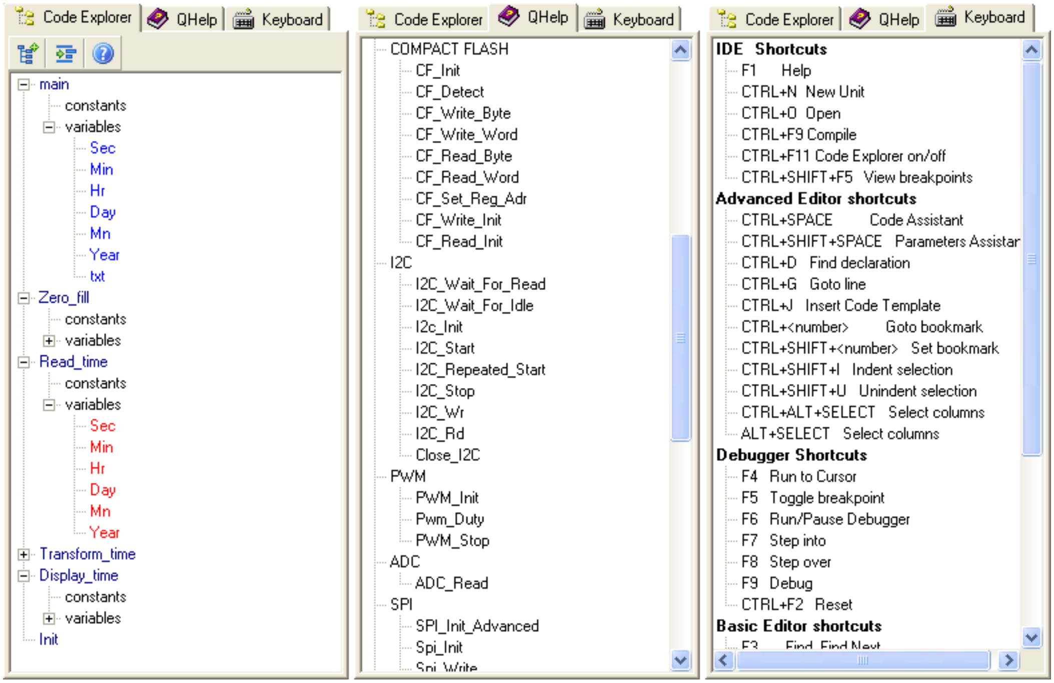
Goto Line option makes navigation through large code easier. Select Search > Goto Line from the drop-down menu, or use the shortcut CTRL+G.

CODE EXPLORER

The 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 the Code Explorer. QHelp Tab lists all the available built-in and library functions, for a quick reference. Double-clicking a routine in the QHelp Tab opens the relevant Help topic. Keyboard Tab lists all the available keyboard shortcuts in mikroBasic.



- main
  - constants
  - variables
    - Sec
    - Min
    - Hr
    - Day
    - Mn
    - Year
    - txt
- Zero_fill
  - constants
  - variables
- Read_time
  - constants
  - variables
    - Sec
    - Min
    - Hr
    - Day
    - Mn
    - Year
- Transform_time
- Display_time
  - constants
  - variables
- Init

- COMPACT FLASH
  - CF_Init
  - CF_Detect
  - CF_Write_Byte
  - CF_Write_Word
  - CF_Read_Byte
  - CF_Read_Word
  - CF_Set_Reg_Adr
  - CF_Write_Init
  - CF_Read_Init
- I2C
  - I2C_Wait_For_Read
  - I2C_Wait_For_Idle
  - I2c_Init
  - I2C_Start
  - I2C_Repeated_Start
  - I2C_Stop
  - I2C_Wr
  - I2C_Rd
  - Close_I2C
- PWM
  - PWM_Init
  - Pwm_Duty
  - PWM_Stop
- ADC
  - ADC_Read
- SPI
  - SPI_Init_Advanced
  - Spi_Init
  - Spi_Write

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 Assistar
- 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
- 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

DEBUGGER

Start Debugger.

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

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.

After you have successfully compiled your project, you can run the Debugger by selecting Run > Debug from the 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.

Pause Debugger.

Debug [F9]
Start the Debugger.

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

Step Into.

Step Into [F7]
Execute the current BASIC (single– or multi–cycle) instruction, then halt. If the instruction is a routine call, enter the routine and halt at the first instruction following the call.

Step Over.

Step Over [F8]
Execute the current BASIC (single– or multi–cycle) instruction, then halt. If the instruction is a routine call, skip it and halt at the first instruction following the call.

Step Out.

Step Out [Ctrl+F8]
Execute the current BASIC (single– or multi–cycle) instruction, then halt. If the instruction is within a routine, execute the instruction and halt at the first instruction following the call.

Run to Cursor.

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

Jump to Interrupt.

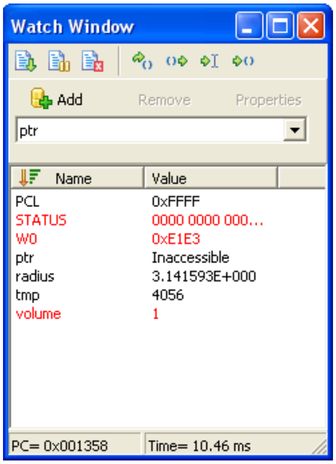
Jump to Interrupt [F2]
Jump to address `$04` for PIC12/16 or to address `$08` for PIC18 and execute the procedure located at that address.

Toggle Breakpoint.

Toggle Breakpoint [F5]
Toggle breakpoint at the current cursor position. To view all the breakpoints, select Run > View Breakpoints from the drop-down menu. Double clicking an item in window list locates the breakpoint.

Watch Window

Debugger Watch Window is the main Debugger window which allows you to monitor program items while running your program. To show the Watch Window, select View > Debug Windows > Watch Window from the drop-down menu.



ptr

Name	Value
PCL	0xFFFF
STATUS	0000 0000 000...
W0	0xE1E3
ptr	Inaccessible
radius	3.141593E+000
tmp	4056
volume	1

PC= 0x001358
Time= 10.46 ms

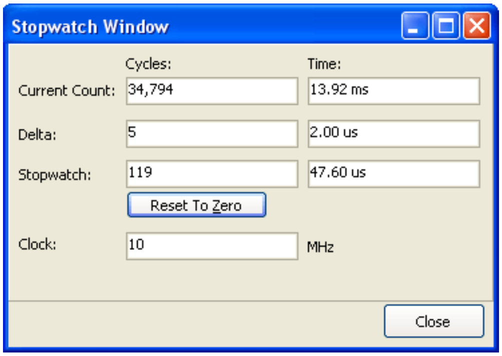
The Watch Window displays variables and registers of PIC, with their addresses and values. Values are updated as you go through the simulation. Use the drop-down menu to add and remove the items that you want to monitor. Recently changed items are colored red.

Double clicking an item opens the Edit Value window in which you can assign a new value to the selected variable/register. Also, you can change view to binary, hex, char, or decimal for the selected item.

Stopwatch Window

Debugger Stopwatch Window is available from the drop-down menu,
View > Debug Windows > Stopwatch.

The Stopwatch Window displays the current count of cycles/time since the last Debugger action. Stopwatch measures the execution time (number of cycles) from the moment Debugger is started, and can be reset at any time. Delta represents the number of cycles between the previous instruction line (line where the Debugger action was performed) and the active instruction line (where the Debugger action landed).



	Cycles:	Time:
Current Count:	34,794	13.92 ms
Delta:	5	2.00 us
Stopwatch:	119	47.60 us
Clock:	10	MHz

Note: You can change the clock in the Stopwatch Window; this will recalculate values for the newly specified frequency. Changing the clock in the Stopwatch Window does not affect the actual project settings – it only provides a simulation.

View RAM Window

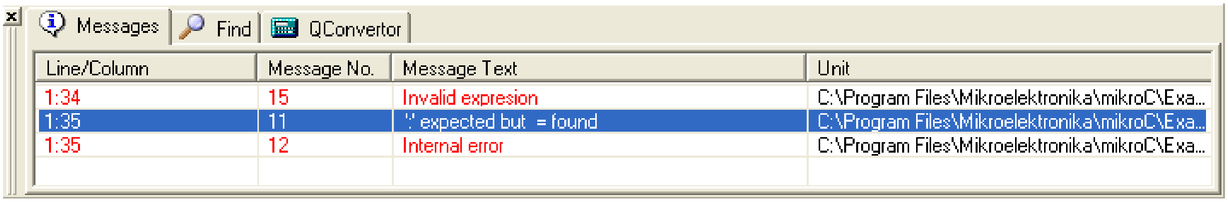
Debugger View RAM Window is available from the drop-down menu,
View > Debug Windows > View RAM.

The View RAM Window displays the map of PIC's RAM, with recently changed items colored red. You can change value of any field by double-clicking it.

ERROR WINDOW

In case that errors were encountered during compiling, compiler will report them and won't generate a hex file. The 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. The 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	Unit
1:34	15	Invalid expresion	C:\Program Files\Mikroelektronika\mikroC\Exa...
1:35	11	':' expected but = found	C:\Program Files\Mikroelektronika\mikroC\Exa...
1:35	12	Internal error	C:\Program Files\Mikroelektronika\mikroC\Exa...

Double click the message line in the Error Window to highlight the line where the error was encountered.

Consult the Error Messages for more information about errors recognized by the compiler.

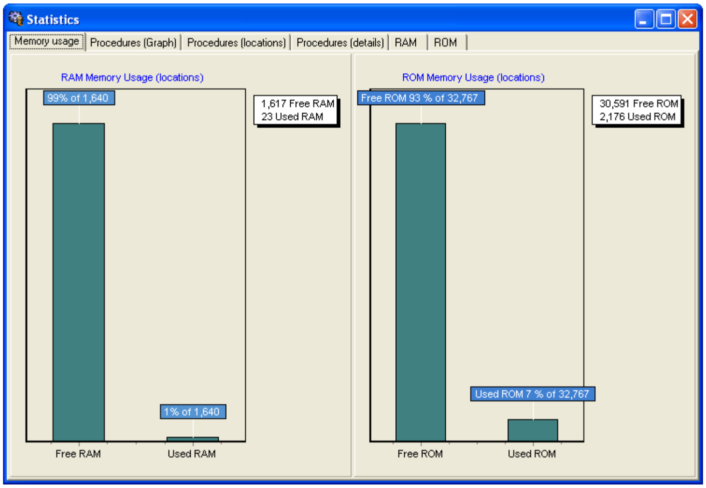
STATISTICS

Statistics Icon.

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

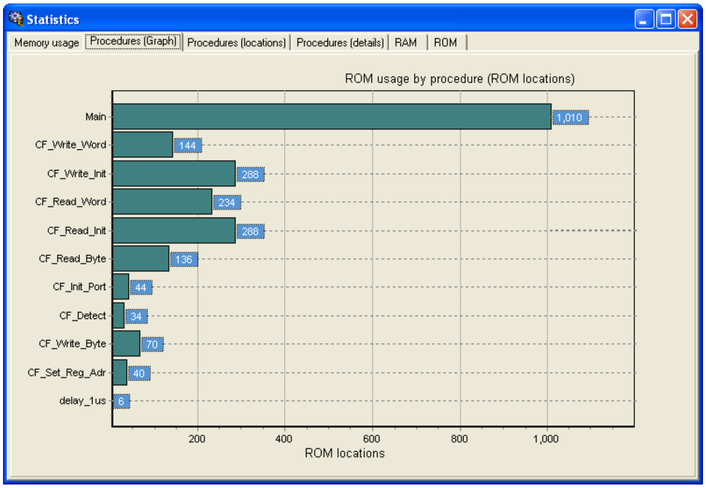
Memory Usage Window

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



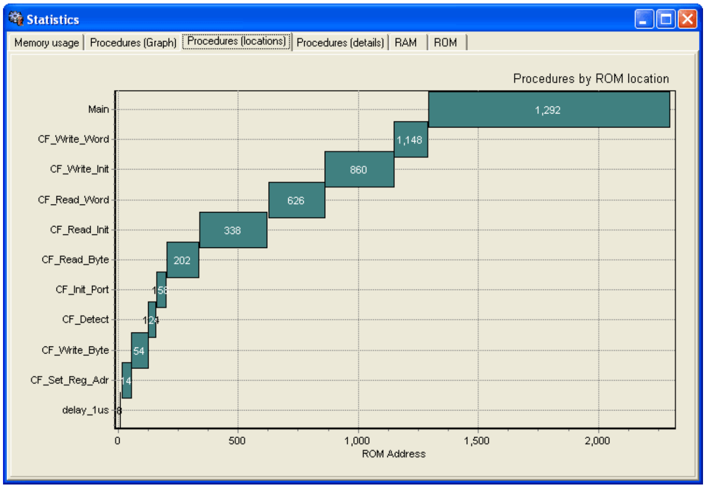
Procedures (Graph) Window

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



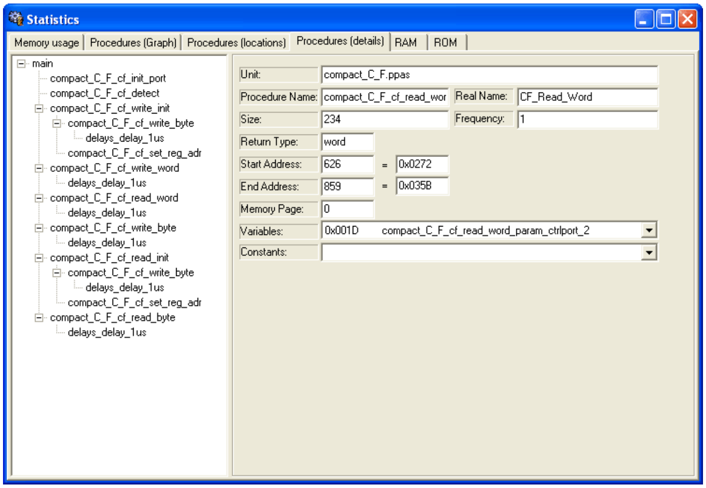
Procedures (Locations) Window

Displays how functions are distributd in microcontroller's memory.



Procedures (Details) Window

Displays complete call tree, along with details for each procedure and function:



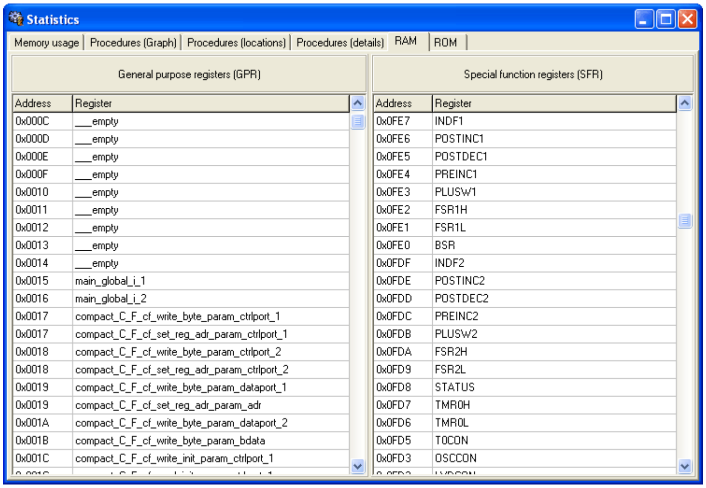
- main
  - compact_C_F_cf_init_port
  - compact_C_F_cf_detect
  - compact_C_F_cf_write_init
    - compact_C_F_cf_write_byte
      - delays_delay_1us
    - compact_C_F_cf_set_reg_adr
  - compact_C_F_cf_write_word
    - delays_delay_1us
  - compact_C_F_cf_read_word
    - delays_delay_1us
  - compact_C_F_cf_write_byte
    - 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:

size, start and end address, calling frequency, return type, etc.

RAM Window

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



General purpose registers (GPR)

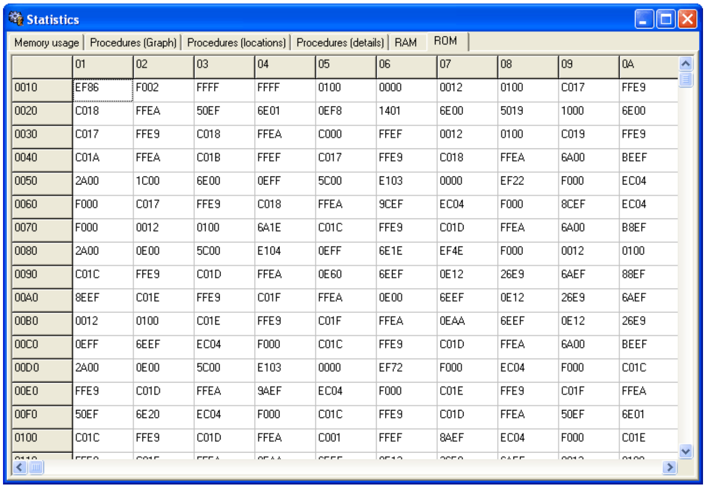
Address	Register
0x000C	___empty
0x000D	___empty
0x000E	___empty
0x000F	___empty
0x0010	___empty
0x0011	___empty
0x0012	___empty
0x0013	___empty
0x0014	___empty
0x0015	main_global_i_1
0x0016	main_global_i_2
0x0017	compact_C_F_cf_write_byte_param_ctrlport_1
0x0017	compact_C_F_cf_set_reg_adr_param_ctrlport_1
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.

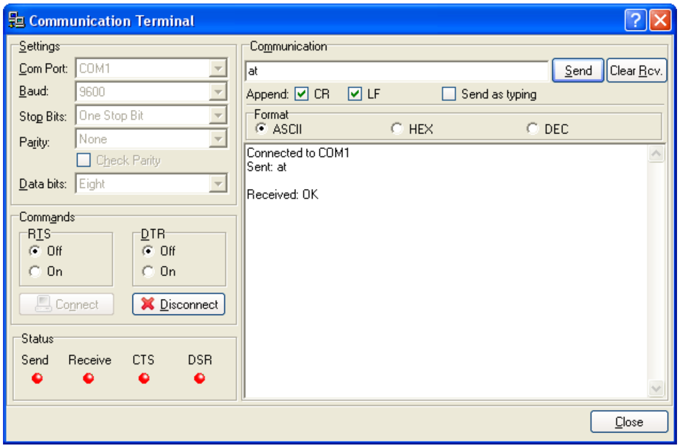


	01	02	03	04	05	06	07	08	09	0A
0010	EF86	F002	FFFF	FFFF	0100	0000	0012	0100	C017	FFE9
0020	C018	FFEA	50EF	6E01	0EF8	1401	6E00	5019	1000	6E00
0030	C017	FFE9	C018	FFEA	C000	FFEF	0012	0100	C019	FFE9
0040	C01A	FFEA	C01B	FFEF	C017	FFE9	C018	FFEA	6A00	BEEF
0050	2A00	1C00	6E00	0EFF	5C00	E103	0000	EF22	F000	EC04
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 the USART (Universal Synchronous Asynchronous Receiver Transmitter) communication terminal for RS232 communication. You can launch it from the drop-down menu Tools > Terminal or by clicking the Terminal icon.



Com Port: COM1
Baud: 9600
Stop Bits: One Stop Bit
Parity: None
☐ Check Parity
Data bits: Eight

RTS: ● Off ○ On
DTR: ● Off ○ On

at

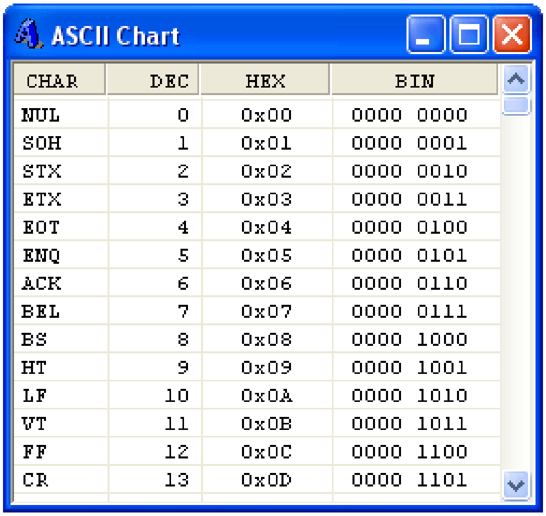
Append: ☑ CR ☑ LF ☐ Send as typing

Format: ● ASCII ○ HEX ○ DEC

```
Connected to COM1
Sent: at

Received: OK
```

ASCII Chart

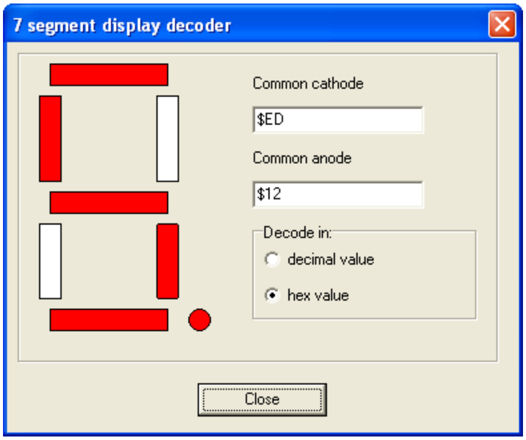
ASCII Chart is a handy tool, particularly useful when working with LCD display. You can launch it from the drop-down menu Tools > ASCII chart.



CHAR	DEC	HEX	BIN
NUL	0	0x00	0000 0000
SOH	1	0x01	0000 0001
STX	2	0x02	0000 0010
ETX	3	0x03	0000 0011
EOT	4	0x04	0000 0100
ENQ	5	0x05	0000 0101
ACK	6	0x06	0000 0110
BEL	7	0x07	0000 0111
BS	8	0x08	0000 1000
HT	9	0x09	0000 1001
LF	10	0x0A	0000 1010
VT	11	0x0B	0000 1011
FF	12	0x0C	0000 1100
CR	13	0x0D	0000 1101

7 Segment Display Decoder

The 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 the drop-down menu Tools > 7 Segment Display.



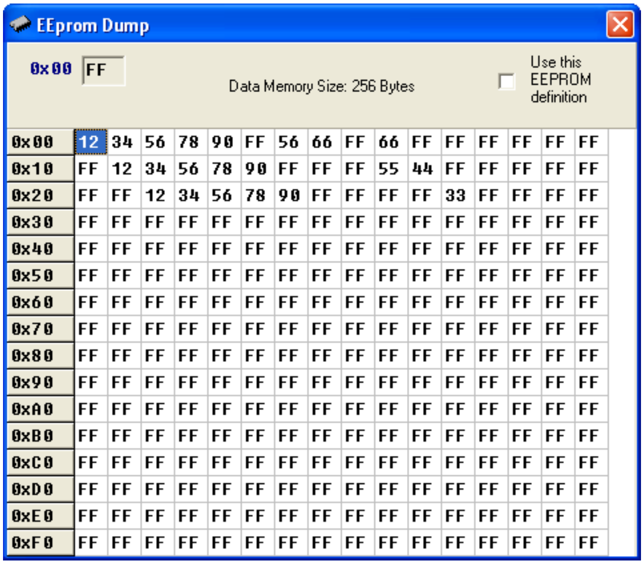
Common cathode: $ED

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

mikroBootloader

mikroBootloader can be used only with PICmicros that support flash write.

1. Load the PIC with the appropriate hex file using the conventional programming techniques (e.g. for PIC16F877A use p16f877a.hex).
2. Start mikroBootloader from the drop-down menu Tools > Bootoader.
3. Click on Setup Port and select the COM port that will be used. Make sure that BAUD is set to 9600 Kpbs.
4. Click on Open File and select the HEX file you would like to upload.
5. Since the bootcode in the PIC only gives the computer 4-5 sec to connect, you should reset the PIC and then click on the Connect button within 4-5 seconds.
6. The last line in then history window should now read “Connected”.
7. To start the upload, just click on the Start Bootloader button.
8. Your program will written to the PIC flash. Bootloader will report an errors that may occur.
9. Reset your PIC and start to execute.

The boot code gives the computer 5 seconds to get connected to it. If not, it starts running the existing user code. If there is a new user code to be downloaded, the boot code receives and writes the data into program memory.

The more common features a bootloader may have are listed below:

- Code at the Reset location.
- Code elsewhere in a small area of memory.
- Checks to see if the user wants new user code to be loaded.
- Starts execution of the user code if no new user code is to be loaded.
- Receives new user code via a communication channel if code is to be loaded.
- Programs the new user code into memory.

Integrating User Code and Boot Code

The boot code almost always uses the Reset location and some additional program memory. It is a simple piece of code that does not need to use interrupts; therefore, the user code can use the normal interrupt vector at 0x0004. The boot code must avoid using the interrupt vector, so it should have a program branch in the address range 0x0000 to 0x0003. The boot code must be programmed into memory using conventional programming techniques, and the configuration bits must be programmed at this time. The boot code is unable to access the configuration bits, since they are not mapped into the program memory space.

KEYBOARD SHORTCUTS

Below is the complete list of keyboard shortcuts available in mikroBasic IDE. You can also view keyboard shortcuts in the Code Explorer, tab Keyboard.

IDE Shortcuts

F1	Help
CTRL+SHIFT+E	Edit Project
Ctrl+N	New Module
Ctrl+O	Open
SHIFT+F9	Build all
Ctrl+F9	Compile
F11	Program
F12	Options
CTRL+F11	Compile and program
Ctrl+Shift+F5	View breakpoints

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 module
CTRL+SHIFT+S	Save As
CTRL+V	Paste
CTRL+X	Cut
CTRL+Y	Redo
CTRL+Z	Undo

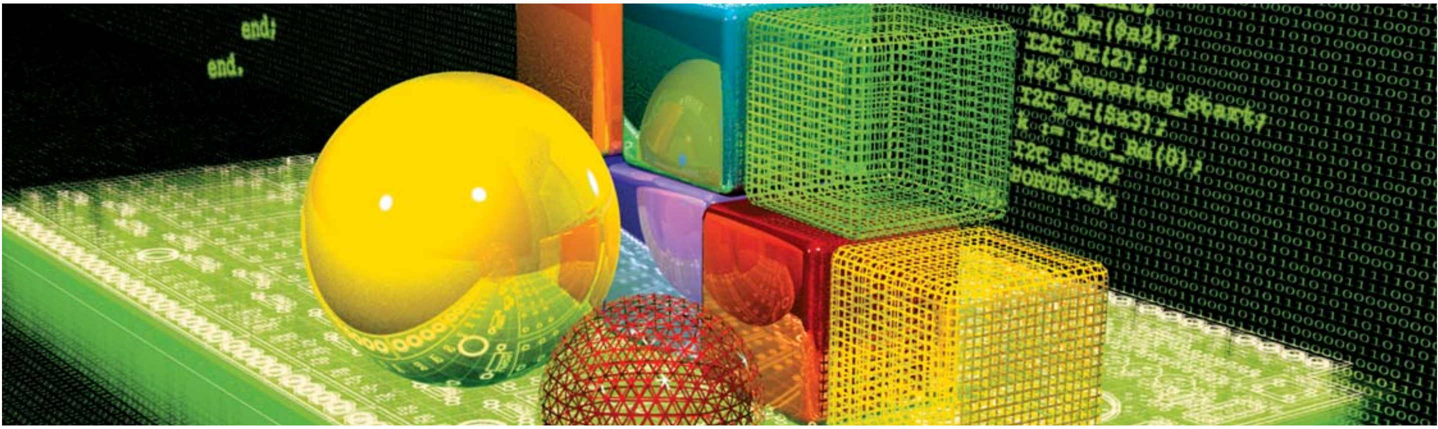
Advanced Editor shortcuts

Ctrl+Space	Code Assistant
Ctrl+Shift+Space	Parameter Assistant
Ctrl+D	Find declaration
CTRL+E	Incremental search
Ctrl+G	Goto line
Ctrl+J	Insert Code Template
Ctrl+L	Procedures list
CTRL+/	Toggle line comment
Ctrl+number	Goto bookmark
Ctrl+Shift+number	Set bookmark
Ctrl+Shift+I	Indent selection
Ctrl+Shift+U	Unindent selection
Alt+Select	Select columns
Tab	Indent selection
Shift+Tab	Unindent selection
Ctrl+Alt+Select	Select columns
Alt+F3	Find in files

Debugger Shortcuts

F4	Run to Cursor
F5	Toggle Breakpoint
F6	Run/Pause Debugger
F7	Step into
F8	Step over
Ctrl+F8	Step out
F9	Debug
F2	Jump to Interrupt
Ctrl+F2	Reset
Ctrl+F5	Add to watch

CHAPTER 2



Building Applications

Creating applications in mikroBasic is easy and intuitive. Project Wizard allows you to set up your project in just few clicks: name your application, select chip, set flags, and get going.

mikroBasic allows you to distribute your projects in as many modules as you find appropriate. You can then share your mikroCompiled Libraries (`.mcl` files) with other developers without disclosing the source code. The best part is that you can use `.mcl` bundles created by mikroPascal or mikroC!

PROJECTS

mikroBasic organizes applications into *projects*, consisting of a single project file (extension `.pbp`) and one or more source files (extension `.pbas`). You can compile source files only if they are part of a project.

Project file carries the following information:

- project name and optional description
- target device
- device flags (config word) and device clock
- list of project source files with paths

New Project.

New Project

The easiest way to create project is by means of New Project Wizard, drop-down menu Project > New Project. Just fill the dialog with desired values (project name and description, location, device, clock, config word) and mikroBasic will create the appropriate project file.

Also, an empty source file named after the project will be created by default. mikroBasic does not require you to have source file named same as the project, it's just a matter of convenience.

Edit Project.

Editing Project

Later, you can change project settings from the drop-down menu Project > Edit. You can add or remove source files from project, rename the project, modify its description, change chip, clock, config word, etc.

To delete a project, simply delete the folder in which the project file is stored.

SOURCE FILES

Source files containing BASIC code should have the extension `.pbas`. List of source files relevant for the application is stored in project file with extension .pbp, along with other project information. You can compile source files only if they are part of a project.

Search Paths

You can specify your own custom search paths. This can be configured by selecting Tools > Options from the drop-down menu and Compiler > Search Paths.

When including source files with the `include` clause, mikroBasic will look for the file in following locations, in this particular order:

1. mikroBasic installation folder > "defs" folder
2. mikroBasic installation folder > "uses" folder
3. your custom search paths
4. the project folder (folder which contains the project file `.pbp`)

Managing Source Files

Creating a new source file

New File.

To create a new source file, do the following:

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

If you have used New Project Wizard, an empty source file, named after the project with extension `.pbas`, is created automatically. mikroBasic does not require you to have the source file named same as the project, it's just a matter of convenience.

Open File.

Opening an Existing File

Select File > Open from the 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.

Print File.

Printing an Open File

Make sure that window containing the file you want to print is the active window. Select File > Print from the 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.

Save File.

Saving File

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

Save File As.

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 the 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. Click the Save button.

Close File.

Closing a File

Make sure that tab containing the file you want to close is the active tab. Select File > Close from the 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.

COMPILATION

Build Icon.

When you have created the project and written the source code, you will want to compile it. Select Project > Build from the drop-down menu, or click the Build Icon, or simply hit CTRL+F9.

Progress bar will appear to inform you about the status of compiling. If there are errors, you will be notified in the Error Window. If no errors are encountered, mikroBasic will generate output files.

Output Files

Upon successful compilation, mikroBasic will generate output files in the project folder (folder which contains the project file `.pbp`). Output files are summarized below:

Intel HEX file (`.hex`)
Intel style hex records. Use this file to program PIC MCU.

Binary mikro Compiled Library (`.mcl`)
Binary distribution of application that can be included in other projects.

List File (`.lst`)
Overview of PIC memory allotment: instruction addresses, registers, routines, etc.

Assembler File (`.asm`)
Human readable assembly with symbolic names, extracted from the List File.

Assembly View

View Assembly Icon.

After compiling your program in mikroBasic, you can click View Assembly Icon or select Project > View Assembly from the drop-down menu to review generated assembly code (`.asm` file) 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 (`.lst`).

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

ERROR MESSAGES

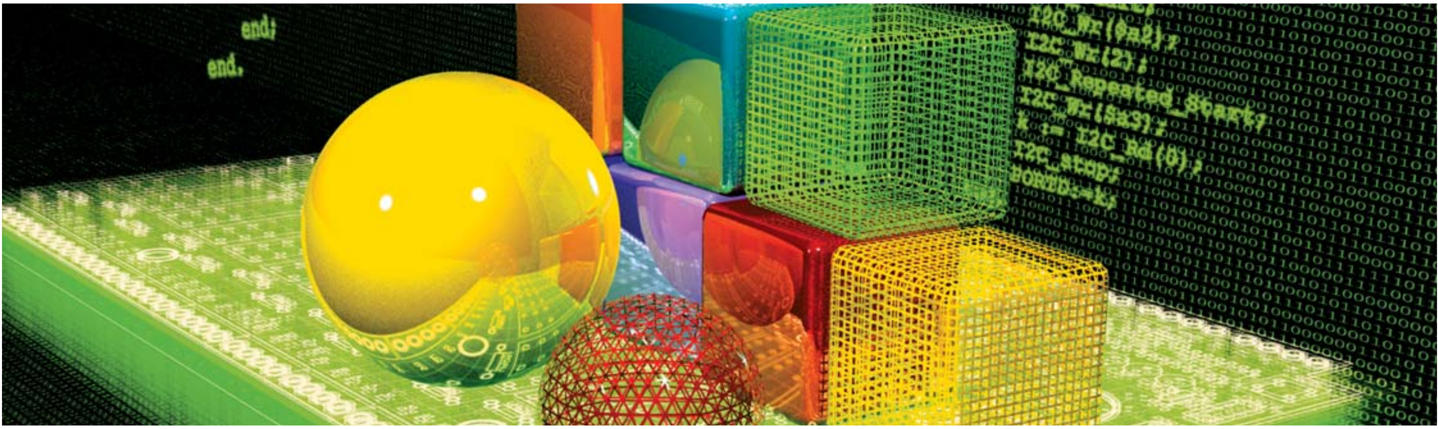
Error Messages

Message	Message Number
Error: "%s" is not a valid identifier	1
Error: Unknown type "%s"	2
Error: Identifier "%s" was not declared	3
Error: Expected "%s" but "%s" found	4
Error: Argument is out of range	5
Error: Syntax error in additive expression	6
Error: File "%s" not found	7
Error: Invalid command "%s"	8
Error: Not enough parameters	9
Error: Too many parameters	10
Error: Too many characters	11
Error: Actual and formal parameters must be identical	12
Error: Invalid ASM instruction: "%s"	13
Error: Identifier "%s" has been already declared	14
Error: Syntax error in multiplicative expression	15
Error: Definition file for "%s" is corrupted	16

Hint and Warning Messages

Message	Message Number
Hint: Variable "%s" has been declared, but was not used	1
Warning: Variable "%s" is not initialized	2
Warning: Return value of the function "%s" is not defined	3
Hint: Constant "%s" has been declared, but was not used	4
Warning: Identifier "%s" overrides declaration in unit "%s"	5

CHAPTER 3



mikroBasic Language Reference

Why BASIC in the first place? The answer is simple: it is legible, easy-to-learn, structured 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.

PIC SPECIFICS

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

Types Efficiency

First of all, you should know that PIC's ALU, which performs arithmetic operations, is optimized for working with bytes. Although mikroBasic is capable of handling very complex data types, PIC may choke on them, especially if you are working on some of the older models. This can dramatically increase the time needed for performing even simple operations. Universal advice is to use the smallest possible type in every situation. It applies to all programming in general, and doubly so with microcontrollers.

When it comes down to calculus, not all PICmicros are of equal performance. For example, PIC16 family lacks hardware resources to multiply two bytes, so it is compensated by a software algorithm. On the other hand, PIC18 family has HW multiplier, and as a result, multiplication works considerably faster.

Nested Calls Limitations

Nested call represents a function call within function body, either to itself (recursive calls) or to another function. Recursive calls, as form of cross-calling, are unsupported by mikroBasic due to the PIC's stack and memory limitations.

mikroBasic limits the number of non-recursive nested calls to:

- 8 calls for PIC12 family,
- 8 calls for PIC16 family,
- 31 calls for PIC18 family.

The number of allowed nested calls decreases by one if you use any of the following operators in the code: `*` `/` `%`. It further decreases by one if you use interrupt in the program. If the allowed number of nested calls is exceeded, compiler will report stack overflow error.

PIC16 Only Specifics

Breaking Through Pages

In applications targeted at PIC16, no single routine should exceed one page (2,000 instructions). If routine does not fit within one page, linker will report an error. When confront with this problem, maybe you should rethink the design of your application – try breaking the particular routine into several chunks, etc.

Limits of Indirect Approach Through FSR

Pointers with PIC16 are “near”: they carry only the lower 8 bits of the address. Compiler will automatically clear the 9th bit upon startup, so that pointers will refer to banks 0 and 1. To access the objects in banks 3 or 4 via pointer, user should manually set the IRP, and restore it to zero after the operation.

Note: It is very important to take care of the IRP properly, if you plan to follow this approach. If you find this method to be inappropriate with too many variables, you might consider upgrading to PIC18.

Note: If you have many variables in the code, try rearranging them with linker directive `absolute`. Variables that are approached only directly should be moved to banks 3 and 4 for increased efficiency.

mikroBASIC SPECIFICS

Predefined Globals and Constants

To facilitate programming, mikroBasic implements a number of predefined globals and constants.

All PIC SFR registers are implicitly declared as global variables of byte type, and are visible in the entire project. When creating a project, mikroBasic will include an appropriate `.def` file, containing declarations of available SFR and constants (such as PORTB, TMR1, etc). Identifiers are all in uppercase, identical to nomenclature in Microchip datasheets. For the complete set of predefined globals and constants, look for “Defs” in your mikroBasic installation folder, or probe the Code Assistant for specific letters (CTRL+SPACE in Editor).

Accessing Individual Bits

mikroBasic allows you to access individual bits of variables. Simply use the dot (`.`) with a variable, followed by a number. For example:

```
dim myvar as longint  ' range of bits is myvar.0 .. myvar.31
'...
' If RB0 is set, set the 28th bit of myvar:
if PORTB.0 = 1 then
  myvar.27 = 1
end if
```

There is no need for any special declarations; this kind of selective access is an intrinsic feature of mikroBasic and can be used anywhere in the code. Provided you are familiar with the particular chip, you can access bits by their name (e.g. `INTCON.TMR0F`).

Interrupts

Interrupts can be easily handled by means of reserved word `interrupt`. mikroBasic implictly declares procedure `interrupt` which cannot be redeclared.

Write your own procedure body to handle interrupts in your application. mikroBasic saves the following SFR on stack when entering interrupt and pops them back upon return:

PIC12 family: `W`, `STATUS`, `FSR`, `PCLATH`
PIC16 family: `W`, `STATUS`, `FSR`, `PCLATH`
PIC18 family: `FSR` (fast context is used to save `WREG`, `STATUS`, `BSR`)

Note: mikroBasic does not support low priority interrupts; for PIC18 family, interrupts must be of high priority.

Routine Calls from Interrupt

Calling functions and procedures from within the interrupt routine is now possible. The compiler takes care about the registers being used, both in "interrupt" and in "main" thread, and performs "smart" context-switching between the two, saving only the registers that have been used in both threads.

The functions and procedures that don't have their own frame (no arguments and local variables) can be called both from the interrupt and the "main" thread.

Interrupt Examples

Here is a simple example of handling the interrupts from TMR0 (if no other interrupts are allowed):

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

Linker Directives

mikroBasic uses internal algorithm to distribute objects within memory. If you need to have variable or routine at specific predefined address, use linker directives `absolute` and `org`.

Directive absolute

Directive `absolute` specifies the starting address in RAM for variable. If variable is multi-byte, higher bytes are stored at consecutive locations.

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

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

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

Be careful when using `absolute` directive, as you may overlap two variables by mistake. For example:

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

dim jjjj as longint absolute $30
' Variable jjjj will occupy bytes at $30, $31, $32, $33; thus,
' changing i changes jjjj highest byte at the same time
```

Directive org

Directive `org` specifies the starting address of routine in ROM. It is appended to the declaration of routine. For example:

```
sub procedure proc(dim par as byte) org $200
' Procedure proc will start at address $200
...
end sub
```

Note: Directive `org` can be applied to any routine except the interrupt procedure. Interrupt will always be located at address $4 (or $8 for P18), Page0.

Directive volatile

Directive volatile gives variable possibilty to change without intervention from code.

Typical volatile variables are: STATUS, TIMER0, TIMER1, PORTA, PORTB etc.

```
dim MyVar as byte absolute $123 register volatile
```

Code Optimization

Optimizer has been added to extend the compiler usability, cuts down the amount of code generated and speed-up its execution. Main features are:

Constant folding
All expressions that can be evaluated in the compile time (i.e. are constant) are being replaced by their result. (3 + 5 -> 8);

Constant propagation
When a constant value is being assigned to certain variable, the compiler recognizes this and replaces the use of the variable in the code that follows by constant, as long as variable's value remains unchanged.

Copy propagation
The compiler recognizes that two variables have same value and eliminates one of them in the further code.

Value numbering
The compiler "recognize" if the two expressions yield the same result, and can therefore eliminate the entire computation for one of them.

"Dead code" ellimination
The code snippets that are not being used elsewhere in the programme do not affect the final result of the application. They are automatically being removed.

Stack allocation
Temporary registers ("Stacks") are being used more rationally, allowing for VERY complex expressions to be evaluated with minimum stack consumption.

Local vars optimization
No local variables are being used if their result does not affect some of the global or volatile variables.

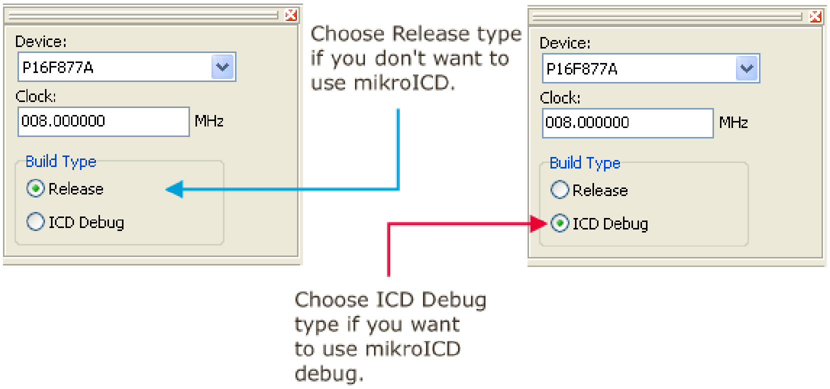
Better code generation and local optimization
Code generation is more consistent, and much attention has been made to implement specific solutions for the code "building bricks" that further reduce output code size.

mikro ICD (In-Circuit Debugger)

mikro ICD is highly effective tool for Real-Time debugging on hardware level. ICD debugger enables you to execute a mikroBasic program on a host PIC microcontroller and view variable values, Special Function Registers (SFR), memory and EEPROM as the program is running.

Step No. 1

If you have appropriate hardware and software for using mikro ICD then you have to upon completion of writing your program to choose between **Release** build Type or **ICD Debug** build type.



Device: P16F877A
Clock: 008.000000 MHz
Build Type
● Release
○ ICD Debug

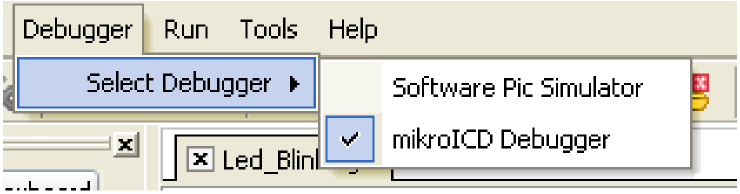
Choose Release type if you don't want to use mikroICD.

Device: P16F877A
Clock: 008.000000 MHz
Build Type
○ Release
● ICD Debug

Choose ICD Debug type if you want to use mikroICD debug.

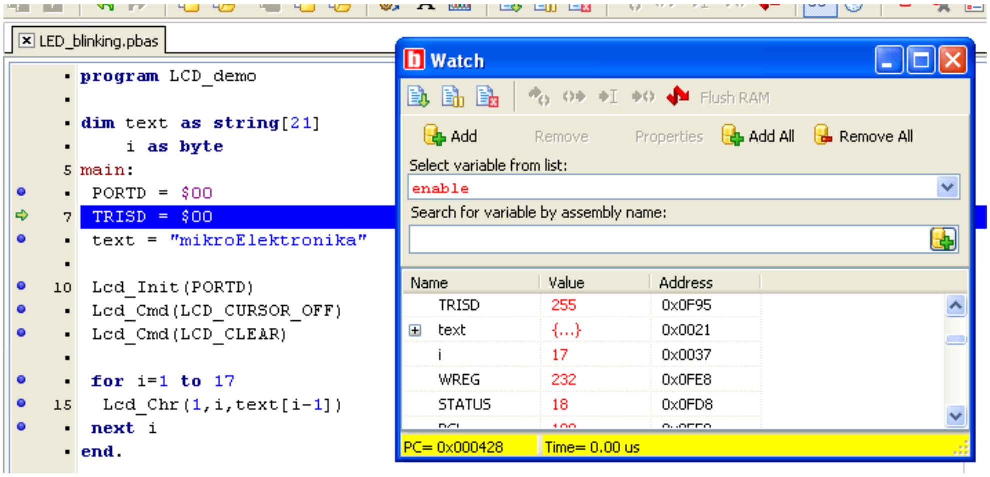
Step No. 2

After you choose **ICD Debug** build type it's time to compile your project. After you have successfully compiled your project you must program PIC using F11 shortcut. After successful PIC programming you have to select mikro ICD by selecting **Debugger › Select Debugger › mikro ICD Debugger** from the drop-down menu.



Step No. 3

You can run the mikro ICD by selecting **Run › Debug** from the 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 by default). There is also notification about program execution and it can be found on Watch Window (yellow status bar). Note that some functions take time to execute, so running of program is indicated on Watch Window.



```
program LCD_demo

dim text as string[21]
    i as byte
main:
  PORTD = $00
  TRISD = $00
  text = "mikroElektronika"

  Lcd_Init(PORTD)
  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Cmd(LCD_CLEAR)

  for i=1 to 17
   Lcd_Chr(1,i,text[i-1])
  next i
end.
```

Select variable from list: enable

Search for variable by assembly name:

Name	Value	Address
TRISD	255	0x0F95
⊞ text	{...}	0x0021
i	17	0x0037
WREG	232	0x0FE8
STATUS	18	0x0FD8

PC= 0x000428 Time= 0.00 us

mikro ICD Debugger Options

Name	Description	Function Key
`Debug`	Starts Debugger.	[F9]
`Run/ Pause Debugger`	Run or pause Debugger.	[F6]
`Toggle Breakpoints`	Toggle breakpoint at the current cursor position. To view all the breakpoints, select Run › View Breakpoints from the drop-down menu. Double clicking an item in window list locates the breakpoint.	[F5]
`Run to cursor`	Execute all instructions between the current instruction and the cursor position.	[F4]
`Step Into`	Execute the current C (single– or multi–cycle) instruction, then halt. If the instruction is a routine call, enter the routine and halt at the first instruction following the call.	[F7]
`Step Over`	Execute the current C (single– or multi–cycle) instruction, then halt. If the instruction is a routine call, skip it and halt at the first instruction following the call.	[F8]
`Flush RAM`	Flushes current PIC RAM. All variable values will be changed according to values from watch window.	N/A

Step No. 1

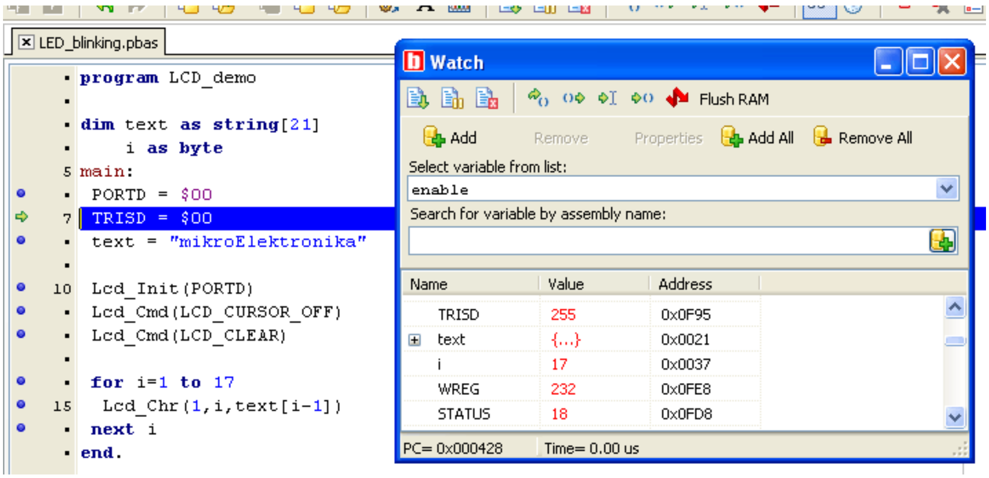
mikro ICD Debugger Example

Here is a step by step mikro ICD Debugger Example. First you have to write a program. We will show how mikro ICD works using this example:

```
program LCD_demo
dim text as string[21]
     i as byte
main:
 PORTD = $00
 TRISD = $00
 text = "mikroElektronika"

 Lcd_Init(PORTD)
 Lcd_Cmd(1)
 Lcd_Cmd(192)

 for i=1 to 17
  Lcd_Chr(1,i,text[i-1])
 next i
end.
```

Step No. 2

After successful compilation and PIC programming press **F9** for starting mikro ICD. After mikro ICD initialization blue active line should appear.



```
program LCD_demo

dim text as string[21]
    i as byte
main:
 PORTD = $00
 TRISD = $00
 text = "mikroElektronika"

 Lcd_Init(PORTD)
 Lcd_Cmd(LCD_CURSOR_OFF)
 Lcd_Cmd(LCD_CLEAR)

 for i=1 to 17
  Lcd_Chr(1,i,text[i-1])
 next i
end.
```

Select variable from list: enable

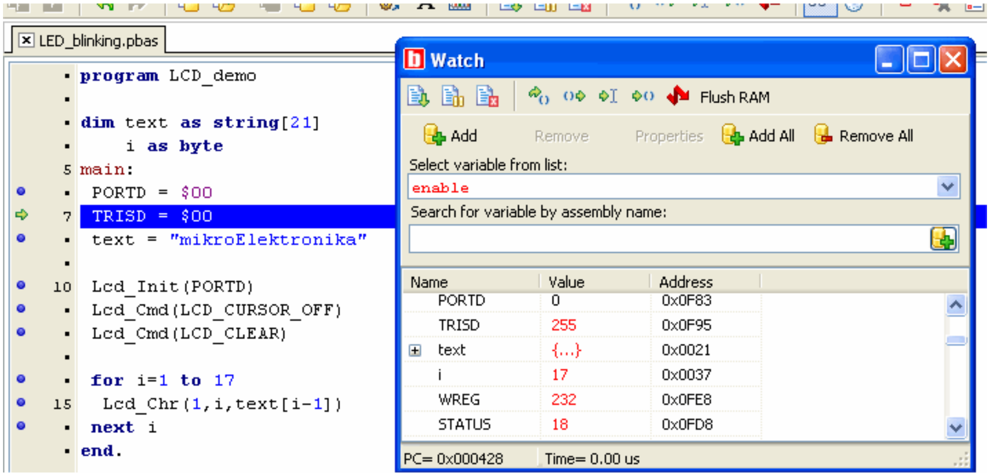
Search for variable by assembly name:

Name	Value	Address
TRISD	255	0x0F95
⊞ text	{...}	0x0021
i	17	0x0037
WREG	232	0x0FE8
STATUS	18	0x0FD8

PC= 0x000428   Time= 0.00 us

Step No. 3

We will debug program line by line. Pressing **F8** we are executing code line by line. It is recommended that user does not use Step Into [**F7**] and Step Over [**F8**] over Delays routines and routines containing delays. Instead use Run to cursor [**F4**] and Breakpoints functions.



```
program LCD_demo

dim text as string[21]
    i as byte
main:
  PORTD = $00
  TRISD = $00
  text = "mikroElektronika"

  Lcd_Init(PORTD)
  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Cmd(LCD_CLEAR)

  for i=1 to 17
   Lcd_Chr(1,i,text[i-1])
  next i
end.
```

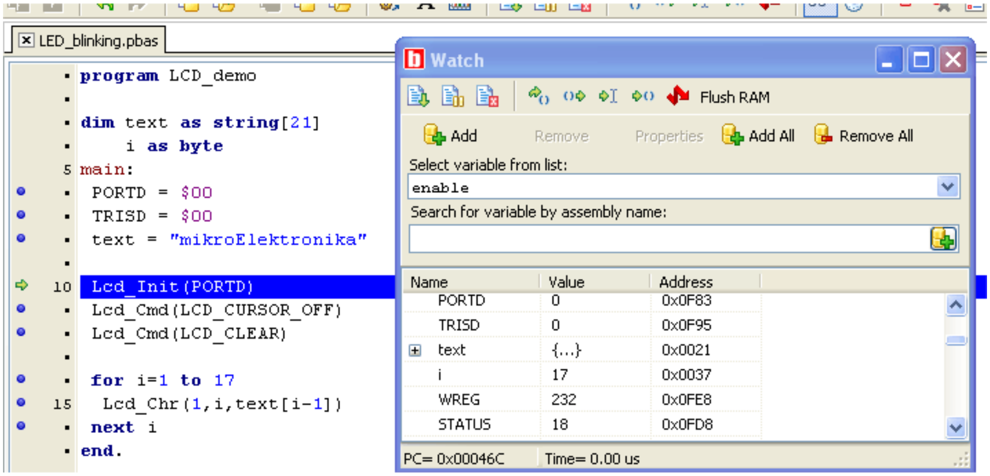
Select variable from list: enable

Search for variable by assembly name:

Name	Value	Address
PORTD	0	0x0F83
TRISD	255	0x0F95
⊞ text	{...}	0x0021
i	17	0x0037
WREG	232	0x0FE8
STATUS	18	0x0FD8

PC= 0x000428 Time= 0.00 us

All changes are read from PIC and loaded into Watch Window. Note that **TRISD** changed its value from 255 to 0.



```
program LCD_demo

dim text as string[21]
    i as byte
main:
  PORTD = $00
  TRISD = $00
  text = "mikroElektronika"

  Lcd_Init(PORTD)
  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Cmd(LCD_CLEAR)

  for i=1 to 17
   Lcd_Chr(1,i,text[i-1])
  next i
end.
```

Select variable from list: enable

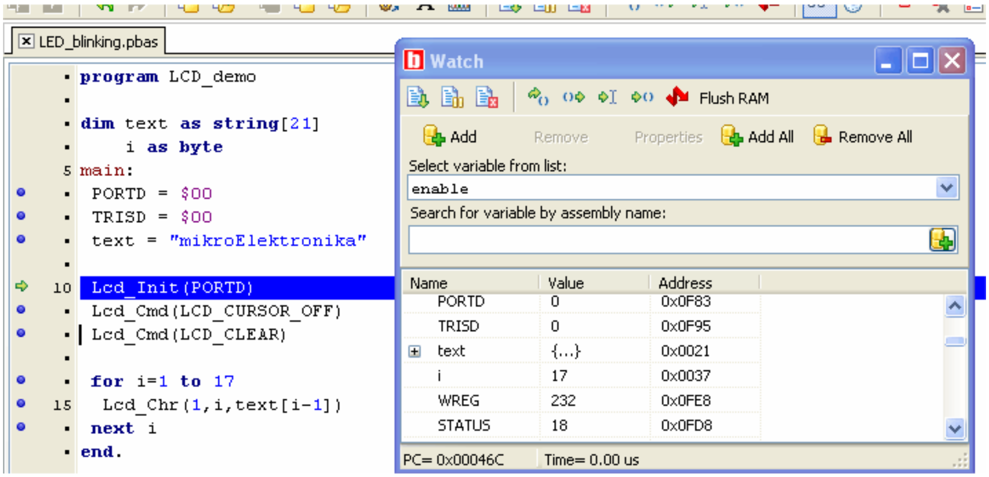
Search for variable by assembly name:

Name	Value	Address
PORTD	0	0x0F83
TRISD	0	0x0F95
⊞ text	{...}	0x0021
i	17	0x0037
WREG	232	0x0FE8
STATUS	18	0x0FD8

PC= 0x00046C Time= 0.00 us

Step No. 4

Step Into [**F7**] and Step Over [**F8**] are mikro ICD debugger functions that are used in stepping mode. There is also Real-Time mode supported by mikro ICD. Functions that are used in Real-Time mode are Run/ Pause Debugger [**F6**] and Run to cursor [**F4**]. Pressing **F4** goes to line selected by user. User just have to select line with cursor and press **F4**, and code will be executed until selected line is reached.



```
program LCD_demo

dim text as string[21]
    i as byte
main:
  PORTD = $00
  TRISD = $00
  text = "mikroElektronika"

  Lcd_Init(PORTD)
  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Cmd(LCD_CLEAR)

  for i=1 to 17
    Lcd_Chr(1,i,text[i-1])
  next i
end.
```

Select variable from list: enable

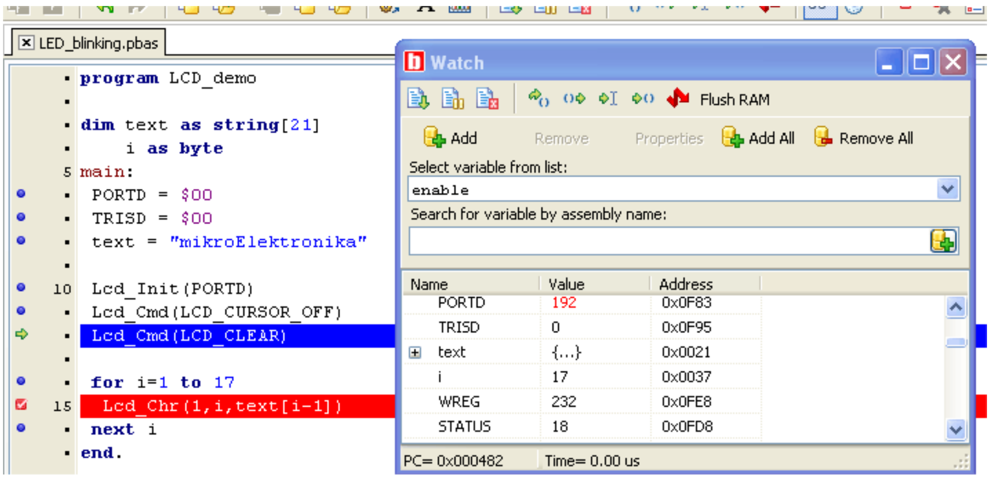
Search for variable by assembly name:

Name	Value	Address
PORTD	0	0x0F83
TRISD	0	0x0F95
⊞ text	{...}	0x0021
i	17	0x0037
WREG	232	0x0FE8
STATUS	18	0x0FD8

PC= 0x00046C Time= 0.00 us

Step No. 5

Run(Pause) Debugger [**F6**] and Toggle Breakpoints [**F5**] are mikro ICD debugger functions that are used in Real-Time mode. Pressing **F5** marks line selected by user for breakpoint. **F6** executes code until breakpoint is reached. After reaching breakpoint Debugger halts. Here at our example we will use breakpoints for writing "mikroElektronika" on LCD char by char. Breakpoint is set on LCD_Chr and program will stop everytime this function is reached. After reaching breakpoint we must press **F6** again for continuing program execution.



```
program LCD_demo

dim text as string[21]
    i as byte
main:
  PORTD = $00
  TRISD = $00
  text = "mikroElektronika"

  Lcd_Init(PORTD)
  Lcd_Cmd(LCD_CURSOR_OFF)
  Lcd_Cmd(LCD_CLEAR)

  for i=1 to 17
   Lcd_Chr(1,i,text[i-1])
  next i
end.
```

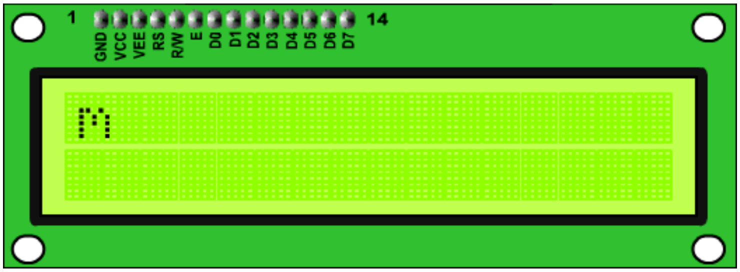
Select variable from list: enable

Search for variable by assembly name:

Name	Value	Address
PORTD	192	0x0F83
TRISD	0	0x0F95
⊞ text	{...}	0x0021
i	17	0x0037
WREG	232	0x0FE8
STATUS	18	0x0FD8

PC= 0x000482 Time= 0.00 us

Breakpoints has been separated into two groups. There are hardware and software break points. Hardware breakpoints are placed in PIC and they provide fastest debug. Number of hardware breakpoints is limited (1 for P16 and 1 or 3 for P18). If all hardware brekpoints are used, next breakpoints that will be used are software breakpoint. Those breakpoints are placed inside mikro ICD, and they simulate hardware breakpoints. Software breakpoints are much slower than hardware breakpoints. This differences between hardware and software differences are not visible in mikro ICD software but their different timings are quite notable, so it is important to know that there is two types of breakpoints.

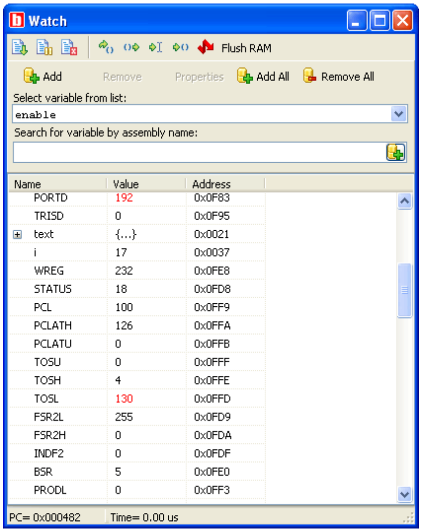


mikro ICD (In-Circuit Debugger) Overview

Watch Window

Debugger Watch Window is the main Debugger window which allows you to monitor program items while running your program. To show the Watch Window, select **View › Debug Windows › Watch Window** from the drop-down menu.

The Watch Window displays variables and registers of PIC, with their addresses and values. Values are updated as you go through the simulation. Use the drop-down menu to add and remove the items that you want to monitor. Recently changed items are colored red.



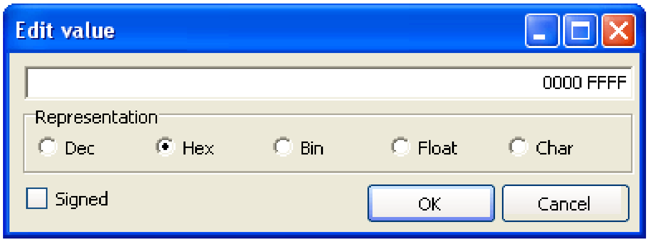
Select variable from list: enable

Search for variable by assembly name:

Name	Value	Address
PORTD	192	0x0F83
TRISD	0	0x0F95
⊞ text	{...}	0x0021
i	17	0x0037
WREG	232	0x0FE8
STATUS	18	0x0FD8
PCL	100	0x0FF9
PCLATH	126	0x0FFA
PCLATU	0	0x0FFB
TOSU	0	0x0FFF
TOSH	4	0x0FFE
TOSL	130	0x0FFD
FSR2L	255	0x0FD9
FSR2H	0	0x0FDA
INDF2	0	0x0FDF
BSR	5	0x0FE0
PRODL	0	0x0FF3

PC= 0x000482 Time= 0.00 us

Double clicking an item opens the Edit Value window in which you can assign a new value to the selected variable/register. Also, you can change view to binary, hex, char, or decimal for the selected item.



0000 FFFF

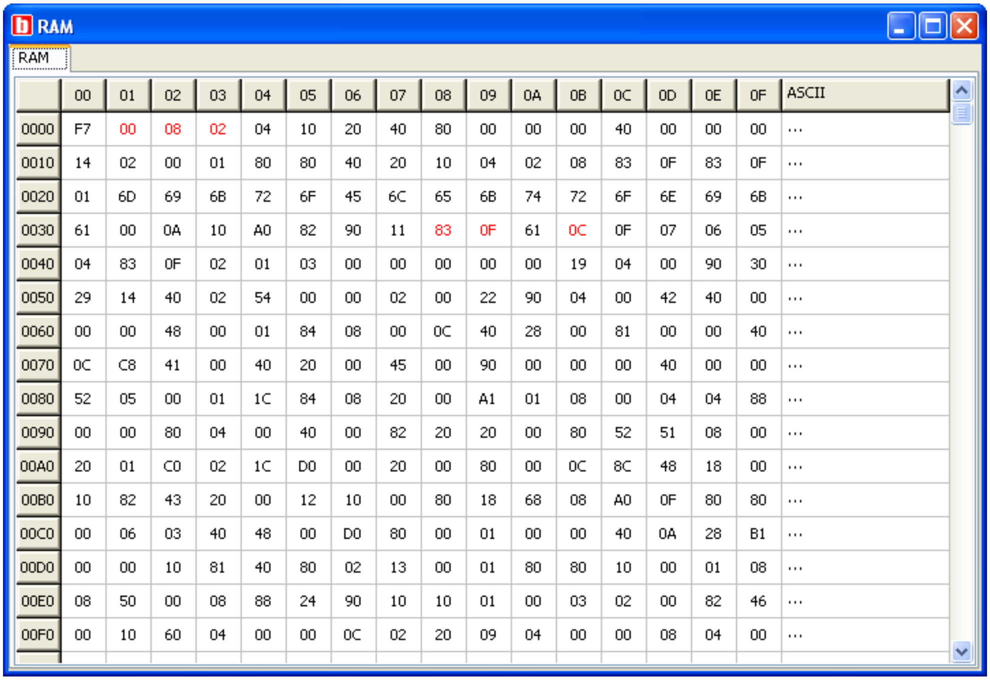
Representation
○ Dec ● Hex ○ Bin ○ Float ○ Char

☐ Signed

View RAM Window

Debugger View RAM Window is available from the drop-down menu, **View › Debug Windows › View RAM**.

The View RAM Window displays the map of PIC’s RAM, with recently changed items colored red.



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	F7	00	08	02	04	10	20	40	80	00	00	00	40	00	00	00	...
0010	14	02	00	01	80	80	40	20	10	04	02	08	83	0F	83	0F	...
0020	01	6D	69	6B	72	6F	45	6C	65	6B	74	72	6F	6E	69	6B	...
0030	61	00	0A	10	A0	82	90	11	83	0F	61	0C	0F	07	06	05	...
0040	04	83	0F	02	01	03	00	00	00	00	00	19	04	00	90	30	...
0050	29	14	40	02	54	00	00	02	00	22	90	04	00	42	40	00	...
0060	00	00	48	00	01	84	08	00	0C	40	28	00	81	00	00	40	...
0070	0C	C8	41	00	40	20	00	45	00	90	00	00	00	40	00	00	...
0080	52	05	00	01	1C	84	08	20	00	A1	01	08	00	04	04	88	...
0090	00	00	80	04	00	40	00	82	20	20	00	80	52	51	08	00	...
00A0	20	01	C0	02	1C	D0	00	20	00	80	00	0C	8C	48	18	00	...
00B0	10	82	43	20	00	12	10	00	80	18	68	08	A0	0F	80	80	...
00C0	00	06	03	40	48	00	D0	80	00	01	00	00	40	0A	28	B1	...
00D0	00	00	10	81	40	80	02	13	00	01	80	80	10	00	01	08	...
00E0	08	50	00	08	88	24	90	10	10	01	00	03	02	00	82	46	...
00F0	00	10	60	04	00	00	0C	02	20	09	04	00	00	08	04	00	...

Common Errors

- Trying to program PIC while mikro ICD is active.
- Trying to debug **Release** build Type version of program.
- Trying to debug changed program code which hasn't been compiled and programmed into PIC.
- Trying to select line that is empty for Run to cursor [**F4**] and Toggle Breakpoints [**F5**] functions.

LEXICAL ELEMENTS

These topics provide a formal definition of the mikroBasic lexical elements. They describe the different categories of word-like units (tokens) recognized by a language.

In the tokenizing phase of compilation, the source code file is parsed (that is, broken down) into *tokens* and *whitespace*. The tokens in mikroBasic are derived from a series of operations performed on your programs by the compiler.

A mikroBasic program starts as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the mikroBasic Code Editor). The basic program unit in mikroBasic is the file. This usually corresponds to a named file located in RAM or on disk and having the extension `.pbas`.

Whitespace

Whitespace is the collective name given to spaces (blanks), horizontal and vertical tabs, and comments. Whitespace serves to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded.

For example, the two sequences

```
dim tmp as byte
dim j as word
```

and

```
dim  tmp   as  byte
dim    j   as   word
```

are lexically equivalent and parse identically.

Note: Newline character (`CR/LF`) is not a whitespace in BASIC, and serves as a statement terminator/separator. In mikroBasic, however, you *may* use newline to break long statements into several lines. Parser will first try to get the longest possible expression (across lines if necessary), and then check for statement terminators.

Newline Character

Newline character (CR/LF) is not a whitespace in BASIC, and serves as a statement terminator/separator. Optionally, you may use newline to break very long statements into several lines, as parser will first try to get the longest possible expression. See Statements for more information.

Whitespace in Strings

The ASCII characters representing whitespace can occur within string literals, in which case they are protected from the normal parsing process (they remain as part of the string). For example, statement

```
some_string = "mikro foo"
```

parses to four tokens, including the single string literal token:

```
some_string
=
"mikro foo"
newline character
```

Comments

Comments are pieces of text used to annotate a program, and are technically another form of whitespace. Comments are for the programmer's use only; they are stripped from the source text before parsing.

Use the apostrophe to create a comment:

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

Multi-line comments are not supported in BASIC.

TOKENS

Token is the smallest element of a BASIC program that is meaningful to the compiler. The parser separates tokens from the input stream by creating the longest token possible using the input characters in a left–to–right scan.

mikroBasic recognizes these kinds of tokens:

- keywords
- identifiers
- constants
- operators
- punctuators (also known as separators)

Token Extraction Example

Here is an example of token extraction. Let's have the following code sequence:

```
end_flag = 0
```

The compiler would parse it as the following four tokens:

```
end_flag      ' variable identifier
=             ' assignment operator
0             ' literal
newline       ' statement terminator
```

Note that `end_flag` would be parsed as a single identifier, rather than as the keyword `end` followed by the identifier `_flag`.

LITERALS

Literals are tokens representing fixed numeric or character values.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code.

Integer Literals

Integral values can be represented in decimal, hexadecimal, or binary notation.

In decimal notation, numerals are represented as a sequence of digits (without commas, spaces, or dots), with optional prefix `+` or `-` operator to indicate the sign. Values default to positive (`6258` is equivalent to `+6258`).

The dollar-sign prefix (`$`) or the prefix `0x` indicates a hexadecimal numeral (for example, `$8F` or `0x8F`).

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

The allowed range of values is imposed by the largest data type in mikroBasic – `longint`. Compiler will report an error if the literal exceeds `2147483647` (`$7FFFFFFF`).

Floating Point Literals

A floating-point value consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- `e` or `E` and a signed integer exponent (optional)

Negative floating constants are taken as positive constants with the unary operator minus (`-`) prefixed.

mikroBasic limits floating-point constants to range
`±1.17549435082E38 .. ±6.80564774407E38`.

Here are some examples:

```
0.             ' = 0.0
-1.23          ' = -1.23
23.45e6        ' = 23.45 * 10^6
2e-5           ' = 2.0 * 10^-5
3E+10          ' = 3.0 * 10^10
.09E34         ' = 0.09 * 10^34
```

Character Literals

Character literal is one character from the extended ASCII character set, enclosed by quotes (for example, `"A"`). Character literal can be assigned to variables of byte and char type (variable of byte will be assigned the ASCII value of the character). Also, you can assign character literal to a string variable.

String Literals

String literal is a sequence of up to 255 characters from the extended ASCII character set, enclosed by quotes. Whitespace is preserved in string literals, i.e. parser does not "go into" strings but treats them as single tokens.

Length of string literal is the number of characters it consists of. String is stored internally as the given sequence of characters plus a final null character (ASCII zero). This appended "stamp" does not count against string's total length. String literal with nothing in between the quotes (*null string*) is stored as a single null character. You can assign string literal to a string variable or to an array of `char`.

Here are several string literals:

```
"Hello world!"       ' message, 12 chars long
"  "                 ' two spaces, 2 chars long
"C"                  ' letter, 1 char long
""                   ' null string, 0 chars long
```

Quote itself cannot be a part of the string literal, i.e. there is no escape sequence.

KEYWORDS

Keywords are words reserved for special purposes and must not be used as normal identifier names.

Beside standard BASIC keywords, all relevant SFR are defined as global variables and represent reserved words that cannot be redefined (for example: TMR0, PCL, etc). Probe the Code Assistant for specific letters (CTRL+SPACE in Editor) or refer to Predefined Globals and Constants.

Here is the alphabetical listing of keywords in mikroBasic:

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

Also, mikroBasic includes a number of predefined identifiers used in libraries. You could replace these by your own definitions, if you plan to develop your own libraries. For more information, see mikroBasic Libraries.

IDENTIFIERS

Identifiers are arbitrary names of any length given to functions, variables, symbolic constants, user-defined data types, and labels. All these program elements will be referred to as objects throughout the help (not to be confused with the meaning of object in object-oriented programming).

Identifiers can contain the letters `a` to `z` and `A` to `Z`, the underscore character `'_'`, and the digits `0` to `9`. The only restriction is that the first character must be a letter or an underscore.

Case Sensitivity

BASIC is not case sensitive, so `Sum`, `sum`, and `suM` represent an equivalent identifier.

Uniqueness and Scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same scope and sharing the same name space. Duplicate names are legal for different name spaces regardless of scope rules. For more information on scope, refer to Scope and Visibility.

Identifier Examples

Here are some valid identifiers:

```
temperature_V1
Pressure
no_hit
dat2string
SUM3
_vtext
```

PUNCTUATORS

The mikroBasic punctuators (also known as separators) include brackets, parentheses, comma, colon, and dot.

Brackets

Brackets `[ ]` indicate single and multidimensional array subscripts:

```
dim alphabet as byte[30]
' ...
alphabet[2] = "c"
```

For more information, refer to Arrays.

Parentheses

Parentheses `( )` are used to group expressions, isolate conditional expressions, and indicate function calls and function declarations:

```
d = c * (a + b)              ' Override normal precedence
if (d = z) then ...          ' Useful with conditional statements
func()                       ' Function call, no args
sub function func2(dim n as word)   ' Function declaration
```

For more information, refer to Operators Precedence and Associativity, Expressions, or Functions and Procedures.

Comma

The comma (`,`) separates the arguments in routine calls:

```
Lcd_Out(1, 1, txt)
```

Further, the comma separates identifiers in declarations:

```
dim i, j, k as word
```

The comma also separates elements in initialization lists of constant arrays:

```
const MONTHS as byte[ 12]  = (31,28,31,30,31,30,31,31,30,31,30,31)
```

Colon

Colon (`:`) is used to indicate a labeled statement:

```
start:  nop
  ...
goto start
```

For more information, refer to Labels.

Dot

Dot (`.`) indicates access to a structure member. For example:

```
person.surname = "Smith"
```

For more information, refer to Structures.

Dot is a necessary part of floating point literals. Also, dot can be used for accessing individual bits of registers in mikroBasic.

PROGRAM ORGANIZATION

mikroBasic imposes strict program organization. Below you can find models for writing legible and organized source files. For more information on file inclusion and scope, refer to Modules and to Scope and Visibility.

Organization of Main Module

Basically, main source file has two sections: declaration and program body. Declarations should be in their proper place in the code, organized in an orderly manner. Otherwise, compiler may not be able to comprehend the program correctly.

When writing code, follow the model presented in the following page.

Organization of Other Modules

Units other than main start with the keyword `module`; implementation section starts with the keyword implements. Follow the models presented in the following two pages.

Main unit should look like this:

```
program <program name>
include <include other modules>

'********************************************************
'* Declarations (globals):
'********************************************************

' symbols declarations
symbol ...

' constants declarations
const ...

' variables declarations
dim ...

' procedures declarations
sub procedure procedure_name(...)
  <local declarations>
  ...
end sub

' functions declarations
sub function function_name(...)
  <local declarations>
  ...
end sub

'********************************************************
'* Program body:
'********************************************************

main:
  ' write your code here
end.
```

Other units should look like this:

```
module  <module name>
include <include other modules>

'**********************************************************
'* Interface (globals):
'**********************************************************

' symbols declarations
symbol ...

' constants declarations
const ...

' variables declarations
dim ...

' procedures prototypes
sub procedure procedure_name(...)

' functions prototypes
sub function function_name(...)

'**********************************************************
'* Implementation:
'**********************************************************

implements

' constants declarations
const ...

' variables declarations
dim ...

' procedures declarations
sub procedure procedure_name(...)
  <local declarations>
  ...
end sub

' functions declarations
sub function function_name(...)
  <local declarations>
  ...
end sub

end.
```

SCOPE AND VISIBILITY

Scope

The scope of identifier is the part of the program in which the identifier can be used to access its object. There are different categories of scope which depend on how and where identifiers are declared:

If identifier is declared in the declaration section of a main module, out of any function or procedure, scope extends from the point where it is declared to the end of the current file, including all routines enclosed within that scope. These identifiers have a file scope, and are referred to as *globals*.

If identifier is declared in the function or procedure, scope extends from the point where it is declared to the end of the current routine. These identifiers are referred to as *locals.*

If identifier is declared in the interface section of a module, scope extends the interface section of a module from the point where it is declared to the end of the module, and to any other module or program that uses that module. The only exception are symbols which have scope limited to the file in which they are declared.

If identifier is declared in the implementation section of a module, but not within any function or procedure, 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.

Visibility

The visibility of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Technically, visibility cannot exceed scope, but scope *can* exceed visibility.

MODULES

In mikroBasic, each project consists of a single project file, and one or more module files. Project file, with extension `.pbp` contains information about the project, while modules, with extension `.pbas`, contain the actual source code.

Modules allow you to:

- break large programs into encapsulated modules that can be edited separately,
- create libraries that can be used in different projects,
- distribute libraries to other developers without disclosing the source code.

Each module is stored in its own file and compiled separately; compiled modules are linked to create an application. To build a project, the compiler needs either a source file or a compiled module file for each module.

Include Clause

mikroBasic includes modules by means of `include` clause. It consists of the reserved word `include`, followed by a quoted module name. Extension of the file should not be included.

You can include one file per `include` clause. There can be any number of `include` clauses in each source file, but they all must be stated immediately after the program (or module) name.

Here's an example:

```
program MyProgram

include "utils"
include "strings"
include "MyUnit"
...
```

Given a module name, compiler will check for the presence of `.mcl` and `.pbas` files, in order specified by the search paths.

- If both `.pbas` and `.mcl` files are found, compiler will check their dates and include the newer one in the project. If the `.pbas` file is newer than the .mcl, new library will be written over the old one;

- If only `.pbas` file is found, compiler will create the `.mcl` file and include it in the project;

- If only `.mcl` file is present, i.e. no source code is available, compiler will include it as found;

- If none found, compiler will issue a “File not found” warning.

Main Module

Every project in mikroBasic requires single main module file. Main module is identified by the keyword `program` at the beginning; it instructs the compiler where to “start”.

After you have successfully created an empty project with Project Wizard, Code Editor will display a new main module. It contains the bare-bones of a program:

```
program MyProject

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

Other than comments, nothing should precede the keyword program. After the program name, you can optionally place the `include` clauses.

Place all global declarations (constants, variables, labels, routines) before the label `main`.

Note: In mikroBasic, the `end.` statement (the closing statement of every program) acts as an endless loop.

Other Modules

Modules other than main start with the keyword `module`. Newly created blank module contains the bare-bones:

```
module MyModule

implements

end.
```

Other than comments, nothing should precede the keyword `module`. After the module name, you can optionally place the `include` clause.

Interface Section

Part of the module above the keyword `implements` is referred to as interface section. Here, you can place global declarations (constants, variables, and labels) for the project.

You do not define routines in the interface section. Instead, state the prototypes of routines (from implementation section) that you want to be visible outside the module. Prototypes must match the declarations exactly.

Implementation Section

Implementation section hides all the irrelevant innards from other modules, allowing encapsulation of code.

Everything declared below the keyword `implements` is *private*, i.e. has its scope limited to the file. When you declare an identifier in the implementation section of a module, you cannot use it outside the module, but you can use it in any block or routine defined within the module.

By placing the prototype in the interface section of the module (above the `implements`) you can make the routine *public*, i.e. visible outside of module. Prototypes must match the declarations exactly.

VARIABLES

Variable is object 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.

Variables are declared in the declaration part of the file or routine — each variable needs to be declared before it can be used. Global variables (those that do not belong to any enclosing block) are declared below the `include` statement, above the label `main`.

Specifying a data type for each variable is mandatory. mikroBasic syntax for variable declaration is:

```
dim identifier_list as type
```

Here, *`identifier_list`* is a comma-delimited list of valid identifiers, and *`type`* can be any data type.

For more details refer to Types and Types Conversions. See also Scope and Visibility.

Here are a few examples of variable declarations:

```
dim i, j, k as byte
dim counter, temp as word
```

Variables and PIC

Every declared variable consumes part of 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. mikroBasic recycles local variable memory space – local variables declared in different functions and procedures share same memory space, if possible.

There is no need to declare SFR explicitly, as mikroBasic automatically declares relevant registers as global variables of `byte`. For example: T0IE, INTF, etc.

CONSTANTS

Constant is data whose value cannot be changed during the runtime. Using a constant in a program consumes no RAM memory. Constants can be used in any expression, but cannot be assigned a new value.

Constants are declared in the declaration part of program or routine. You can declare any number of constants after the keyword `const`:

```
const constant_name [ as  type]  = value
```

Every constant is declared under unique *`constant_name`* which must be a valid identifier. It is a tradition to write constant names in uppercase. Constant requires you to specify *`value`*, which is a literal appropriate for the given type. The *`type`* is optional; in the absence of *`type`*, compiler assumes the “smallest” of the types that can accommodate *`value`*.

Note: You cannot omit type if declaring a constant array.

Here are a few examples:

```
const MAX as longint = 10000
const MIN = 1000            ' compiler will assume word type
const SWITCH = "n"          ' compiler will assume char type
const MSG = "Hello"         ' compiler will assume string type
const MONTHS as byte[ 12]  = (31,28,31,30,31,30,31,31,30,31,30,31)
```

LABELS

Labels serve as targets for `goto` and `gosub` statements. Mark the desired statement with label and a colon like this:

```
label_identifier : statement
```

No special declaration of label is necessary in mikroBasic.

Name of the label needs to be a valid identifier. The labeled statement, and `goto/gosub` statement must belong to the same block. Hence it is not possible to jump into or out of a procedure or a function. Do not mark more than one statement in a block with the same label.

Note: Label `main` marks the entry point of a program and must be present in the main module of every project. See Program Organization for more information.

Here is an example of an infinite loop that calls the procedure `Beep` repeatedly:

```
loop: Beep
goto loop
```

SYMBOLS

BASIC symbols allow you to create simple macros without parameters. You can replace any one line of code with a single identifier alias. Symbols, when properly used, can increase code legibility and reusability.

Symbols need to be declared at the very beginning of the module, right after the module name and the (optional) `include` clauses. Check Program Organization for more details. Scope of a symbol is always limited to the file in which it has been declared.

Symbol is declared as:

```
symbol alias = code
```

Here, *`alias`* must be a valid identifier which you will be using throughout the code. This identifier has file scope. The *`code`* can be any one line of code (literals, assignments, function calls, etc).

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

Here are a few examples:

```
symbol MAXALLOWED = 216    ' Symbol as alias for numeric value
symbol PORT = PORTC        ' Symbol as alias for SFR
symbol MYDELAY = Delay_ms(1000)   ' Symbol as alias for proc. call

dim cnt as byte   ' Some variable

'...
main:

if cnt > MAXALLOWED then
  cnt = 0
  PORT.1 = 0
  MYDELAY
end if
```

Note: Symbols do not support macro expansion in the way C preprocessor does.

FUNCTIONS AND PROCEDURES

Functions and procedures, collectively referred to as *routines*, are subprograms (self-contained statement blocks) which perform a certain task based on a number of input parameters. Function returns a value when executed, and procedure does not.

mikroBasic does not support inline routines.

Functions

Function is declared like this:

```
sub function function_name(parameter_list) as return_type
  [  local declarations ]
   function body
end sub
```

The `function_name` represents a function's name and can be any valid identifier. The `return_type` is the type of return value and can be any simple type. Within parentheses, `parameter_list` is a formal parameter list similar to variable declaration. In mikroBasic, parameters are always passed to function by value; to pass argument by the address, add the keyword byref ahead of identifier.

`Local declarations` are optional declarations of variables and/or constants, local for the given function. `Function body` is a sequence of statements to be executed upon calling the function.

A function is called by its name, with actual arguments placed in the same sequence as their matching formal parameters. The compiler is able to coerce mismatching arguments to the proper type according to implicit conversion rules. Upon function call, all formal parameters are created as local objects initialized by values of actual arguments. Upon return from a function, temporary object is created in the place of the call, and it is initialized by the expression of `return` statement. This means that function call as an operand in complex expression is treated as the function result.

Use the variable `result` (automatically created local) to assign the return value of a function.

Function calls are considered to be primary expressions, and can be used in situations where expression is expected. Function call can also be a self-contained statement, in which case the return value is discarded.

Here's a simple function which calculates x^n based on input parameters `x` and `n` (`n > 0`):

```
sub function power(dim x, n as byte) as longint
dim i as byte
  i = 0
  result = 1
  if n > 0 then
    for i = 1 to n
      result = result*x
    next i
  end if
end sub
```

Now we could call it to calculate, say, 3^{12}:

```
tmp = power(3, 12)
```

Procedures

Procedure is declared like this:

```
sub procedure procedure_name(parameter_list)
  [ local declarations ]
  procedure body
end sub
```

The *`procedure_name`* represents a procedure's name and can be any valid identifier. Within parentheses, *`parameter_list`* is a formal parameter list similar to variable declaration. In mikroBasic, parameters are always passed to procedure by value; to pass argument by the address, add the keyword `byref` ahead of identifier.

`Local declarations` are optional declaration of variables and/or constants, local for the given procedure. *`Procedure body`* is a sequence of statements to be executed upon calling the procedure.

A procedure is called by its name, with actual arguments placed in the same sequence as their matching formal parameters. The compiler is able to coerce mismatching arguments to the proper type according to implicit conversion rules. Upon procedure call, all formal parameters are created as local objects initialized by values of actual arguments.

Procedure call is a self-contained statement.

Here's an example procedure which transforms its input time parameters, preparing them for output on LCD:

```
sub procedure time_prep(dim byref sec, min, hr as byte)
  sec  = ((sec and $F0) >> 4)*10 + (sec and $0F)
  min  = ((min and $F0) >> 4)*10 + (min and $0F)
  hr   = ((hr  and $F0) >> 4)*10 + (hr  and $0F)
end sub
```

TYPES

BASIC is a strictly typed language, which means that every variable and constant need to have a strictly defined type, known at the time of compilation.

The type serves:

- to determine the correct memory allocation required,
- to interpret the bit patterns found in the object during subsequent accesses,
- in many type-checking situations, to ensure that illegal assignments are trapped.

mikroBasic supports many standard (predefined) and user-defined data types, including signed and unsigned integers of various sizes, arrays, strings, pointers, and structures.

Type Categories

Types can be divided into:

- simple types
- arrays
- strings
- pointers
- structures (user defined types)

SIMPLE TYPES

Simple types represent types that cannot be divided into more basic elements, and are the model for representing elementary data on machine level.

Here is an overview of simple types in mikroBasic:

Type	Size	Range
`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 .. 2147483647
`float`	32-bit	$\pm 1.17549435082 * 10^{-38}$ .. $\pm 6.80564774407 * 10^{38}$

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

You can assign signed to unsigned or vice versa only using the explicit conversion. Refer to Types Conversions for more information.

ARRAYS

An array represents an indexed collection of elements of the same type (called the base type). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once.

Array types are denoted by constructions of the form:

```
type[ array_length]
```

Each of the elements of an array is numbered from 0 through the *`array_length`* - 1. Every element of an array is of *`type`* and can be accessed by specifying array name followed by element's index within brackets.

Here are a few examples of array declaration:

```
dim weekdays as byte[ 7]
dim samples  as word[ 50]

begin
  ' Now we can access elements of array variables, for example:
  samples[ 0]  = 1
  if samples[ 37]  = 0 then
    ...
```

Constant Arrays

Constant array is initialized by assigning it a comma-delimited sequence of values within parentheses. For example:

```
' Declare a constant array which holds no. of days in each month:
const MONTHS as byte[ 12]  = (31,28,31,30,31,30,31,31,30,31,30,31)
' Declare constant numbers:
const NUMBER as byte[ 4][ 4]  = ((0, 1, 2, 3), (5, 6, 7, 8), (9, 10,
11,12), (13,14, 15, 16))
```

Note that indexing is zero based; in the previous example, number of days in January is `MONTHS[ 0]` , and number of days in December is `MONTHS[ 11]` .

The number of assigned values must not exceed the specified length. Vice versa is possible, when the trailing "excess" elements will be assigned zeroes.

For more information on arrays of char, refer to Strings.

MULTI-DIMENSIONAL ARRAYS

An array is one-dimensional if it is of scalar type. One-dimensional arrays are sometimes referred to as vectors.

Multidimensional arrays are constructed by declaring arrays of array type. These arrays are stored in memory in such way that the right most subscript changes fastest, i.e. arrays are stored “in rows”. Here is a sample 2-dimensional array:

```
dim m as byte[50][20]    '2-dimensional array of size 50x20
```

Variable m is an array of 50 elements, which in turn are arrays of 20 bytes each. Thus, we have a matrix of 50x20 elements: first element is m[0][0], last one is m[49][19]. First element of the 5th row would be m[0][5].

If you are not initializing the array in the declaration, you can omit the first dimension of multi-dimensional array. In that case, array is located elsewhere, e.g. in another file. This is a commonly used technique when passing arrays as function parameters:

```
sub procedure example(dim byref m as byte[50][20])
' we can omit first dimension
...
inc(m[1][1])
end sub

dim m as byte[50][20]   '2-dimensional array of size 50x20
dim n as byte[4][2][7]  '3-dimensional array of size 4x2x7
main:
...
func(m)
end.
```

STRINGS

A string represents a sequence of characters, and is an equivalent to an array of `char`. It is declared like:

```
string[ string_length]
```

Specifier *`string_length`* is the number of characters string consists of. String is stored internally as the given sequence of characters plus a final null character (zero). This appended “stamp” does not count against string’s total length.

A null string (`""`) is stored as a single null character.

You can assign string literals or other strings to string variables. String on the right side of an assignment operator has to be the shorter of the two, or of equal length. For example:

```
dim msg1 as string[ 20]
dim msg2 as string[ 19]

begin
  msg1 = "This is some message"
  msg2 = "Yet another message"
  msg1 = msg2 ' this is ok, but vice versa would be illegal
```

Alternately, you can handle strings element–by–element. For example:

```
dim s as string[ 5]
...
s = "mik"
' s[0] is char literal "m"
' s[1] is char literal "i"
' s[2] is char literal "k"
' s[3] is zero
' s[4] is undefined
' s[5] is undefined
```

Be careful when handling strings in this way, since overwriting the end of a string can cause access violations.

POINTERS

A pointer is a data type which holds a memory address. While a variable accesses that memory address directly, a pointer can be thought of as a reference to that memory address.

To declare a pointer data type, add a carat prefix (`^`) before type. For example, if you are creating a pointer to an `integer`, you would write:

```
^integer
```

To access the data at the pointer's memory location, you add a carat after the variable name. For example, let's declare variable `p` which points to `integer`, and then assign the pointed memory location value 5:

```
dim p as ^integer
...
p^ = 5
```

A pointer can be assigned to another pointer. However, note that only the address, not the value, is copied. Once you modify the data located at one pointer, the other pointer, when dereferenced, also yields modified data.

@ Operator

The `@` operator returns the address of a variable or routine; that is, `@` constructs a pointer to its operand. The following rules apply to `@`:

- If `X` is a variable, `@X` returns the address of `X`.
- If `F` is a routine (a function or procedure), `@F` returns `F`'s entry point (result is of `longint`).

STRUCTURES

A structure represents a heterogeneous set of elements. Each element is called a member; the declaration of a structure type specifies a name and type for each member. The syntax of a `structure` type declaration is

```
structure structname
   dim member1 as type1
   ...
   dim membern as typen
end structure
```

where *`structname`* is a valid identifier, each *`type`* denotes a type, and each member is a valid identifier. The scope of a member identifier is limited to the structure in which it occurs, so you don't have to worry about naming conflicts between member identifiers and other variables.

For example, the following declaration creates a structure type called `Dot`:

```
structure Dot
  dim x as float
  dim y as float
end structure
```

Each Dot contains two members: x and y coordinates; memory is allocated when you instantiate the structure, like this:

```
dim m as Dot
dim n as Dot
```

This variable declaration creates two instances of `Dot`, called `m` and `n`.

A member can be of previously defined structure type. For example:

```
' Structure defining a circle:
structure Circle
  dim radius as real
  dim center as Dot
end structure
```

Structure Member Access

You can access the members of a structure by means of dot (`.`). If we had declared variables `circle1` and `circle2` of previously defined type `Circle`:

```
dim circle1, circle2 as Circle
```

we could access their individual members like this:

```
circle1.radius = 3.7
circle1.center.x = 0
circle1.center.y = 0
```

You can also commit assignments between complex variables, if they are of the same type:

```
circle2 = circle1   ' This will copy values of all members
```

TYPES CONVERSIONS

Conversion of object of one type is changing it to the same object of another type (i.e. applying another type to a given object). mikroBasic supports both implicit and explicit conversions for built-in types.

Implicit Conversion

You cannot mix signed and unsigned objects in expressions with arithmetic or logical operators. You can assign signed to unsigned or vice versa only using the explicit conversion.

Compiler will provide an automatic implicit conversion in the following situations:

- statement requires an expression of particular type (according to language definition), and we use an expression of different type,
- operator requires an operand of particular type, and we use an operand of different type,
- function requires a formal parameter of particular type, and we pass it an object of different type,
- result does not match the declared function return type.

Promotion

When operands are of different types, implicit conversion promotes the less complex to more complex type taking the following steps:

```
byte/char     ->     word
short         ->     integer
short         ->     longint
integer       ->     longint
integral      ->     float
```

Higher bytes of extended unsigned operand are filled with zeroes. Higher bytes of extended signed operand are filled with bit sign (if number is negative, fill higher bytes with one, otherwise with zeroes).

Clipping

In assignments, and statements that require an expression of particular type, 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).

If expression evaluates to more complex type than expected, excess data will be simply clipped (higher bytes are lost).

```
dim i as byte
dim j as word
...
j = $FF0F
i = j   ' i becomes $0F, higher byte $FF is lost
```

Explicit Conversion

Explicit conversion can be executed at any point by inserting type keyword (`byte`, `word`, `short`, `integer`, or `longint`) ahead of the expression to be converted. The expression must be enclosed in parentheses. Explicit conversion can be performed only on the operand left of the assignment operator.

Special case is conversion between signed and unsigned types. Explicit conversion between signed and unsigned data does not change binary representation of data; it merely allows copying of source to destination.

For example:

```
dim a as byte
dim b as short
...
b = -1
a = byte(b)   ' a is 255, not 1

' This is because binary representation remains
' 11111111; it's just interpreted differently now
```

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

Arithmetic Conversions

When you use an arithmetic expression, such as a + b, where a and b are of different arithmetic types, mikroBasic performs implicit type conversions before the expression is evaluated. These standard conversions include promotions of “lower” types to “higher” types in the interests of accuracy and consistency.

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type short always use sign extension; objects of type byte always set the high byte to zero when converted to int.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged. Converting a shorter integral type to a longer type either sign-extends or zero-fills the extra bits of the new value, depending on whether the shorter type is signed or unsigned, respectively.

Note: Conversion of floating point data into integral value (in assignments or via explicit typecast) produces correct results only if the float value does not exceed the scope of destination integral type.

In details:

Here are the steps mikroBasic uses to convert the operands in an arithmetic expression:

First, any small integral types are converted according to the following rules:

byte converts to integer
short converts to integer, with the same value
short converts to integer, with the same value, sign-extended
byte converts to integer, with the same value, zero-filled
The result of the expression is the same type as that of the two operands.

Here are several examples of implicit conversion:

```
2 + 3.1       ' -> 2. + 3.1 -> 5.1
5 / 4 * 3.    ' -> (5/4)*3. -> 1*3. -> 1.*3. -> 3.
3. * 5 / 4    ' -> (3.*5)/4 -> (3.*5.)/4 -> 15./4 -> 15./4. -> 3.75
```

OPERATORS

Operators are tokens that trigger some computation when applied to variables and other objects in an expression.

There are four types of operators in mikroBasic:

- Arithmetic Operators
- Bitwise Operators
- Boolean Operators
- Relational Operators

Operators Precedence and Associativity

There are 4 precedence categories in mikroBasic. Operators in the same category have equal precedence with each other.

Each category has an associativity rule: left-to-right, or right-to-left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

Precedence	Operands	Operators	Associativity
4	1	`@` `not` `+` `-`	right-to-left
3	2	`*` `/` `div` `mod` `and` `<<` `>>`	left-to-right
2	2	`+` `-` `or` `xor`	left-to-right
1	2	`=` `<>` `<` `>` `<=` `>=`	left-to-right

Arithmetic Operators

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results. As `char` operators are technically `bytes`, they can be also used as unsigned operands in arithmetic operations. Operands need to be either both signed or both unsigned.

All arithmetic operators associate from left to right.

Operator	Operation	Precedence
`+`	addition	2
`-`	subtraction	2
`*`	multiplication	3
`/`	division	3
`div`	division, rounds down to nearest integer	3
`mod`	returns the remainder of integer division (cannot be used with floating points)	3

Operator `-` can be used as a prefix unary operator to change sign of a signed value. Unary prefix operator `+` can be used, but it doesn't affect the data.

For example: `b = -a`

Division by Zero

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`).

Relational Operators

Use relational operators to test equality or inequality of expressions. All relational operators return TRUE or FALSE.

All relational operators associate from left to right.

Operator	Operation	Precedence
=	equal	1
<>	not equal	1
>	greater than	1
<	less than	1
>=	greater than or equal	1
<=	less than or equal	1

Relational Operators in Expressions

Precedence of arithmetic and relational operators was designated in such a way to allow complex expressions without parentheses to have expected meaning:

```
a + 5 >= c - 1.0 / e    ' -> (a + 5) >= (c - (1.0 / e))
```

Bitwise Operators

Use the bitwise operators to modify the individual bits of numerical operands. Operands need to be either both signed or both unsigned.

Bitwise operators associate from left to right. The only exception is the bitwise complement operator not which associates from right to left.

Operator	Operation	Precedence
`and`	bitwise AND; returns 1 if both bits are 1, otherwise returns 0	3
`or`	bitwise (inclusive) OR; returns 1 if either or both bits are 1, otherwise returns 0	2
`xor`	bitwise exclusive OR (XOR); returns 1 if the bits are complementary, otherwise 0	2
`not`	bitwise complement (unary); inverts each bit	4
`<<`	bitwise shift left; moves the bits to the left, see below	3
`>>`	bitwise shift right; moves the bits to the right, see below	3

Bitwise operators `and`, `or`, and `xor` perform logical operations on appropriate pairs of bits of their operands. Operator `not` complements each bit of its operand. For example:

```
$1234 and $5678          ' equals $1230

' because ..

' $1234 : 0001 0010 0011 0100
' $5678 : 0101 0110 0111 1000
' ----------------------------
'  and  : 0001 0010 0011 0000

' .. that is, $1230
```

Similarly:

```
$1234 or  $5678      ' equals $567C
$1234 xor $5678      ' equals $444C
not $1234            ' equals $EDCB
```

Unsigned and Conversions

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

For example:

```
dim a as byte
dim b as word
...
  a = $AA
  b = $F0F0
  b = b and a
  ' a is extended with zeroes; b becomes $00A0
```

Signed and Conversions

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 zeroes 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).

For example:

```
dim a as byte
dim b as word
...
  a = -12
  b = $70FF
  b = b and a

  ' a is sign extended, upper byte is $FF;
  ' b becomes $70F4
```

Bitwise Shift Operators

Binary operators << and >> move the bits of the left operand for a number of positions specified by the right operand, to the left or right, respectively. Right operand has to be positive and less than 255.

With shift left (<<), left most bits are discarded, and “new” bits on the right are assigned zeroes. Thus, shifting unsigned operand to left by `n` positions is equivalent to multiplying it by 2^n if all the discarded bits are zero. This is also true for signed operands if all the discarded bits are equal to sign bit.

With shift right (>>), right most bits are discarded, and the “freed” bits on the left are assigned zeroes (in case of unsigned operand) or the value of the sign bit (in case of signed operand). Shifting operand to right by `n` positions is equivalent to dividing it by 2^n.

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

```
PORTB = word(temp >> 8)
```

EXPRESSIONS

An expression is a sequence of operators, operands, and punctuators that returns a value.

The primary expressions include: literals, variables, and function calls. From these, using operators, more complex expressions can be created. Formally, expressions are defined recursively: subexpressions can be nested up to the limits of memory.

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules that depend on the operators used, the presence of parentheses, and the data types of the operands. The precedence and associativity of the operators are summarized in Operator Precedence and Associativity. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by mikroBasic.

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

STATEMENTS

Statements define algorithmic actions within a program. Each statement needs to be terminated by a newline character (`CR/LF`).

The simplest statements include assignments, routine calls, and jump statements. These can be combined to form loops, branches, and other structured statements. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code.

Statements can be roughly divided into:

- `asm` Statement
- Assignment Statements
- Conditional Statements
- Iteration Statements (Loops)
- Jump Statements

asm Statement

mikroBasic allows embedding assembly in the source code by means of `asm` statement. Note that you cannot use numerals as absolute addresses for register variables in assembly instructions. You may use symbolic names instead (listing will display these names as well as addresses).

You can group assembly instructions with the `asm` keyword:

```
asm
   block of assembly instructions
end asm
```

BASIC comments are not allowed in embedded assembly code. Instead, you may use one-line assembly comments starting with semicolon. If you plan to use a certain BASIC variable in embedded assembly only, be sure to at least initialize it (assign it initial value) in BASIC code; otherwise, linker will issue an error. This does not apply to predefined globals such as PORTB.

Note: mikroBasic will not check if the banks are set appropriately for your variable. You need to set the banks manually in assembly code.

Migration from older compiler versions (v2.xx)

The syntax that is being used in the asm blocks is somewhat different than it has been in version 2. The differences are:

For example, for variable named :

_myVar, if it is global.
FARG_+XX, if it is local (this is myVar's actual position in the local function frame.
_myVar_L0(+XX), if it is a local static variable (+XX to access further individual bytes).
The only types whose name remains the same in asm as it is in Basic are constants, e.g. INTCON, PORTB, WREG, GIE, etc.
Accessing individual bytes is different as well. For example, if you have a global variable "g_var", that is of type long (i.e. 4 bytes), you are to access it like this:

```
MOVF   _g_var+0, 0    ;puts least-significant byte of g_var in W register
MOVF   _g_var+1, 0    ;second byte of _g_var; corresponds to Hi(g_var)
MOVF   _g_var+2, 0    ;Higher(g_var)
MOVF   _g_var+3, 0    ;Highest(g_var)
... etc.
```

Syntax for retrieving address of an object is different. For objects located in flash ROM:

```
MOVLW   #_g_var        ;first byte of address
MOVLW   @#_g_var        ;second byte of address
MOVLW   @@#_g_var        ;third byte of address
... and so on.
```

For objects located in RAM:

```
MOVLW   CONST1         ;first byte of address
MOVLW   @CONST1         ;second byte of address
... and so on.
```

Assignment Statements

Assignment statements have the form:

```
variable = expression
```

The statement evaluates the `expression` and assigns its value to the `variable`. All rules of the implicit conversion apply. `Variable` can be any declared variable or array element, and `expression` can be any expression.

Do not confuse the assignment with relational operator = which tests for equality. mikroBasic will interpret meaning of the character = from the context.

Conditional Statements

Conditional or selection statements select from alternative courses of action by testing certain values. There are two types of selection statements in mikroBasic: `if` and `select case`.

If Statement

Use `if` to implement a conditional statement. Syntax of `if` statement has the form:

```
if expression then
  statements
[ else
  other statements]
end if
```

When `expression` evaluates to true, `statements` execute. If `expression` is false, `other statements` execute. The `expression` must convert to a boolean type; otherwise, the condition is ill-formed. The `else` keyword with an alternate block of statements (`other statements`) is optional.

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.

Select Case Statement

Use the `select case` statement to pass control to a specific program branch, based on a certain condition. The `select case` statement consists of a selector expression (a condition) and a list of possible values. Syntax of `select case` statement is:

```
select case selector
  case value_1
    statements_1
  ...
  case value_n
    statements_n
  [ case else
    default_statements]
end select
```

The *`selector`* is an expression which should evaluate as integral value. The *`values`* can be literals, constants, or expressions. The *`statements`* can be any statements. The `else` clause is optional.

First, the *`selector`* expression (condition) is evaluated. The `select case` statement then compares it against all the available *`values`*. If the match is found, the *`statements`* following the match evaluate, and `select case` statement terminates. In case there are multiple matches, the first matching *`statement`* will be executed. If none of the *`values`* matches the *`selector`*, then the `default_statements` in the `else` clause (if there is one) are executed.

Here is a simple example of `select case` statement:

```
select case operator
  case "*"
    res = n1 * n2
  case "/"
    res = n1 / n2
  case "+"
    res = n1 + n2
  case "-"
    res = n1 - n2
  case else
    res = 0
    Inc(cnt)
end select
```

Also, you can group *`values`* together for a match. Simply separate the items by commas:

```
select case reg
  case 0
    opmode = 0
  case 1,2,3,4
    opmode = 1
  case 5,6,7
    opmode = 2
end select
```

Nested Case Statements

Note that `select case` statements can be nested – values are then assigned to the innermost enclosing `select case` statement.

Iteration Statements (Loops)

Iteration statements let you loop a set of statements. There are three forms of iteration statements in mikroBasic: `for`, `while`, and `do`.

You can use the statements `break` and `continue` to control the flow of a loop statement. The `break` terminates the statement in which it occurs, while `continue` begins executing the next iteration of the sequence.

For Statement

The `for` statement implements an iterative loop and requires you to specify the number of iterations. Syntax of `for` statement is:

```
for counter = initial_value to final_value [step step_value]
  statements
next counter
```

The *`counter`* is a variable which increases by *`step_value`* with each iteration of the loop. Parameter *`step_value`* is an optional integral value, and defaults to 1 if omitted. Before the first iteration, *`counter`* is set to the *`initial_value`* and will increment until it reaches (or exceeds) the *`final_value`*.

The *`initial_value`* and *`final_value`* should be expressions compatible with the *`counter`*; *`statements`* can be any statements that do not change the value of *`counter`*.

Note that parameter *`step_value`* may be negative, allowing you to create a countdown.

Here is an example of calculating scalar product of two vectors, `a` and `b`, of length `n`, using `for` statement:

```
s = 0
for i = 0 to n
  s = s + a[i] * b[i]
next i
```

The `for` statement results in an endless loop if the *`final_value`* equals or exceeds the range of *`counter`*'s type.

While Statement

Use the `while` keyword to conditionally iterate a statement. Syntax of `while` statement is:

```
while expression
  statements
wend
```

The *`statements`* are executed repeatedly as long as the *`expression`* evaluates true. The test takes place before the *`statements`* are executed. Thus, if *`expression`* evaluates false on the first pass, the loop does not execute.

Here is an example of calculating scalar product of two vectors, using the `while` statement:

```
while i < n
  s = s + a[i] * b[i]
  i = i + 1
wend
```

Do Statement

The `do` statement executes until the condition becomes true. Syntax of `do` statement is:

```
do
  statements
loop until expression
```

The *`statements`* are executed repeatedly until the *`expression`* evaluates true. The *`expression`* is evaluated *after* each iteration, so the loop will execute *`statements`* at least once.

Here is an example of calculating scalar product of two vectors, using the `do` statement:

```
do
  s = s + a[i] * b[i]
  i = i + 1
loop until i = n
```

Jump Statements

A jump statement, when executed, transfers control unconditionally. There are four such statements in mikroBasic: `break`, `continue`, `goto`, and `gosub`.

Break Statement

Sometimes, you might need to stop the loop from within its body. Use the `break` statement within loops to pass control to the first statement following the innermost loop (`for`, `while`, and `do`).

For example:

```
' Wait for CF card to be plugged; refresh every second
while true
  Lcd_Out(1,1,"No card inserted")
  if Cf_Detect() = 1 then
    break
  end if
  Delay_ms(1000)
wend

' Now we can work with CF card ...
Lcd_Out(1,1,"Card detected   ")
```

Continue Statement

You can use the `continue` statement within loops to “skip the cycle”:

- `continue` statement in `for` loop moves program counter to the line with keyword `for`; it does not change the loop counter,
- `continue` statement in `while` loop moves program counter to the line with loop condition (top of the loop),
- `continue` statement in `do` loop moves program counter to the line with loop condition (bottom of the loop).

Goto Statement

Use the `goto` statement to unconditionally jump to a local label — for more information, refer to Labels. Syntax of `goto` statement is:

```
goto label_name
```

This will transfer control to the location of a local label specified by *`label_name`*. The `goto` line can come before or after the label. It is not possible to jump into or out of routine.

You can use `goto` to break out from any level of nested control structures. Never jump into a loop or other structured statement, since this can have unpredictable effects. Use of `goto` statement is generally discouraged as practically every algorithm can be realized without it, resulting in legible structured programs. One possible application of `goto` statement is breaking out from deeply nested control structures.

Gosub Statement

Use the `gosub` statement to unconditionally jump to a local label — for more information, refer to Labels. Syntax of `gosub` statement is:

```
gosub label_name
...
label_name:
...
return
```

This will transfer control to the location of a local label specified by *`label_name`*. Also, the calling point is remembered. Upon encountering a `return` statement, program execution will continue with the next statement (line) after the `gosub`. The `gosub` line can come before or after the label.

It is not possible to jump into or out of routine by means of `gosub`. Never jump into a loop or other structured statement, since this can have unpredictable effects.

Note: Like with `goto`, use of `gosub` statement is generally discouraged. mikroBasic supports `gosub` only for the sake of backward compatibility. It is better to rely on functions and procedures, creating legible structured programs.

Exit Statement

The `exit` statement allows you to break out of a routine (function or procedure). It passes the control to the first statement following the routine call.

Here is a simple example:

```
sub procedure Proc1()
dim error as byte
  ... ' we're doing something here
  if error = TRUE then
    exit
  end if
  ... ' some code, which won't be executed if error is true
end sub
```

Note: If breaking out of a function, return value will be the value of the local variable `result` at the moment of exit.

COMPILER DIRECTIVES

Any line in source code with a leading `#` is taken as a compiler directive. The initial `#` can be preceded or followed by whitespace (excluding new lines). Compiler directives are not case sensitive.

You can use conditional compilation to select particular sections of code to compile while excluding other sections. All compiler directives must be completed in the source file in which they begun.

Directives #DEFINE and #UNDEFINE

Use directive `#DEFINE` to define a conditional compiler constant ("flag"). You can use any identifier for a flag, with no limitations. No conflicts with program identifiers are possible, as flags have a separate name space. Only one flag can be set per directive.

For example:

```
#DEFINE extended_format
```

Use `#UNDEFINE` to undefine ("clear") previously defined flag.

Directives #IF..THEN..#ELSE

Conditional compilation is carried out by `#IFDEF..THEN` directive. The `#IFDEF` tests whether a flag is currently defined or not; that is, whether a previous `#DEFINE` directive has been processed for that flag and is still in force.

Directive `#IFDEF..THEN` is terminated by the `#ENDIF` directive, and can have any number of `#ELSEIF` clauses and an optional `#ELSE` clause:

```
#IFDEF flag THEN
  block of code
...
[ #ELSE
  alternate block of code ]
#ENDIF
```

First, `#IFDEF` checks if *`flag`* is set by means of `#DEFINE`. If so, only *`block of code`* will be compiled. Otherwise, compiler will check flags *`flag_1`* .. *`flag_n`*, and execute the appropriate *`block of code i`*. Eventually, if none of the *`flags`* is set, *`alternate block of code`* in the `#ELSE` (if present) will be compiled.

The `#ENDIF` ends the conditional sequence. The result of the preceding scenario is that only one section of code (possibly empty) is passed on for further processing. The processed section can contain further conditional clauses, nested to any depth; each `#IFDEF` must be matched with a closing `#ENDIF`.

Here is a simple example:

```
' Uncomment the appropriate flag for your application:
'#DEFINE resolution8

#IFDEF resolution8 THEN
  ... ' code specific to 8-bit resolution
#ELSE
  ... ' default code
#ENDIF
```

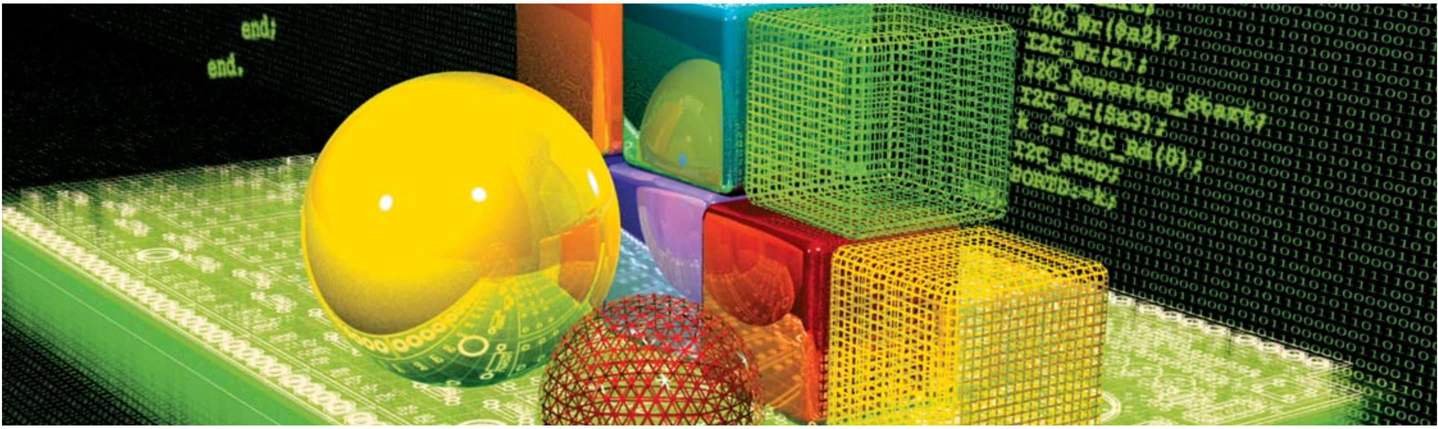
#I is compiler directive for inserting content of given file into place where this directive is called.

```
#I filename.txt
```

Predefined Flags

mikroBasic has several predefined flags for configuring hardware. These can be found in definition files (“def” folder), specifying hardware settings for individual chips. SFR are sorted under categories: `___SFR` (umbrella for all registers), `___CONFIG_OSC` (oscillator), `___CONFIG_WDT` (Watchdog timer), and `___CONFIG_BORPOR` (brown–out reset and power–on–timer).

CHAPTER 4



mikroBasic Libraries

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, Serial Ethernet, Toshiba GLCD, Port Expander, Serial GLCD, Serial Toshiba GLCD, Serial LCD (LCD8), numeric formatting, bit manipulation, and many other are included along with practical, ready-to-use code examples.

BUILT-IN ROUTINES

mikroBasic compiler provides a set of useful built-in utility functions. Built-in routines can be used in any part of the project.

Currently, mikroBasic includes the following built-in functions:

```
Inc
Dec
Chr
Ord
SetBit
ClearBit
TestBit
Lo
Hi
Higher
Highest
Swap
Clock_Khz
Clock_Mhz
Reset
ClrWdt
```

Inc

Prototype	`sub function Inc(dim byref par as longint) as longint`
Description	Increases parameter `par` by 1. Note that the function may be called as a self-contained statement. Function returns the value of increased parameter. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.

Dec

Prototype	`sub function Dec(dim byref par as longint) as longint`
Description	Decreases parameter `par` by 1. Note that the function may be called as a self-contained statement. Function returns the value of decreased parameter. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.

Chr

Prototype	`sub function Chr(dim code as byte) as char`
Returns	Returns a character associated with the specified character code.
Description	Function returns a character associated with the specified character `code`. Numbers from 0 to 31 are the standard nonprintable ASCII codes. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Example	`c = Chr(10)   ' returns a linefeed character`

Ord

Prototype	`sub function Ord(dim character as char) as byte`
Returns	ASCII code of the character.
Description	Function returns ASCII code of the character. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Example	`c = Ord("A")   ' returns 65`

SetBit

Prototype	`sub procedure SetBit(dim byref register as byte, dim rbit as byte)`
Description	Function sets the bit `rbit` of `register`. Parameter `rbit` needs to be a variable or literal with value 0..7. See Predefined globals and constants for more information on register identifiers. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Example	`SetBit(PORTB, 2)   ' Set RB2`

ClearBit

Prototype	`sub procedure ClearBit(dim byref register as byte, dim rbit as byte)`
Description	Function clears the bit `rbit` of `register`. Parameter `rbit` needs to be a variable or literal with value 0..7. See Predefined globals and constants for more information on register identifiers. This is an "inline" routine; code is generated in the place of the call, so the call doesn't count against the nested call limit.
Example	`ClearBit(PORTC, 7)   ' Clear RC7`

TestBit

Prototype	`sub function TestBit(dim register, rbit as byte) as byte`
Returns	If bit is set, returns 1, otherwise returns 0.
Description	Function tests if the bit `rbit` of `register` is set. If set, function returns 1, otherwise returns 0. Parameter `rbit` needs to be a variable or literal with value 0..7. See Predefined globals and constants for more information on register identifiers. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Example	`flag = TestBit(PORTE, 2)   ' 1 if RE2 is set, otherwise 0`

Lo

Prototype	`sub function Lo(dim number as byte..longint) as byte`
Returns	Returns the lowest 8 bits (byte) of `number`, bits 0..7.
Description	Function returns the lowest byte of `number`. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register.
Example	`= Lo(0x1AC30F4)   ' Equals 0xF4`

Hi

Prototype	`sub function Hi(dim number as word..longint) as byte`
Returns	Returns byte next to the lowest byte of `number`, bits 8..15.
Description	Function returns byte next to the lowest byte of `number`. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register.
Example	`a = Hi(0x1AC30F4)   ' Equals 0x30`

Higher

Prototype	`sub function Higher(dim number as longint) as byte`
Returns	Returns byte next to the highest byte of `number`, bits 16..23.
Description	Function returns byte next to the highest byte of `number`. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register.
Example	`a = Higher(0x1AC30F4)   ' Equals 0xAC`

Highest

Prototype	`sub function Highest(dim number as longint) as byte`
Returns	Returns the highest byte of `number`, bits 24..31.
Description	Function returns the highest byte of `number`. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register.
Example	`a = Highest(0x1AC30F4)   ' Equals 0x01`

Swap

Prototype	`sub function Swap(dim byref arg as byte) as byte`
Returns	Returns byte consisting of swapped nibbles.
Description	Swaps higher nibble (bits <7..4>) and lower nibble (bits <3..0>) of `arg`.
Example	`PORTB = 0xF0` `PORTA = Swap(PORTB)   ' PORTA = PORTB = 0x0F`

Clock_Khz

Prototype	`sub function Clock_Khz as word`
Returns	Device clock in KHz.
Description	Returns device clock in KHz, rounded to the nearest integer.
Example	`clk := Clock_Khz()`

Clock_Mhz

Prototype	`sub function Clock_Mhz as byte`
Returns	Device clock in MHz.
Description	Returns device clock in MHz, rounded to the nearest integer.
Example	`clk := Clock_Mhz()`

Reset

Prototype	`sub procedure Reset`
Description	This procedure is equal to assembler instruction **reset**. This procedure works only for P18.
Example	`Reset 'Resets the PIC MCU`

ClrWdt

Prototype	`sub procedure ClrWdt`
Description	This procedure is equal to assembler instruction **clrwdt**.
Example	`ClrWdt 'Clears PIC's WDT`

LIBRARY ROUTINES

mikroBasic provides a set of libraries which simplifies the initialization and use of PIC MCU and its modules. Library functions do not require any header files to be included; you can use them anywhere in your projects.
Currently available libraries include:

- ADC Library
- CAN Library
- CANSPI Library
- Compact Flash Library
- EEPROM Library
- Ethernet Library
- SPI Ethernet Library
- Flash Memory Library
- Graphic LCD Library
- T6963C Graphic LCD Library
- I²C Library
- Keypad Library
- LCD Library
- LCD8 Library
- Manchester Code Library
- Multi Media Card Library
- OneWire Library
- PS/2 Library
- PWM Library
- RS-485 Library
- Software I²C Library
- Software SPI Library
- Software UART Library
- Sound Library
- SPI Library
- USART Library
- USB HID Library
- Util Library
- Port Expander Library
- SPI GLCD Library
- SPI LCD Library
- SPI LCD8 Library
- SPI T6963C Graphic LCD Library
- Conversions Library
- Delays Library
- Math Library
- String Library

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.

Adc_Read

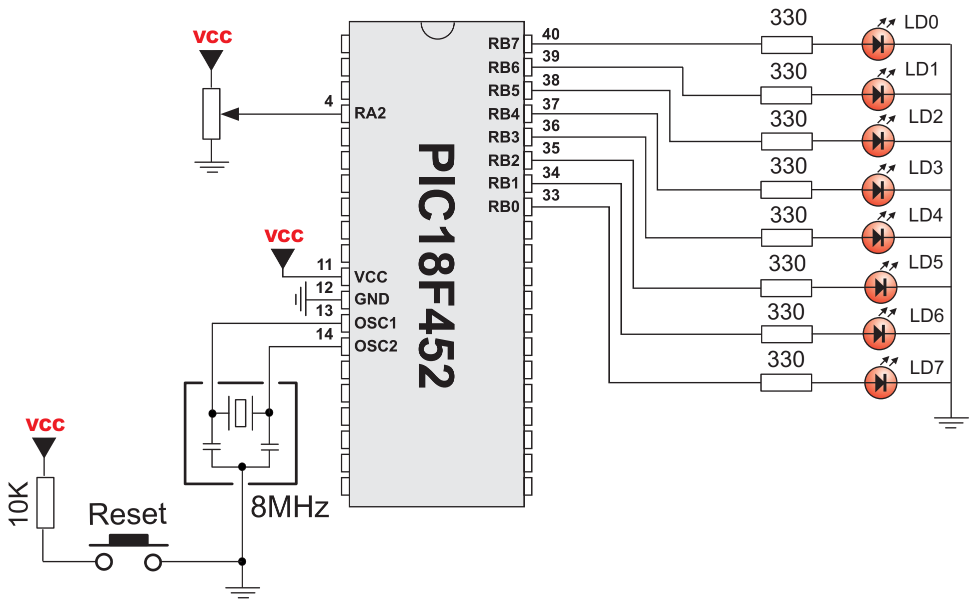
Prototype	`sub function Adc_Read(dim channel as byte) as word`
Returns	10-bit unsigned value read from the specified ADC channel.
Description	Initializes PIC's internal ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12TAD). RC sources typically have Tad 4uS. Parameter `channel` represents the channel from which the analog value is to be acquired. For channel-to-pin mapping please refer to documentation for the appropriate PIC MCU.
Requires	PIC MCU with built-in ADC module. You should consult the Datasheet documentation for specific device (most devices from PIC16/18 families have it). Before using the function, 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	`tmp = Adc_Read(1)   ' Read analog value from channel 1`

Library Example

This code snippet reads analog value from channel 2 and displays it on PORTD (lower 8 bits) and PORTB (2 most significant bits).

```
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 and RB6 are output
  TRISD  = $0                     ' PORTD is output

  while TRUE
    temp_res = Adc_Read(2)
    PORTD = temp_res              ' Send lower 8 bits to PORTD
    PORTB = word(temp_res >> 2)  ' Send 2 most significant bits to PORTB
  wend
end.
```

Hardware Connection



CAN Library

mikroBasic provides a library (driver) for working with the CAN module.

CAN is a very robust protocol that has error detection and signalling, self–checking and fault confinement. Faulty CAN data and remote frames are re-transmitted automatically, similar to the Ethernet.

Data transfer rates vary from up to 1 Mbit/s at network lengths below 40m to 250 Kbit/s at 250m cables, and can go even lower at greater network distances, down to 200Kbit/s, which is the minimum bitrate defined by the standard. Cables used are shielded twisted pairs, and maximum cable length is 1000m.

CAN supports two message formats:

Standard format, with 11 identifier bits, and
Extended format, with 29 identifier bits

Note: CAN routines are currently supported only by P18XXX8 PICmicros. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.

Note: Be sure to check CAN constants necessary for using some of the functions. See page 99.

Library Routines

```
CANSetOperationMode
CANGetOperationMode
CANInitialize
CANSetBaudRate
CANSetMask
CANSetFilter
CANRead
CANWrite
```

Following routines are for the internal use by compiler only:

```
RegsToCANID
CANIDToRegs
```

CANSetOperationMode

Prototype	`sub procedure CANSetOperationMode(dim mode, wait_flag as byte)`
Description	Sets CAN to requested mode, i.e. copies mode to CANSTAT. Parameter `mode` needs to be one of CAN_OP_MODE constants (see CAN constants). Parameter `wait_flag` needs to be either 0 or $FF: If set to $FF, this is a blocking call – the function won't "return" until the requested mode is set. If 0, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use function `CANGetOperationMode` to verify correct operation mode before performing mode specific operation.
Requires	CAN routines are currently supported only by P18XXX8 PICmicros. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	`CANSetOperationMode(CAN_MODE_CONFIG, $FF)`

CANGetOperationMode

Prototype	`sub function CANGetOperationMode as byte`
Returns	Current opmode.
Description	Function returns current operational mode of CAN module.
Requires	CAN routines are currently supported only by P18XXX8 PICmicros. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	`if CANGetOperationMode = CAN_MODE_NORMAL then ...`

CANInitialize

Prototype

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

Description

Initializes CAN. All pending transmissions are aborted. Sets all mask registers 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.
```

Parameters:

`SJW` as defined in 18XXX8 datasheet (1–4)
`BRP` as defined in 18XXX8 datasheet (1–64)
`PHSEG1` as defined in 18XXX8 datasheet (1–8)
`PHSEG2` as defined in 18XXX8 datasheet (1–8)
`PROPSEG` as defined in 18XXX8 datasheet (1–8)
`CAN_CONFIG_FLAGS` is formed from predefined constants (see CAN constants).

Requires

CAN must be in Config mode; otherwise the function will be ignored.

Example

```
init = 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   and
       CAN_CONFIG_LINE_FILTER_OFF
...
CANInitialize(1, 1, 3, 3, 1, init)    ' Initialize CAN
```

CANSetBaudRate

Prototype	`sub procedure CANSetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as byte)`
Description	Sets CAN baud rate. Due to complexity of CAN protocol, you cannot simply force a bps value. Instead, use this function when CAN is in Config mode. Refer to datasheet for details. Parameters: `SJW` as defined in 18XXX8 datasheet (1–4) `BRP` as defined in 18XXX8 datasheet (1–64) `PHSEG1` as defined in 18XXX8 datasheet (1–8) `PHSEG2` as defined in 18XXX8 datasheet (1–8) `PROPSEG` as defined in 18XXX8 datasheet (1–8) `CAN_CONFIG_FLAGS` is formed from predefined constants (see CAN constants)
Requires	CAN must be in Config mode; otherwise the function will be ignored.
Example	

```
init = 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   and
       CAN_CONFIG_LINE_FILTER_OFF
...
CANSetBaudRate(1, 1, 3, 3, 1, init)
```

CANSetMask

Prototype	`sub procedure CANSetMask(dim CAN_MASK as byte, dim value as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers. Parameters: `CAN_MASK` is one of predefined constant values (see CAN constants); `value` is the mask register value; `CAN_CONFIG_FLAGS` selects type of message to filter, either `CAN_CONFIG_XTD_MSG` or `CAN_CONFIG_STD_MSG`.
Requires	CAN must be in Config mode; otherwise the function will be ignored.

Example	`' Set all mask bits to 1, i.e. all filtered bits are relevant:`
	`CANSetMask(CAN_MASK_B1, -1, CAN_CONFIG_XTD_MSG)`
	`' Note that -1 is just a cheaper way to write $FFFFFFFF.`
	`' Complement will do the trick and fill it up with ones.`

CANSetFilter

Prototype	`sub procedure CANSetFilter(dim CAN_FILTER as byte, dim value as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers. Parameters: `CAN_MASK` is one of predefined constant values (see CAN constants); `value` is the filter register value; `CAN_CONFIG_FLAGS` selects type of message to filter, either `CAN_CONFIG_XTD_MSG` or `CAN_CONFIG_STD_MSG`.
Requires	CAN must be in Config mode; otherwise the function will be ignored.
Example	`' Set id of filter B1_F1 to 3:` `CANSetFilter(CAN_FILTER_B1_F1, 3, CAN_CONFIG_XTD_MSG)`

CANRead

Prototype	`sub function CANRead(dim byref id as longint, dim byref data as byte[8], dim byref datalen, CAN_RX_MSG_FLAGS as byte) as byte`
Returns	Message from receive buffer or zero if no message found.
Description	Function reads message from receive buffer. If at least one full receive buffer is found, it is extracted and returned. If none found, function returns zero. Parameters: `id` is message identifier; `data` is an array of bytes up to 8 bytes in length; `datalen` is data length, from 1–8; `CAN_RX_MSG_FLAGS` is value formed from constants (see CAN constants).
Requires	CAN must be in mode in which receiving is possible.
Example	`rcv = CANRead(id, data, len, 0)`

CANWrite

Prototype	`sub function CANWrite(dim id as longint, dim byref data as byte[8], dim datalen, CAN_TX_MSG_FLAGS as byte) as byte`
Returns	Returns zero if message cannot be queued (buffer full).
Description	If at least one empty transmit buffer is found, function sends message on queue for transmission. If buffer is full, function returns 0. Parameters: `id` is CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended); `data` is array of bytes up to 8 bytes in length; `datalen` is data length from 1–8; `CAN_TX_MSG_FLAGS` is value formed from constants (see CAN constants).
Requires	CAN must be in Normal mode.
Example	`tx = CAN_TX_PRIORITY_0 and CAN_TX_XTD_FRAME` `CANWrite(id, data, 2, tx)`

CAN Constants

There is a number of constants predefined in CAN library. To be able to use the library effectively, you need to be familiar with these. You might want to check the example at the end of the chapter.

CAN_OP_MODE

`CAN_OP_MODE` constants define CAN operation mode. Function `CANSetOperationMode` expects one of these as its argument:

```
const CAN_MODE_BITS         = $E0    ' Use it to access mode 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_CONFIG_FLAGS

`CAN_CONFIG_FLAGS` constants define flags related to CAN module configuration. Functions `CANInitialize` and `CANSetBaudRate` expect one of these (or a bitwise combination) as their argument:

```
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

' continues..
```

```
' ..continued

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
```

You may use bitwise AND to form config byte out of these values. For example:

```
init =  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  and CAN_CONFIG_LINE_FILTER_OFF
'...
CANInitialize(1, 1, 3, 3, 1, init)    ' initialize CAN
```

CAN_TX_MSG_FLAGS

`CAN_TX_MSG_FLAGS` are flags related to transmission of a CAN message:

```
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
```

You may use bitwise AND to adjust the appropriate flags. For example:

```
' form value to be used with CANSendMessage:
send_config = CAN_TX_PRIORITY_0 and CAN_TX_XTD_FRAME and
              CAN_TX_NO_RTR_FRAME
'...
CANSendMessage(id, data, 1, send_config)
```

CAN_RX_MSG_FLAGS

`CAN_RX_MSG_FLAGS` are flags related to reception of CAN message. If a particular bit is set; corresponding meaning is TRUE or else it will be FALSE.

```
const CAN_RX_FILTER_BITS    = $07 ' Use it 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 clear
const CAN_RX_INVALID_MSG    = $10 ' Set if invalid; else clear
const CAN_RX_XTD_FRAME      = $20 ' Set if XTD msg; else clear
const CAN_RX_RTR_FRAME      = $40 ' Set if RTR msg; else clear
const CAN_RX_DBL_BUFFERED   = $80 ' Set if msg was
                                  '   hardware double-buffered
```

You may use bitwise AND to adjust the appropriate flags. For example:

```
if MsgFlag and CAN_RX_OVERFLOW = 0 then
  ... ' Receiver overflow has occurred; previous message is lost.
```

CAN_MASK

`CAN_MASK` constants define mask codes. Function `CANSetMask` expects one of these as its argument:

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

CAN_FILTER

`CAN_FILTER` constants define filter codes. Function `CANSetFilter` expects one of these as its argument:

```
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
```

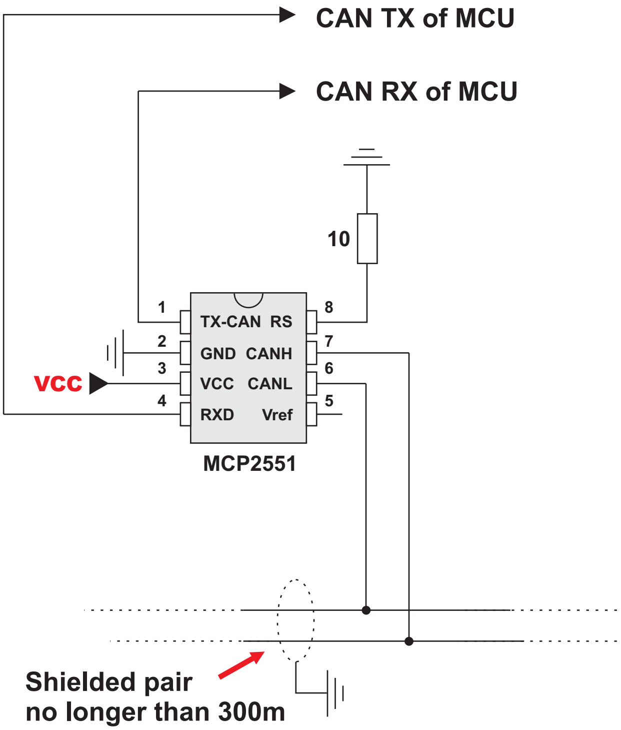
Library Example

The example demonstrates CAN protocol. It is a simple data exchange between 2 PIC's, where data is incremented upon each bounce. Data is printed on PORTC (lower byte) and PORTD (higher byte) for a visual check. Note that the data exchange doesn't start until you press a button; check the code below.

```
program can_test

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

sub function TestButton 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   ' RB0 is input
  PORTC = 0
  TRISC = 0
  PORTD = 0
  TRISD = 0
  aa     = 0
  aa1    = 0
  aa2    = 0

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

  ' Form value to be used with CANInitialize:
  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        and
        CAN_CONFIG_LINE_FILTER_OFF

' continues ..
```

```
' .. continued

  cont = true
  while cont
    cont = TestButton
  wend

  ' Initialize CAN
  CANInitialize( 1,1,3,3,1,aa)

  ' Set CONFIG mode
  CANSetOperationMode(CAN_MODE_CONFIG,TRUE)
  ID = -1

  ' Set all mask1 bits to ones
  CANSetMask(CAN_MASK_B1,ID,CAN_CONFIG_XTD_MSG)

  ' Set all mask2 bits to ones
  CANSetMask(CAN_MASK_B2,ID,CAN_CONFIG_XTD_MSG)

  ' Set id of filter B1_F1 to 3
  CANSetFilter(CAN_FILTER_B1_F1,3,CAN_CONFIG_XTD_MSG)

  ' Set NORMAL mode
  CANSetOperationMode(CAN_MODE_NORMAL,TRUE)

  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]                  ' Print data at PORTC
      data[ 0]  = data[ 0] +1
      id = 12111
      CANWrite(id, data, 1, aa1)        ' Send incremented data back
      if lenn = 2 then                  ' If message contains two data bytes,
        PORTD = data[ 1]                '   print second byte on at PORTD
      end if
    end if
  wend

end.
```

Hardware Connection



CANSPI Library

SPI module is available with a number of PICmicros. mikroBasic provides a library (driver) for working with the external CAN modules (such as MCP2515 or MCP2510) via SPI.

In mikroBasic, each routine of CAN library has its CANSPI counterpart with identical syntax. For more information on the Controller Area Network, consult the CAN Library. Note that the effective communication speed depends on the SPI, and is certainly slower than the “real” CAN.

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

Note: Be sure to check CAN constants necessary for using some of the functions. See page 99.

Note: `SPI_Init` must be called before initializing CANSPI.

Library Routines

```
CANSPISetOperationMode
CANSPIGetOperationMode
CANSPIInitialize
CANSPISetBaudRate
CANSPISetMask
CANSPISetFilter
CANSPIRead
CANSPIWrite
```

Following routines are for the internal use by compiler only:

```
RegsToCANSPIID
CANSPIIDToRegs
```

CANSPISetOperationMode

Prototype	`sub procedure CANSPISetOperationMode(dim mode, wait_flag as byte)`
Description	Sets CAN to requested mode, i.e. copies mode to CANSTAT. Parameter `mode` needs to be one of CAN_OP_MODE constants (see CAN constants, page 141). Parameter `wait_flag` needs to be either 0 or 0xFF: If set to 0xFF, this is a blocking call – the function won't "return" until the requested mode is set. If 0, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use function `CANSPIGetOperationMode` to verify correct operation mode before performing mode specific operation.
Requires	CANSPI functions are supported by any PIC MCU that has SPI interface on PORTC. Also, CS pin of MCP2510 or MCP2515 must be connected to RC0.
Example	`CANSPISetOperationMode(CAN_MODE_CONFIG, $FF)`

CANSPIGetOperationMode

Prototype	`sub function CANSPIGetOperationMode as byte`
Returns	Current opmode.
Description	Function returns current operational mode of CAN module.
Example	`if (CANSPIGetOperationMode = CAN_MODE_CONFIG) then ...`

CANSPIInitialize

Prototype

```
sub procedure CANSPIInitialize(dim SJW, BRP, PHSEG1, PHSEG2,
PROPSEG, CAN_CONFIG_FLAGS as byte, dim byref RstPort as byte, dim
RstPin as byte, dim byref CSPort as byte, dim CSPin as byte)
```

Description

Initializes CANSPI. All pending transmissions are aborted. Sets all mask registers 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) then
  ' Set all filters to XTD_MSG
else if ((config and CONFIG_VALID_STD_MSG) = 0) then
  ' Set all filters to STD_MSG
else
  ' Set half the filters to STD, and the rest to XTD_MSG
```

Parameters:

`SJW` as defined in 18XXX8 datasheet (1–4)
`BRP` as defined in 18XXX8 datasheet (1–64)
`PHSEG1` as defined in 18XXX8 datasheet (1–8)
`PHSEG2` as defined in 18XXX8 datasheet (1–8)
`PROPSEG` as defined in 18XXX8 datasheet (1–8)
`CAN_CONFIG_FLAGS` is formed from predefined constants (see CAN constants, page 99).

Requires

`SPI_Init` must be called before initializing CANSPI.
CANSPI must be in Config mode; otherwise the function will be ignored.

Example

```
init = 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 and
         CAN_CONFIG_LINE_FILTER_OFF
...
' initialize external CAN module
CANSPIInitialize(1,1,3,3,1,aa, PORTC, 2, PORTC, 0)
```

CANSPISetBaudRate

Prototype	`sub procedure CANSPISetBaudRate(dim SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS as byte)`
Description	Sets CANSPI baud rate. Due to complexity of CANSPI protocol, you cannot simply force a bps value. Instead, use this function when CANSPI is in Config mode. Refer to datasheet for details. Parameters: `SJW` as defined in 18XXX8 datasheet (1–4) `BRP` as defined in 18XXX8 datasheet (1–64) `PHSEG1` as defined in 18XXX8 datasheet (1–8) `PHSEG2` as defined in 18XXX8 datasheet (1–8) `PROPSEG` as defined in 18XXX8 datasheet (1–8) `CAN_CONFIG_FLAGS` is formed from predefined constants (see CAN constants)
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	

```
init = 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   and
       CAN_CONFIG_LINE_FILTER_OFF
...
CANSPISetBaudRate(1, 1, 3, 3, 1, init)
```

CANSPISetMask

Prototype	`sub procedure CANSPISetMask(dim CAN_MASK as byte, dim value as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers. Parameters: `CAN_MASK` is one of predefined constant values (see CAN constants); `value` is the mask register value; `CAN_CONFIG_FLAGS` selects type of message to filter, either `CAN_CONFIG_XTD_MSG` or `CAN_CONFIG_STD_MSG`.
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	

```
' Set all mask bits to 1, i.e. all filtered bits are relevant:
CANSPISetMask(CAN_MASK_B1, -1, CAN_CONFIG_XTD_MSG)

' Note that -1 is just a cheaper way to write $FFFFFFFF.
' Complement will do the trick and fill it up with ones
```

CANSPISetFilter

Prototype	`sub procedure CANSPISetFilter(dim CAN_FILTER as byte, dim val as longint, dim CAN_CONFIG_FLAGS as byte)`
Description	Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers. Parameters: `CAN_MASK` is one of predefined constant values (see CAN constants); `value` is the filter register value; `CAN_CONFIG_FLAGS` selects type of message to filter, either `CAN_CONFIG_XTD_MSG` or `CAN_CONFIG_STD_MSG`.
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	

```
' Set id of filter B1_F1 to 3:
CANSPISetFilter(CAN_FILTER_B1_F1, 3, CAN_CONFIG_XTD_MSG)
```

CANSPIRead

Prototype	`sub function CANSPIRead(dim byref id as longint, dim byref data as byte[ 8] , dim byref DataLen, CAN_RX_MSG_FLAGS as byte) as byte`
Returns	Message from receive buffer or zero if no message found.
Description	Function reads message from receive buffer. If at least one full receive buffer is found, it is extracted and returned. If none found, function returns zero. Parameters: `id` is message identifier; `data` is an array of bytes up to 8 bytes in length; `datalen` is data length, from 1–8; `CAN_RX_MSG_FLAGS` is value formed from constants (see CAN constants).
Requires	CANSPI must be in mode in which receiving is possible.
Example	`rcv = CANSPIRead(id, data, len, rx)`

CANSPIWrite

Prototype	`sub function CANSPIWrite(dim id as longint, dim byref data as byte[ 8] , dim datalen, CAN_TX_MSG_FLAGS as byte) as byte`
Returns	Returns zero if message cannot be queued (buffer full).
Description	If at least one empty transmit buffer is found, function sends message on queue for transmission. If buffer is full, function returns 0. Parameters: `id` is CANSPI message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended); `data` is array of bytes up to 8 bytes in length; `datalen` is data length from 1–8; `CAN_TX_MSG_FLAGS` is value formed from constants (see CAN constants).
Requires	CANSPI must be in Normal mode.
Example	`tx = CAN_TX_PRIORITY_0 and CAN_TX_XTD_FRAME` `CANSPIWrite(id, data, 2, tx)`

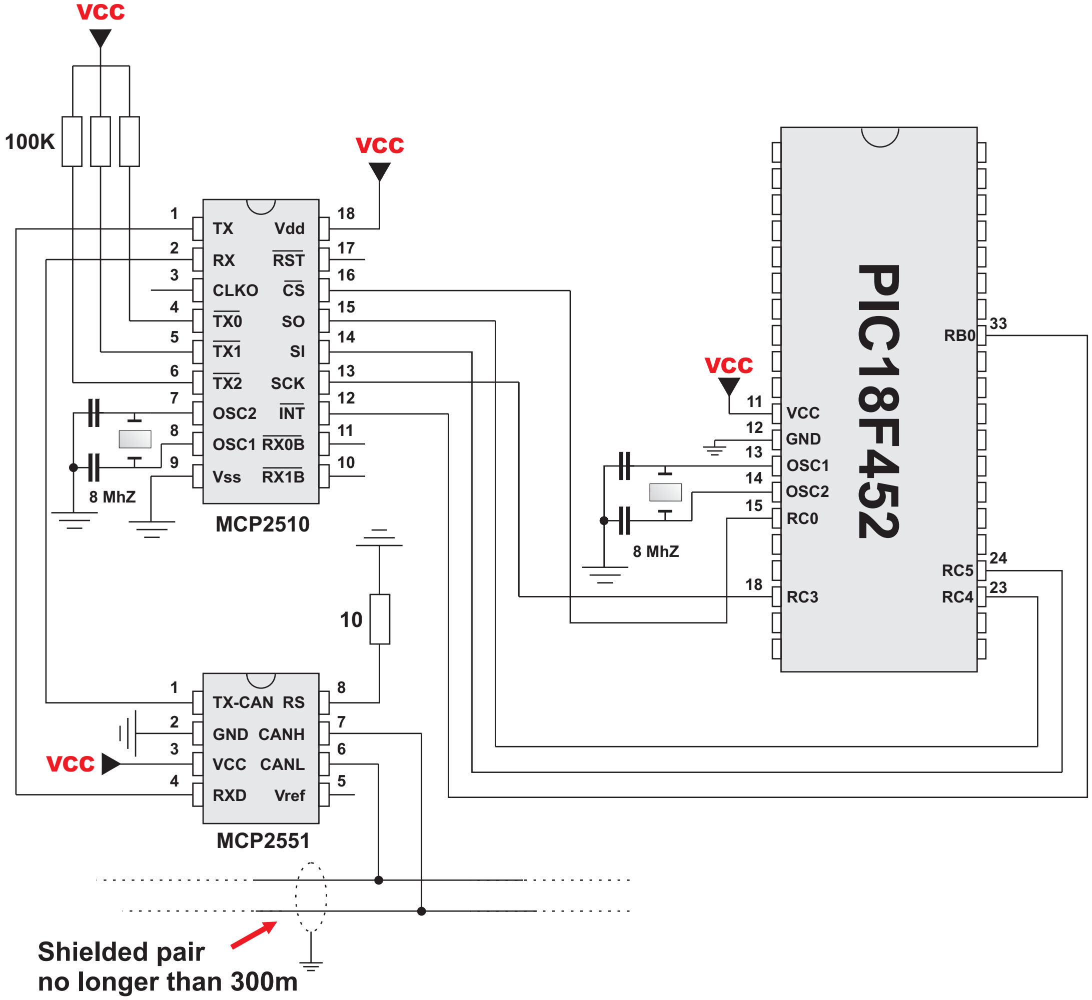
Library Example

The example demonstrates CANSPI protocol. It is a simple data exchange between 2 PIC's, where data is incremented upon each bounce. Data is printed on PORTC (lower byte) and PORTD (higher byte) for a visual check.

```
program canspi_test

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

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       ' RC0 is output
  PORTD  = 0
  TRISD  = 0        ' PORTD is output
  aa = 0
  aa1 = 0
  aa2 = 0

  ' Prepare flags for CANSPIinitialize
  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

  ' Activate MCP2510 chip
  PORTC.2 = 1

  ' Prepare flags for CANSPIWrite
  aa1 =  CAN_TX_PRIORITY_BITS and
         CAN_TX_FRAME_BIT     and
         CAN_TX_RTR_BIT



' continues ..
```

```
' ... continued

 Spi_Init ' initialize SPI

 ' Initialize MCP2510
 CANSPIInitialize(1,1,3,3,1,aa, PORTC, 2, PORTC, 0)


 ' Set Config mode
 CANSPISetOperationMode(CAN_MODE_CONFIG,true)
 ID = -1

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

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

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

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

 while true
   zr = CANSPIRead(id, Data, lenn, aa2)
   if (id = 12111) and zr then
      PORTD = $AA
      PORTB = data[0]
      Inc(data[0])
      id = 3
      Delay_ms(10)
      CANSPIWrite(id, data, 1, aa1)
     if lenn = 2 then
       PORTD = data[1]
     end if
   end if
 wend

end.
```

Hardware Connection



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.

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. 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.

Important! 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.

Library Routines

```
Cf_Init
Cf_Detect
Cf_Read_Init
Cf_Read_Byte
Cf_Write_Init
Cf_Write_Byte
Cf_Write_Sector
Cf_Read_Sector

Cf_Fat_Init
Cf_Fat_Assign
Cf_Fat_Reset
Cf_Fat_Read
Cf_Fat_Rewrite
Cf_Fat_Append
Cf_Fat_Delete
Cf_Fat_Write
Cf_Fat_Set_File_Date
Cf_Fat_Get_File_Date
Cf_Fat_Get_File_Size
Cf_Fat_Get_Swap_File
```

Function `Cf_Set_Reg_Adr` is for compiler internal purpose only.

Cf_Init

Prototype	`sub procedure Cf_Init(dim byref ctrlport, dataport as byte)`
Description	Initializes ports appropriately for communication with CF card. Specify two different ports: `ctrlport` and `dataport`.
Example	`Cf_Init(PORTB, PORTD)`

Cf_Detect

Prototype	`sub function Cf_Detect as byte`
Returns	Returns 1 if CF is present, otherwise returns 0.
Description	Checks for presence of CF card on `ctrlport`.

Example

```
' Wait until CF card is inserted:
do
  nop
loop until Cf_Detect = 1
```

Cf_Read_Init

Prototype	`sub procedure Cf_Read_Init(dim address as longint, dim sectcnt as byte)`
Description	Initializes CF card for reading. Parameters: `ctrlport` is control port, `dataport` is data port , `address` specifies sector address from where data will be read, and `sectcnt` is total number of sectors prepared for read operation.
Requires	Ports must be initialized. See `Cf_Init`.
Example	`Cf_Read_Init(590, 1)`

Cf_Read_Byte

Prototype	`sub function Cf_Read_Byte as byte`
Returns	Returns byte from CF.
Description	Reads one byte from CF.
Requires	CF must be initialized for read operation. See `Cf_Read_Init`.
Example	`PORTC = Cf_Read_Byte`

Cf_Write_Init

Prototype	`sub procedure Cf_Write_Init(dim address as longint, dim sectcnt as byte)`
Description	Initializes CF card for writing. Parameter `ctrlport` is control port, `dataport` is data port , `address` specifies sector address where data will be stored, and `sectcnt` is total number of sectors prepared for write operation.
Requires	Ports must be initialized. See `Cf_Init`.
Example	`Cf_Write_Init(590, 1)`

Cf_Write_Byte

Prototype	`sub procedure Cf_Write_Byte(dim data as byte)`
Description	Writes one byte (`data`) to CF. All 512 bytes are transferred to a buffer.
Requires	CF must be initialized for write operation. See `Cf_Write_Init`.
Example	`Cf_Write_Byte(100)`

Cf_Write_Sector

Prototype	`sub function Cf_Write_Sector(dim sector as longint, dim byref data as byte[512]) as byte`
Returns	Returns 0 if write was successful; returns 1 if there was an error in sending write command; returns 2 if there was an error in writing.
Description	Function writes 512 bytes of data to CF card at sector address sector. Function returns 0 if write was successful, or 1 if there was an error in sending write command, or 2 if there was an error in writing.
Requires	Ports must be initialized. See `Cf_Init`.
Example	`error = Cf_Write_Sector(sector, data)`

Cf_Read_Sector

Prototype	`sub function Cf_Read_Sector(dim sector as longint, dim byref data as byte[512]) as byte`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads one sector (512 bytes) from CF card at sector address sector. Read data is stored in the array data. Function returns 0 if read was successful, or 1 if an error occurred.
Requires	Ports must be initialized. See `Cf_Init`.
Example	`error = Cf_Read_Sector(sector, data)`

Cf_Fat_Init

Prototype	**sub function** Cf_Fat_Init(**dim byref** ctrlPort **as byte, dim byref** dataPort **as byte**) **as byte**
Returns	Returns 0 if initialization is successful, 1 if boot sector was not found and 255 if card was not detected.
Description	Initializes ports appropriately for FAT operations with CF card. Specify two different ports: ctrlport and dataport.
Requires	Nothing.
Example	`CF_Fat_Init(PORTD,PORTC)`

Cf_Fat_Assign

Prototype	**sub function** Cf_Fat_Assign(**dim byref** filename **as array**[12] **of char, dim** create_file **as byte**)**as byte**
Returns	"1" is file is present(or file isn't present but new file is created), or "0" if file isn't present and no new file is created.
Description	Assigns file for FAT operations. If file isn't present, function creates new file with given filename. filename parameter is name of file (filename must be in format 8.3 UPPER-CASE). create_file is a parameter for creating new files. if create_file if different from 0 then new file is created (if there is no file with given filename).
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init.
Example	`Cf_Fat_Assign('MIKROELE.TXT',1)`

Cf_Fat_Reset

Prototype	**sub procedure** Cf_Fat_Reset(**dim byref** size **as longint**)
Returns	Size of file in bytes. Size is stored on address of input variable.
Description	Opens assigned file for reading.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	`Cf_Fat_Reset(size)`

Cf_Fat_Read

Prototype	`sub procedure Cf_Fat_Read(dim byref bdata as byte)`
Returns	Nothing.
Description	Reads data from file. bdata is data read from file.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign. File must be open for reading. See Cf_Fat_Reset.
Example	`Cf_Fat_Read(character)`

Cf_Fat_Rewrite

Prototype	`sub procedure Cf_Fat_Rewrite`
Returns	Nothing.
Description	Rewrites assigned file.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	`Cf_Fat_Rewrite`

Cf_Fat_Append

Prototype	`sub procedure Cf_Fat_Append`
Returns	Nothing.
Description	Opens file for writing. This procedure continues writing from the last byte in file.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	`Cf_Fat_Append`

Cf_Fat_Delete

Prototype	`sub procedure Cf_Fat_Delete`
Returns	Nothing.
Description	Deletes file from CF.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	`Cf_Fat_Delete`

Cf_Fat_Write

Prototype	`sub procedure Cf_Fat_Write(dim byref fdata as array[512] of byte, dim data_len as word)`
Returns	Nothing.
Description	Writes data to CF.fdata parameter is data written to CF. data_len number of bytes that is written to CF.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.
Example	`Cf_Fat_Write(file_contents, 42)  ' write data to the assigned file`

Cf_Fat_Set_File_Date

Prototype	`sub procedure Cf_Fat_Set_File_Date(dim year as word, dim month, day, hours, mins, seconds as byte)`
Returns	Nothing.
Description	Sets time attributes of file.You can set file year, month, day. hours, mins, seconds.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.
Example	`Cf_Fat_Set_File_Date(2005,9,30,17,41,0)`

Cf_Fat_Get_File_Date

Prototype	`sub procedure Cf_Fat_Get_File_Date(dim byref year as word, dim byref month as word, dim byref day as word, dim byref hours as word, dim byref mins as word)`
Returns	Nothing.
Description	Reads time attributes of file.You can read file year, month, day. hours, mins.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	`Cf_Fat_Get_File_Date(year, month, day, hours, mins)`

Cf_Fat_Get_File_Size

Prototype	`sub function Cf_Fat_Get_File_Size as longint`
Returns	Size of file in bytes.
Description	This function returns size of file in bytes.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	`Cf_Fat_Get_File_Size`

Cf_Fat_Get_Swap_File

Prototype	`sub function Cf_Fat_Get_Swap_File(dim sectors_cnt as longint)as longint`
Returns	No. of start sector for the newly created swap file, if swap file was created; otherwise, the function returns zero.
Description	This function is used to create a swap file on the CF media. It accepts as sectors_cnt argument the number of consecutive sectors that user wants the swap file to have. During its execution, the function searches for the available consecutive sectors, their number being specified by the sectors_cnt argument. If there is such space on the media, the swap file named MIKROSWP.SYS is created, and that space is designated (in FAT tables) to it. The attributes of this file are: system, archive and hidden, in order to distinct it from other files. If a file named MIKROSWP.SYS already exists on the media, this function deletes it upon creating the new one. The purpose of the swap file is to make reading and writing to CF media as fast as possible, without potentially damaging the FAT system. Swap file can be considered as a "window" on the media where user can freely write/read the data, in any way (s)he wants to. Its main purpose in mikroBasic library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a "normal" file, and formatted in the most suitable way.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init.
Example	

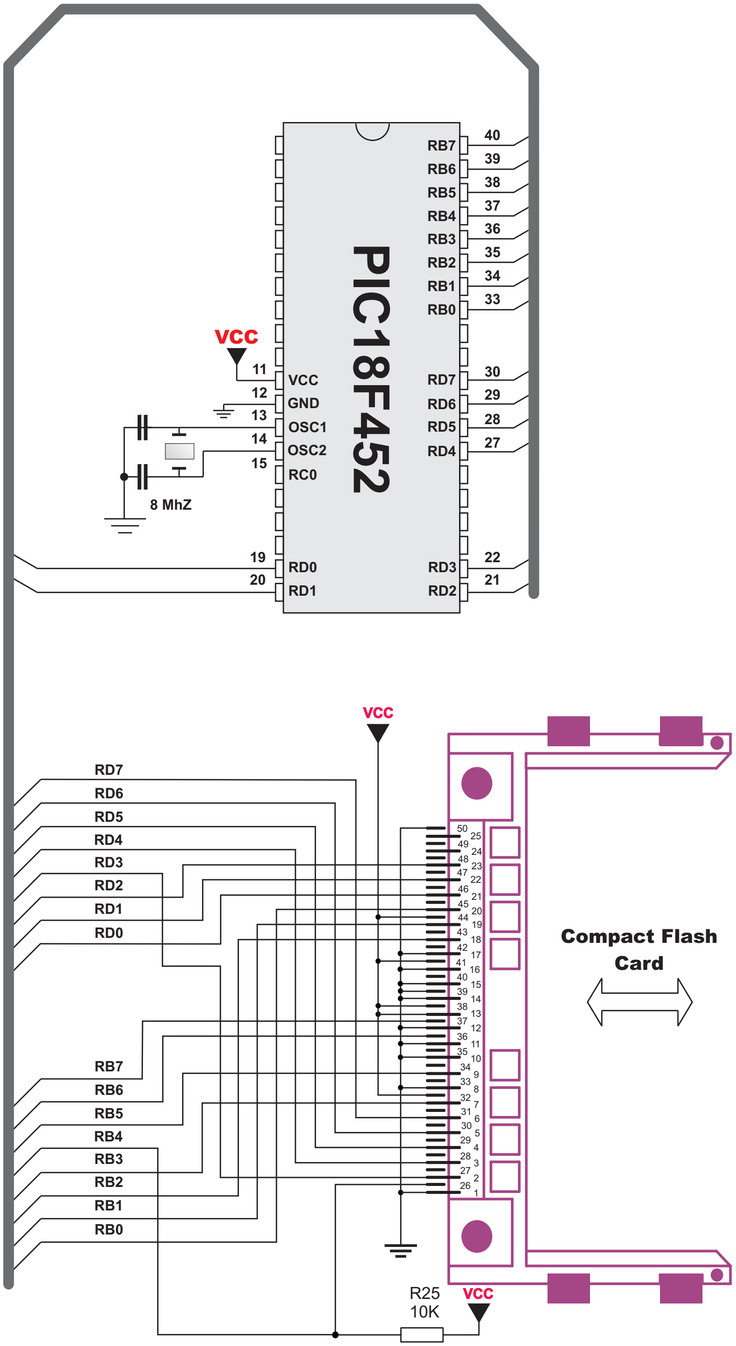
```
'-------------- Tries to create a swap file, whose size will be
'at least 100 sectors.
'If it succeeds, it sends the No. of start sector over USART

sub procedure C_Create_Swap_File
  size = Cf_Fat_Get_Swap_File(100)
  if (size) then
    Usart_Write($AA)
    Usart_Write(Lo(size))
    Usart_Write(Hi(size))
    Usart_Write(Higher(size))
    Usart_Write(Highest(size))
    Usart_Write($AA)
   end if
end sub
```

Library Examples

The following example writes 512 bytes at sector no.590, and then reads the data and prints on PORTC for a visual check.

```
program Cf_example
dim i as word
dim temp, k as longint

main:
  TRISC = 0                             ' PORTC is output
  Cf_Init(PORTB, PORTD)                 ' Initialize ports

  do
    nop
  loop until Cf_Detect = true           ' Wait until CF card is inserted

  Delay_ms(500)
  Cf_Write_Init(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(i + 11)
  next i

  PORTC = $FF
  Delay_ms(1000)
  Cf_Read_Init(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                ' Read byte and display on PORTC
    Delay_ms(1000)
  next i
end.
```

HW Connection



EEPROM Library

EEPROM data memory is available with a number of PICmicros. mikroBasic includes library for comfortable work with EEPROM.

Library Routines

`Eeprom_Read`
`Eeprom_Write`

Eeprom_Read

Prototype	`sub function EEprom_read(dim Address as word) as byte`
Returns	Returns byte from specified address.
Description	Reads data from specified address. Parameter address is of byte type, which means it can address only 1024 locations. For PIC18 micros with more EEPROM data locations, it is programmer's responsibility to set EEADRH register appropriately.
Requires	Requires EEPROM module. Ensure minimum 20ms delay between successive use of routines `Eeprom_Write` and `Eeprom_Read`. Although PIC will write the correct value, `Eeprom_Read` might return an undefined result.
Example	`take = Eeprom_Read($3F)`

Eeprom_Write

Prototype	`sub procedure EEprom_write(dim Address as word, dim Data as byte)`
Description	Writes data to specified address. Parameter address is of byte type, which means it can address only 1024 locations. For PIC18 micros with more EEPROM data locations, it is programmer's responsibility to set EEADRH register appropriately. Be aware that all interrupts will be disabled during execution of `Eeprom_Write` routine (GIE bit of INTCON register will be cleared). Routine will set this bit on exit.
Requires	Requires EEPROM module. Ensure minimum 20ms delay between successive use of routines `Eeprom_Write` and `Eeprom_Read`. Although PIC will write the correct value, `Eeprom_Read` might return an undefined result.
Example	`Eeprom_Write($32)`

Library Example

The example writes values at 20 successive locations of EEPROM. Then, it reads the written data and prints on PORTB for a visual check.

```
program eeprom_test
dim i, j as char

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.
```

Ethernet Library

This library is designed to simplify handling of the underlying hardware (RTL8019AS). However, certain level of knowledge about the Ethernet and Ethernet-based protocols (ARP, IP, TCP/IP, UDP/IP, ICMP/IP) is expected from the user. The Ethernet is a high–speed and versatile protocol, but it is not a simple one. Once you get used to it, however, you will make your favorite PIC available to a much broader audience than you could do with the RS232/485 or CAN.

Library Routines

- `Eth_Init`
- `Eth_Set_Ip_Address`
- `Eth_Inport`
- `Eth_Scan_For_Event`
- `Eth_Get_Ip_Hdr_Len`
- `Eth_Load_Ip_Packet`
- `Eth_Get_Hdr_Chksum`
- `Eth_Get_Source_Ip_Address`
- `Eth_Get_Dest_Ip_Address`
- `Eth_Arp_Response`
- `Eth_Get_Icmp_Info`
- `Eth_Ping_Response`
- `Eth_Get_Udp_Source_Port`
- `Eth_Get_Udp_Dest_Port`
- `Eth_Get_Udp_Port`
- `Eth_Set_Udp_Port`
- `Eth_Send_Udp`
- `Eth_Load_Tcp_Header`
- `Eth_Get_Tcp_Hdr_Offset`
- `Eth_Get_Tcp_Flags`
- `Eth_Set_Tcp_Data`
- `Eth_Tcp_Response`

Eth_Init

Prototype	`sub procedure Eth_Init(dim byref addrP, dataP, ctrlP as byte, dim pinReset, pinIOW, pinIOR as byte)`
Description	Performs initialization of Ethernet card and library. This includes: - Setting of control and data ports; - Initialization of the Ethernet card (also called the Network Interface Card, or NIC); - Retrieval and local storage of the NIC's hardware (MAC) address; - Putting the NIC into the LISTEN mode. Parameter `addrP` is a pointer to address port, which handles the addressing lines. Parameter `dataP` is pointer to data port. Parameter `ctrlP` is the control port. Parameter `pinReset` is the reset/enable pin for the ethernet card chip (on control port). Parameter `pinIOW` is the I/O Write request control pin. Parameter `pinIOR` is the I/O read request control pin.
Requires	As specified for the entire library.
Example	`Eth_Init(PORTB, PORTD, PORTE, 2, 1, 0)`

Eth_Set_Ip_Address

Prototype	`sub procedure Eth_Set_Ip_Address(dim ip1, ip2, ip3, ip4 as byte)`
Description	Sets the IP address of the connected and initialized Ethernet network card. The arguments are the IP address numbers, in IPv4 format (e.g. 127.0.0.1).
Requires	This procedure should be called immediately after the NIC initialization (see `Eth_Init`). You can change your IP address at any time, anywhere in the code.
Example	`' Set IP address 192.168.20.25` `Eth_Set_Ip_Address(192, 168, 20, 25)`

Eth_Set_Inport

Prototype	`sub function Eth_Inport(dim address as byte) as byte`
Returns	One byte from the specified address.
Description	Retrieves a byte from the specified address of the Ethernet card chip.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`.
Example	`udp_length = udp_length or Eth_Inport(NIC_DATA)`

Eth_Scan_For_Event

Prototype	`sub function Eth_Scan_For_Event(dim byref next_ptr as byte) as word`
Returns	Type of the ethernet packet received. Two types are distinguished: `ARP` (MAC-IP address data request) and `IP` (Internet Protocol).
Description	Retrieves sender's MAC (hardware) address and type of the packet received. The function argument is an (internal) pointer to the next data packet in RTL8019's buffer, and is of no particular importance to the end user.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`. Also, the function must be called in a proper sequence, i.e. right after the card init and IP address/UDP port init.

Example

```
while TRUE
  event_type = Eth_Scan_For_Event(next_ptr)   ' Scan for event
  select case event_type
    case ARP
      Arp_Event()   ' Some event handler
    case IP
      Ip_Event()    ' Some event handler
  end select
  Eth_Outport(CR, $22)
  Eth_Outport(BNDRY, next_ptr)
wend
```

Eth_Get_Ip_Hdr_Len

Prototype	`sub function Eth_Get_Ip_Hdr_Len as byte`
Returns	Header length of the received IP packet.
Description	Function returns header length of the received IP packet. Before other data based upon the IP protocol (TCP, UDP, ICMP) can be analyzed, the sub-protocol data must be properly loaded from the received IP packet.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`. The function must be called in a proper sequence, i.e. immediately after determining that the packet received is the IP packet.
Example	`' Receive IP Header` `opt_len = Eth_Get_Ip_Hdr_Len() - 20`

Eth_Load_Ip_Packet

Prototype	`sub procedure Eth_Load_Ip_Packet`
Description	Loads various IP packet data into PIC's Ethernet variables.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`. Also, a proper sequence of calls must be obeyed (see the `Ip_Event` function in the supplied Ethernet example).
Example	`Eth_Load_Ip_Packet()`

Eth_Get_Hdr_Chksum

Prototype	`sub procedure Eth_Get_Hdr_Chksum`
Description	Loads and returns the header checksum of the received IP packet.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`. Also, a proper sequence of calls must be obeyed (see the `Ip_Event` function in the supplied Ethernet example).
Example	`Eth_Get_Hdr_Chksum()`

Eth_Get_Source_Ip_Address

Prototype	`sub procedure Eth_Get_Source_Ip_Address`
Description	Loads and returns the IP address of the sender of the received IP packet.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`. Also, a proper sequence of calls must be obeyed (see the `Ip_Event` function in the supplied Ethernet example).
Example	`Eth_Get_Source_Ip_Address()`

Eth_Get_Dest_Ip_Address

Prototype	`sub procedure Eth_Get_Dest_Ip_Address`
Description	Loads the IP address of the received IP packet for which the packet is designated.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`. Also, a proper sequence of calls must be obeyed (see the `Ip_Event` function in the supplied Ethernet example).
Example	`Eth_Get_Dest_Ip_Address()`

Eth_Arp_Response

Prototype	**`sub procedure`** `Eth_Arp_Response`
Description	An automated ARP response. User should simply call this function once he detects the ARP event on the NIC.
Requires	As specified for the entire library.
Example	`Eth_Arp_Response()`

Eth_Get_Icmp_Info

Prototype	**`sub procedure`** `Eth_Get_Icmp_Info`
Description	Loads ICMP protocol information (from the header of the received ICMP packet) and stores it to the PIC's Ethernet variables.
Requires	The card (NIC) must be properly initialized. See `Eth_Init`. Also, this function must be called in a proper sequence, and before the `Eth_Ping_Response`.
Example	`Eth_Get_Icmp_Info()`

Eth_Ping_Response

Prototype	**`sub procedure`** `Eth_Ping_Response`
Description	An automated ICMP (Ping) response. User should call this function when answerring to an ICMP/IP event.
Requires	As specified for the entire library.
Example	`Eth_Ping_Response()`

Eth_Get_Udp_Source_Port

Prototype	`sub function Eth_Get_Udp_Source_Port as word`
Returns	Returns the source port (socket) of the received UDP packet.
Description	The function returns the source port (socket) of the received UDP packet. After the reception of valid IP packet is detected and its type is determined to be UDP, the UDP packet header must be interpreted. UDP source port is the first data in the UDP header.
Requires	This function must be called in a proper sequence, i.e. immediately after interpretation of the IP packet header (at the very beginning of UDP packet header retrieval).
Example	`udp_source_port = Eth_Get_Udp_Source_Port()`

Eth_Get_Udp_Dest_Port

Prototype	`sub function Eth_Get_Udp_Dest_Port as word`
Returns	Returns the destination port of the received UDP packet.
Description	The function returns the destination port of the received UDP packet. The second information contained in the UDP packet header is the destination port (socket) to which the packet is targeted.
Requires	This function must be called in a proper sequence, i.e. immediately after calling the `Eth_Get_Udp_Source_Port` function.
Example	`udp_dest_port = Eth_Get_Udp_Dest_Port()`

Eth_Get_Udp_Port

Prototype	`sub function Eth_Get_Udp_Port as byte`
Returns	Returns the UDP port (socket) number that is set for the PIC’s Ethernet card.
Description	The function returns the UDP port (socket) number that is set for the PIC's Ethernet card. After the UDP port is set at the beginning of the session (`Eth_Set_Udp_Port`), its number is later used to test whether the received UDP packet is targeted at the port we are using.
Requires	The network card must be properly initialized (see `Eth_Init`), and the UDP port propely set (see `Eth_Set_Udp_Port`). This library currently supports working with only one UDP port (socket) at a time.
Example	`if udp_dest_port = Eth_Get_Udp_Port() then` `  ... ' Respond to action`

Eth_Set_Udp_Port

Prototype	`sub procedure Eth_Set_Udp_Port(dim udp_port as word)`
Description	Sets up the default UDP port, which will handle user requests. The user can decide, upon receiving the UDP packet, which port was this packet sent to, and whether it will be handled or rejected.
Requires	As specified for the entire library.
Example	`Eth_Set_Udp_Port(10001)`

Eth_Send_Udp

Prototype	`sub procedure Eth_Send_Udp(dim msg as string[16])`
Description	Sends the prepared UDP message (`msg`), of up to 16 bytes (characters). Unlike ICMP and TCP, the UDP packets are generally not generated as a response to the client request. UDP provides no guarantees for message delivery and sender retains no state on UDP messages once sent onto the network. This is why UDP packets are simply sent, instead of being a response to someone's request.
Requires	As specified for the entire library. Also, the message to be sent must be formatted as a null-terminated string. The message length, including the trailing "0", must not exceed 16 characters.
Example	`Eth_Send_Udp(udp_tx_message)`

Eth_Load_Tcp_Header

Prototype	`sub procedure Eth_Load_Tcp_Header`
Description	Loads various TCP Header data into PIC's Ethernet variables.
Requires	This function must be called in a proper sequence, i.e. immediately after retrieving the source and destination port (socket) of the TCP message.
Example	

```
tcp_source_port = Eth_Inport(NIC_DATA) << 8   ' get src port
tcp_source_port = tcp_source_port or Eth_Inport(NIC_DATA)
tcp_dest_port = Eth_Inport(NIC_DATA) << 8   ' get dest port
tcp_dest_port = tcp_dest_port or Eth_Inport(NIC_DATA)

' We only respond to port 80 (HTML requests)
if tcp_dest_port = 80 then
  ' retrieve TCP Header data (most of it)
  Eth_Load_Tcp_Header()
  '...
end if
```

Eth_Get_Tcp_Hdr_Offset

Prototype	`sub function Eth_Get_Tcp_Hdr_Offset as byte`
Returns	Returns the length (or offset) of the TCP packet header in bytes.
Description	The function returns the length (or offset) of the TCP packet header in bytes. Upon receiving a valid TCP packet, its header is to be analyzed in order to respond properly (e.g. respond to other's request, merge several packets into the message, etc.). The header length is important to know in order to be able to extract the information contained in it.
Requires	This function must be called after the `Eth_Load_Tcp_Header`, since it initializes the private variables used for this function.
Example	`' calculate offset (TCP header length)` `tcp_options = Eth_Get_Tcp_Hdr_Offset() - 20`

Eth_Get_Tcp_Flags

Prototype	`sub function Eth_Get_Tcp_Flags as byte`
Returns	Returns the flags data from the header of the received TCP packet.
Description	The function returns the flags data from the header of the received TCP packet. TCP flags show various information, e.g. SYN (syncronize request), ACK (acknowledge receipt), and similar. It is upon these flags that, for example, a proper HTTP communication is established.
Requires	This function must be called after the `Eth_Load_Tcp_Header`, since it initializes the private variables used for this function.
Example	`flags = Eth_Get_Tcp_Flags()`

Eth_Set_Tcp_Data

Prototype	`sub procedure Eth_Set_Tcp_Data(const data as ^byte)`
Description	Prepares data to be sent on HTTP request. This library can handle only HTTP requests, so sending other TCP-based protocols, such as FTP, will cause an error. Note that TCP/IP was not designed with 8-bit MCU's in mind, so be gentle with your HTTP requests.
Requires	As specified for the entire library.
Example	

```
' Let's prepare a simple HTML page in our string:
const httpPage1 =
  "HTTP/1.0 200 OK" + Chr(13) + Chr(10) +
  "Content-type: text/html" + Chr(13) + Chr(10) +
  "<html>" + Chr(10) + "<body>" + Chr(10) +
  "<h1>Hello world!</h1>" + Chr(10) +
  "</body>" + Chr(10) + "</html>"
'...
Eth_Set_Tcp_Data(@httpPage1)
```

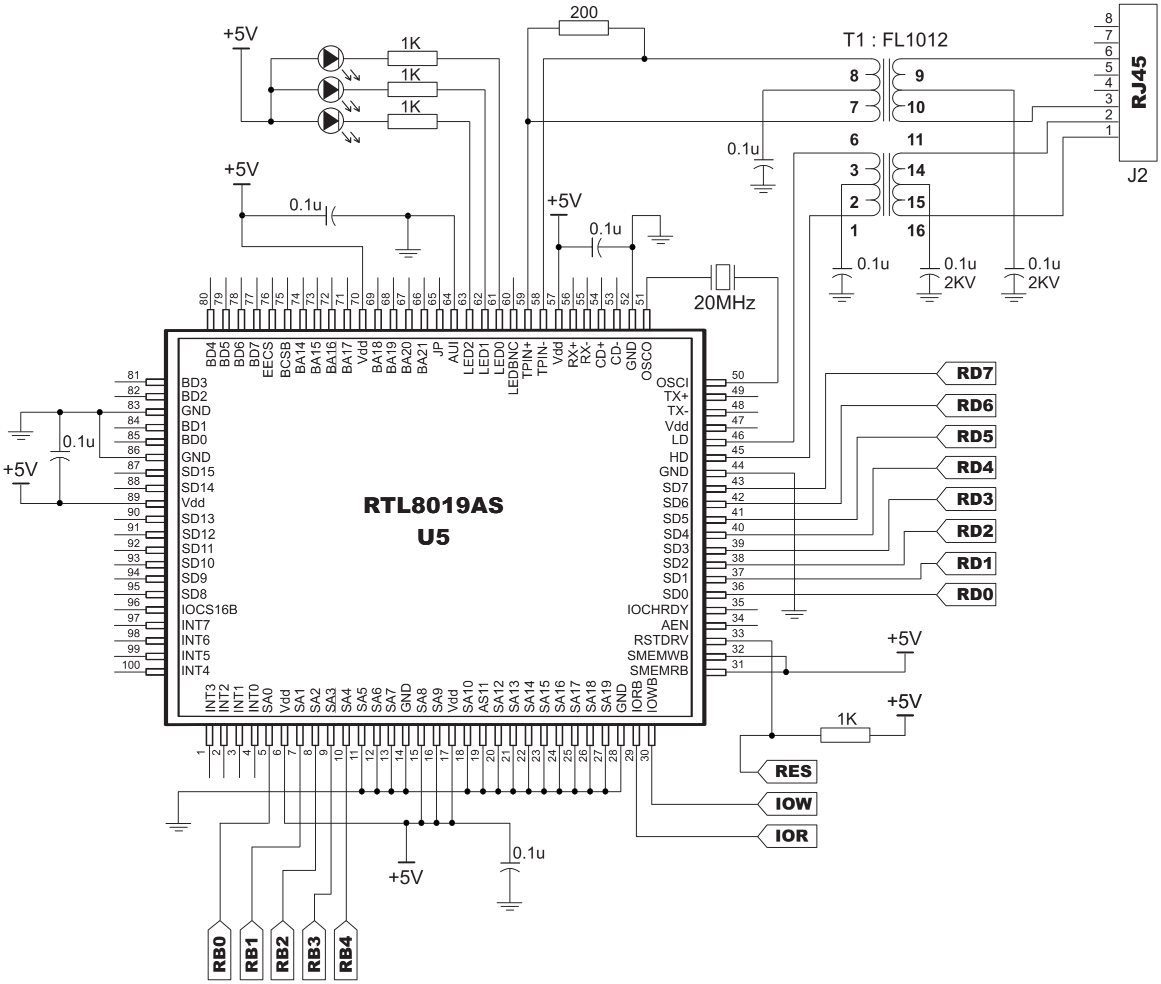
Eth_Tcp_Response

Prototype	`sub procedure Eth_Tcp_Response`
Description	Performs user response to TCP/IP event. User specifies data to be sent, depending on the request received (HTTP, HTTPD, FTP, etc). This is performed by the function `Eth_Set_Tcp_Data`.
Requires	Hardware requirements are as specified for the entire library. Prior to using this procedure, user must prepare the data to be sent through TCP; see `Eth_Set_Tcp_Data`.
Example	`Eth_Tcp_Response()`

Library Example

Check the supplied Ethernet example in the *Examples* folder.

HW Connection



SPI Ethernet Library

The ENC28J60 is a stand-alone Ethernet controller with an industry standard Serial Peripheral Interface (SPI™). It is designed to serve as an Ethernet network interface for any controller equipped with SPI.

The ENC28J60 meets all of the IEEE 802.3 specifications. It incorporates a number of packet filtering schemes to limit incoming packets. It also provides an internal DMA module for fast data throughput and hardware assisted IP checksum calculations. Communication with the host controller is implemented via two interrupt pins and the SPI, with data rates of up to 10 Mb/s. Two dedicated pins are used for LED link and network activity indication.

This library is designed to simplify handling of the underlying hardware (ENC28J60). It works with any PIC with integrated SPI and more than 4 Kb ROM memory. 38 to 40 MHz clock is recommended to get from 8 to 10 Mhz SPI clock, otherwise PIC should be clocked by ENC clock output due to ENC silicon bug in SPI hardware. if you try lower PIC clock speed, there might be board hang or miss some requests. This library is tested with PIC16F877A@10Mhz, PIC18F452@40Mhz.

Note: For advanced users there is a header in Uses\P16 and Uses\P18 folder ("enc28j60_libprivate.pbas") with detailed description of all functions which are implemented in SPI Ethernet Library.

Note: `SPI_Init` must be called before initializing SPI Ethernet.

Library Routines

```
SPI_Ethernet_Init
SPI_Ethernet_doPacket
SPI_Ethernet_putByte
SPI_Ethernet_getByte
SPI_Ethernet_UserTCP
SPI_Ethernet_UserUDP
```

SPI_Ethernet_Init

Prototype	**`sub procedure`** `SPI_Ethernet_Init(`**`dim byref`**` resetPort `**`as byte, dim`**` resetBit `**`as byte, dim byref`**` CSportPtr `**`as byte, dim`**` CSbit `**`as byte, dim byref`**` mac `**`as byte`**`[ 6] , `**`dim byref`**` ip `**`as byte`**`[ 4] , `**`dim`**` fullDuplex `**`as byte`**`)`
Returns	Nothing.
Description	Initialize ENC controller. This function is splited into 2 parts to help linker when coming short of memory. **`resetPort`** `- pointer to reset pin port` **`resetBit`** `- reset bit number on resetPort` **`CSport`** `- pointer to CS pin port` **`CSbit`** `- CS bit number on CSport` **`mac`** `- pointer to array of 6 char with MAC address` **`ip`** `- pointer to array of 4 char with IP address` **`fullDuplex`** `- either SPI_Ethernet_HALFDUPLEX for half duplex or SPI_Ethernet_FULLDUPLEX for full duplex`
Requires	`SPI_Init` must be called before initializing SPI Ethernet.
Example	`SPI_Ethernet_Init(PORTC, 0, PORTC, 1, myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX)`

SPI_Ethernet_doPacket

Prototype	**`sub procedure`** `SPI_Ethernet_doPacket`
Returns	Nothing.
Description	Process one incoming packet if available.
Requires	SPI_Ethernet_init must have been called before using this function. This function must be called as often as possible by user.
Example	`SPI_Ethernet_doPacket`

SPI_Ethernet_putByte

Prototype	`sub procedure ENC28J60_putByte(dim v as byte)`
Returns	Nothing.
Description	`v - value to store` Store one byte to current EWRPT ENC location.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	`SPI_Ethernet_putByte(0xa0)`

SPI_Ethernet_getByte

Prototype	`dim function SPI_Ethernet_getByte as byte`
Returns	Value of byte @ addr.
Description	Get next byte from current ERDPT ENC location.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	`b = SPI_Ethernet_getByte`

SPI_Ethernet_UserTCP

Prototype	`sub function SPI_Ethernet_UserTCP(dim byref remoteHost as byte[4], dim remotePort, localPort, reqLength as word) as word`
Returns	Returns the length in bytes of the HTTP reply, or 0 if nothing to transmit.
Description	This function is called by the library. The user accesses to the HTTP request by successive calls to SPI_Ethernet_getByte the user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte the function must return the length in bytes of the HTTP reply, or 0 if nothing to transmit. If you don't need to reply to HTTP requests, just define this function with a return(0) as single statement.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	

SPI_Ethernet_UserUDP

Prototype	`sub function SPI_Ethernet_UserUDP(dim byref remoteHost as byte[4], remotePort, destPort, reqLength as word) as word`
Returns	Returns the length in bytes of the UDP reply, or 0 if nothing to transmit.
Description	This function is called by the library. The user accesses to the UDP request by successive calls to SPI_Ethernet_getByte. The user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte. The function must return the length in bytes of the UDP reply, or 0 if nothing to transmit. If you don't need to reply to UDP requests,just define this function with a return(0) as single statement.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	

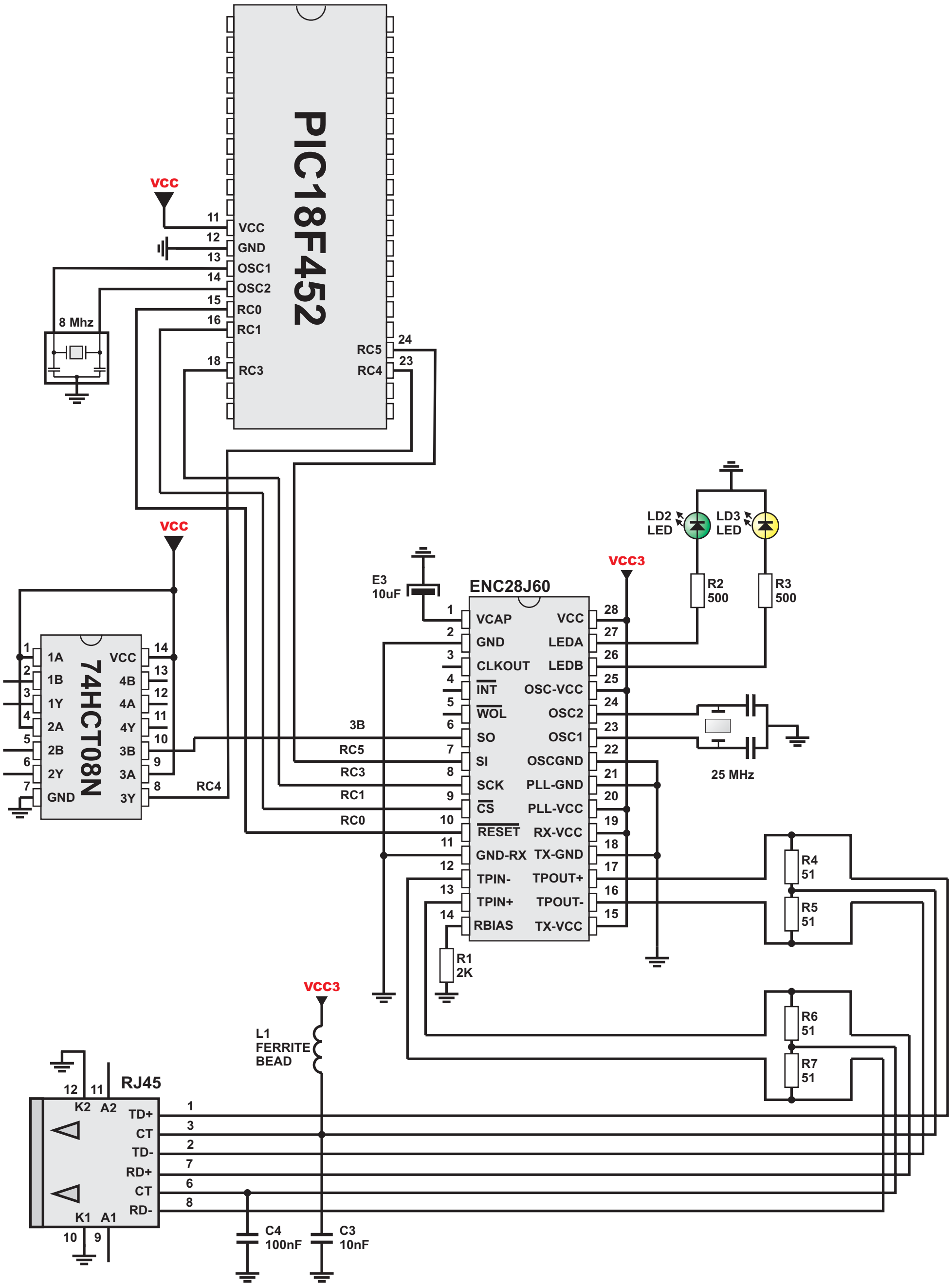
Library Example

The following example is a simple demonstration of the SPI Ethernet Library. PIC is assigned an IP address of 192.168.20.60, and will respond to ping if connected to a local area network.

```
program enc_ethernet

include "enc_utils" ' this is where you should write implementation for UDP and HTTP
include "enc_eth"

'{***********************************
' * RAM variables
' *}

dim myMacAddr    as byte[ 6]      ' my MAC address
     myIpAddr     as byte[ 4]      ' my IP address

main:

  ADCON1 = 0x00              ' ADC convertors will be used

  PORTA = 0
  TRISA = 0xff               ' set PORTA as input for ADC

  PORTB = 0
  TRISB = 0xff               ' set PORTB as input for buttons

  PORTD = 0
  TRISD = 0                   ' set PORTD as output

  httpCounter = 0

  myMacAddr[ 0]  = 0x00
  myMacAddr[ 1]  = 0x14
  myMacAddr[ 2]  = 0xA5
  myMacAddr[ 3]  = 0x76
  myMacAddr[ 4]  = 0x19
  myMacAddr[ 5]  = 0x3F

  myIpAddr[ 0]  = 192
  myIpAddr[ 1]  = 168
  myIpAddr[ 2]  = 20
  myIpAddr[ 3]  = 60

' continues...
```

```
    ' continued...

    '*
    '* starts ENC28J60 with :
    '* reset bit on RC0
    '* CS bit on RC1
    '* my MAC & IP address
    '* full duplex
    '*

    Spi_Init ' initialize SPI

    SPI_Ethernet_Init(PORTC, 0, PORTC, 1, myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX)

    while true                    ' do forever
        SPI_Ethernet_doPacket()      ' process incoming Ethernet packets

        '{*
        ' * add your stuff here if needed
        ' * SPI_Ethernet_doPacket() must be called as often as possible
        ' * otherwise packets could be lost
        ' *}
    wend
  end.
```

HW Connection



Flash Memory Library

This library provides routines for accessing microcontroller Flash memory. Note that prototypes differ for PIC16 and PIC18 families.

Note: Due to P16 family flash specifics, flash library is MCU dependent. There are three kinds of MCU's that support flash memory operations:

1. Only flash Read operation supported. For this group of MCU's only Flash_Read function is implemented.
2. Read and Write operations are supported (write is executed as erase-and-write). For this group of MCU's read and write functions are implemented.
3. Read, Write and Erase operations supported. For this group of MCU's read, write and erase functions are implemented. Further more, flash memory block has to be erased prior to writting (write operation is not executed as erase_and_write).

Please refer to datasheet before using flash library.

Library Routines

```
Flash_Write
Flash_Read
Flash_Erase
```

Flash_Write

Prototype	`sub procedure Flash_Write(dim Address as word, dim byref Data as word[4]) ' for PIC16` `sub procedure Flash_Write(dim Address as longint, dim byref Data as word[64]) ' for PIC18`
Description	Writes chunk of data to Flash memory. With PIC18, data needs to be exactly 64 bytes in size. The function erases target memory before writing data to it. This means that if write was unsuccessful, previous data will be lost.
Example	

```
' Write consecutive values in 64 consecutive locations
for i = 0 to 63
  toWrite[i] = i
next i

Flash_Write($0D00, toWrite)
```

Flash_Read

Prototype	`' for PIC16`
	`sub function Flash_Read(dim address as word) as byte`
	`' for PIC18`
	`sub function Flash_Read(dim address as longint) as byte`

Returns	Returns data byte from Flash memory.
Description	Reads data from the specified address in Flash memory.
Example	`Flash_Read($D00)`

Flash_Erase

Prototype	`sub procedure Flash_Erase(dim address as word)`
Returns	Nothing.
Description	Erases 32 bytes memory block starting from a given address. Implemented only for those MCU's whose flash memory does not support erase-and-write operations (refer to datasheet for details).
Example	`' Erase 32 byte memory memory block, starting from address $0D00:` `Flash_Erase($0D00)`

Library Example

The following examples write 64 consecutive values to 64 consecutive locations in flash memory. Then, the written data is verified, with error indication on PORTB.

```
' For PIC18

program Flash_P18

dim i   as byte
  addr as longint
  dataRd as byte
  dataWr as byte[ 64]

main:
  PORTB = 0
  TRISB = 0

  for i = 0 to 63
    dataWr[ i]  = i
  next i

  addr = 0x00000A30          ' valid for P18F452
  Flash_Write(addr, dataWr)

  addr = 0x00000A30
  for i = 0 to 63
    dataRd = Flash_Read(addr)
    PORTB =  dataRd
    addr  =  addr + 1
    Delay_ms(200)
  next i
end.
```

I2C Library

I²C full master MSSP module is available with a number of PIC MCU models. mikroBasic provides I2C library which supports the master I²C mode.

Note: This library supports module on PORTB or PORTC, and will not work with modules on other ports. Examples for PICmicros with module on other ports can be found in your mikroBasic installation folder, subfolder “Examples”.

Library Routines

```
I2C_Init
I2C_Start
I2C_Repeated_Start
I2C_Is_Idle
I2C_Rd
I2C_Wr
I2C_Stop
```

I2C_Init

Prototype	`sub procedure I2C_Init(const clock as longint)`
Description	Initializes I²C with desired `clock` (refer to device data sheet for correct values in respect with Fosc). Needs to be called before using other functions of I2C Library.
Requires	Library requires MSSP module on PORTB or PORTC.
Example	`I2C_Init(100000)`

I2C_Start

Prototype	`sub function I2C_Start as byte`
Returns	If there is no error, function returns 0.
Description	Determines if I²C bus is free and issues START signal.
Requires	I²C must be configured before using this function. See `I2C_Init`.
Example	`if I2C_Start = 0 then` `...`

I2C_Repeated_Start

Prototype	`sub procedure I2C_Repeated_Start`
Description	Issues repeated START signal.
Requires	I²C must be configured before using this function. See `I2C_Init`.
Example	`I2C_Repeated_Start`

I2C_Is_Idle

Prototype	`sub function I2C_Is_Idle as byte`
Returns	Returns 1 if I²C bus is free, otherwise returns 0.
Description	Tests if I²C bus is free.
Requires	I²C must be configured before using this function. See `I2C_Init`.
Example	`if I2C_Is_Idle then` `...`

I2C_Rd

Prototype	`sub function I2C_Rd(dim ack as byte) as byte`
Returns	Returns one byte from the slave.
Description	Reads one byte from the slave, and sends not acknowledge signal if parameter `ack` is 0, otherwise it sends acknowledge.
Requires	START signal needs to be issued in order to use this function. See `I2C_Start`.
Example	`tmp = I2C_Rd(0)  ' Read data and send not acknowledge signal`

I2C_Wr

Prototype	`sub function I2C_Wr(dim data as byte) as byte`
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter data) via I²C bus.
Requires	START signal needs to be issued in order to use this function. See `I2C_Start`.
Example	`I2C_Write($A3)`

I2C_Stop

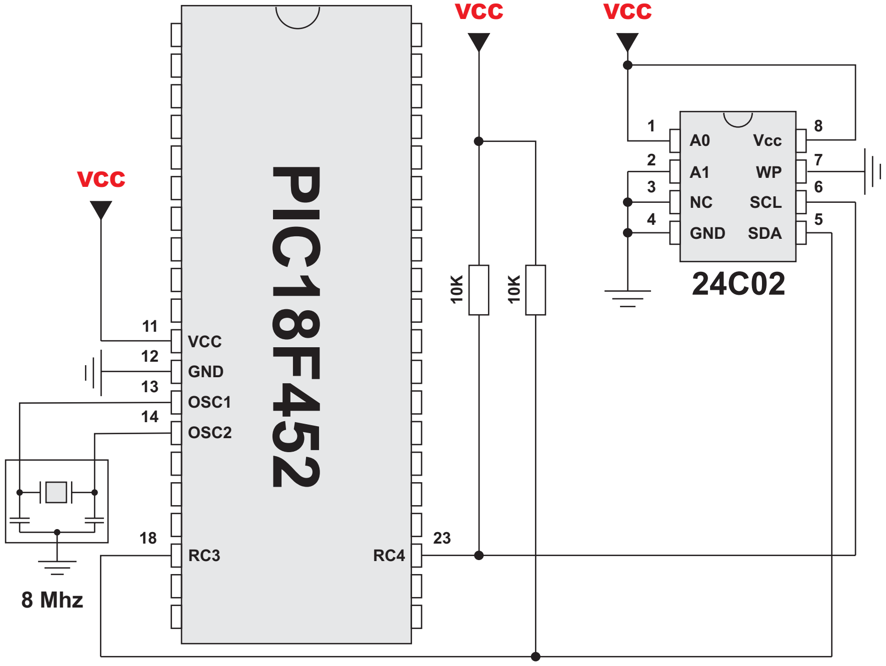
Prototype	`sub procedure I2C_Stop`
Description	Issues STOP signal.
Requires	I²C must be configured before using this function. See `I2C_Init`.

Library Example

This code demonstrates use of I2C Library procedures and functions. PIC MCU is connected (pins SCL, SDA) 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 (figure on the following page shows how to interface 24c02 to PIC).

```
program Eeprom_test

dim EE_adr, EE_data, k as byte
dim jj as word

main:
  I2C_Init(100000)          ' Initialize full master mode
  TRISD = 0                 ' PORTD is 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(data that will be written)
  I2C_Stop                  ' Issue I2C stop signal

  ' Pause while EEPROM writes data
  for jj = 0 to 65500
    nop
  next jj

  I2C_Start                 ' Issue I2C start signal
  I2C_Wr($A2)               ' Send byte via I2C
  EE_adr = 2
  I2C_Wr(EE_adr)            ' Send byte(address for EEPROM)
  I2C_Repeated_Start        ' Issue I2C signal repeated start
  I2C_Wr($A3)               ' Send byte (request data from EEPROM)
  k = I2C_Rd(1)             ' Read the data
  I2C_Stop                  ' Issue I2C stop signal
  PORTD = k                 ' Show data on PORTD

  ' Endless loop
  while true
    nop
  wend

end.
```

HW Connection



Keypad Library

mikroBasic provides library for working with 4x4 keypad; routines can also be used with 4x1, 4x2, or 4x3 keypad. Check the connection scheme at the end of the topic.

Library Routines

```
Keypad_Init
Keypad_Read
Keypad_Released
```

Keypad_Init

Prototype	`sub procedure Keypad_Init(dim byref port as word)`
Description	Initializes port to work with keypad. The procedure needs to be called before using other routines from Keypad library.
Example	`Keypad_Init(PORTB)`

Keypad_Read

Prototype	`sub function Keypad_Read as word`
Returns	1..16, depending on the key pressed, or 0 if no key is pressed.
Description	Checks if any key is pressed. Function returns 1 to 16, depending on the key pressed, or 0 if no key is pressed.
Requires	Port needs to be appropriately initialized; see `Keypad_Init`.
Example	`kp = Keypad_Read`

Keypad_Released

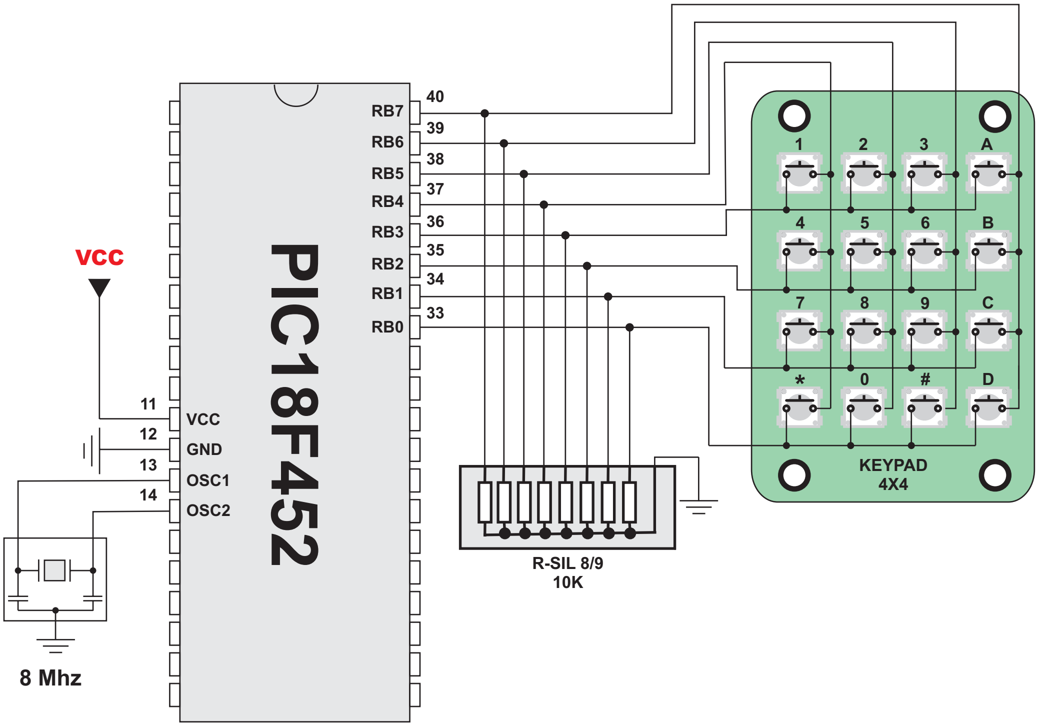
Prototype	`sub function Keypad_Released as word`
Returns	1..16, depending on the key.
Description	Call to `Keypad_Released` is a blocking call: function waits until any key is pressed and released. When released, function returns 1 to 16, depending on the key.
Requires	Port needs to be appropriately initialized; see `Keypad_Init`.
Example	`kp = Keypad_Released`

Library Example

The following code can be used for testing the keypad. It supports keypads with 1 to 4 rows and 1 to 4 columns. The code returned by the keypad functions (1..16) is transformed into ASCII codes [0..9,A..F]. In addition, a small single-byte counter displays the total number of keys pressed in the second LCD row.

```
program keypad_test

dim kp, cnt as byte
dim txt as string[ 5]

main:
  cnt = 0
  Keypad_Init(PORTC)
  Lcd_Init(PORTB)           ' Initialize LCD on PORTC
  Lcd_Cmd(LCD_CLEAR)        ' Clear display
  Lcd_Cmd(LCD_CURSOR_OFF)   ' Cursor off

  Lcd_Out(1, 1, "Key  :")
  Lcd_Out(2, 1, "Times:")

  while TRUE
    kp = 0

    '--- Wait for key to be pressed
    while kp = 0
      '--- un-comment one of the keypad reading functions
      kp = Keypad_Released
      'kp = Keypad_Read
    wend

    Inc(cnt)

    '--- prepare value for output
    if kp > 10 then
      kp = kp + 54
    else
      kp = kp + 47
    end if

    '--- print it on LCD
    Lcd_Chr(1, 10, kp)
    WordToStr(cnt, txt)
    Lcd_Out(2, 10, txt)
  wend
end.
```

HW Connection



LCD Library (4-bit interface)

mikroBasic provides a library for communicating with commonly used LCD (4-bit interface). Figures showing HW connection of PIC and LCD are given at the end of the chapter.

Library Routines

```
Lcd_Config
Lcd_Init
Lcd_Out
Lcd_Out_Cp
Lcd_Chr
Lcd_Chr_Cp
Lcd_Cmd
```

Lcd_Config

Prototype	`sub procedure Lcd_Config(dim byref data_port as byte,dim D7, D6, D5, D4 as byte,dim byref ctrl_port as byte,dim RS, WR, EN as byte)`
Description	Initializes LCD data port and control port with pin settings you specify.
Example	`Lcd_Config(PORTD,3,2,1,0,PORTB,2,3,4)`

Lcd_Init

Prototype	`sub procedure Lcd_Init(dim byref port as byte)`
Description	Initializes LCD at port with default pin settings (see the connection scheme at the end of the chapter): D7 -> PORT.7, D6 -> PORT.6, D5 -> PORT.5, D4 -> PORT.4, E -> PORT.3, RS -> PORT.2.
Example	`Lcd_Init(PORTB)`

Lcd_Out

Prototype	`sub procedure Lcd_Out(dim row, col as byte, dim byref text as char[ 255] )`
Description	Prints `text` on LCD at specified row and column (parameter `row` and `col`). Both string variables and literals can be passed as `text`.
Requires	Port with LCD must be initialized. See `Lcd_Config` or `Lcd_Init`.
Example	`Lcd_Out(1, 3, "Hello!")   ' Print "Hello!" at line 1, char 3`

Lcd_Out_Cp

Prototype	`sub procedure Lcd_Out_Cp(dim byref text as char[ 255] )`
Description	Prints `text` on LCD at current cursor position. Both string variables and literals can be passed as `text`.
Requires	Port with LCD must be initialized. See `Lcd_Config` or `Lcd_Init`.
Example	`Lcd_Out_Cp("Here!")   ' Print "Here!" at current cursor position`

Lcd_Chr

Prototype	`sub procedure Lcd_Chr(dim row, col, character as byte)`
Description	Prints `character` on LCD at specified row and column (parameters `row` and `col`). Both variables and literals can be passed as `character`.
Requires	Port with LCD must be initialized. See `Lcd_Config` or `Lcd_Init`.
Example	`Lcd_Chr(2, 3, "i")   ' Print "i" at line 2, char 3`

Lcd_Chr_Cp

Prototype	`sub procedure Lcd_Chr_Cp(dim character as byte)`
Description	Prints `character` on LCD at current cursor position. Both variables and literals can be passed as `character`.
Requires	Port with LCD must be initialized. See `Lcd_Config` or `Lcd_Init`.
Example	`Lcd_Chr_Cp("e")   ' Print "e" at current cursor position`

Lcd_Cmd

Prototype	`sub procedure Lcd_Cmd(dim command as byte)`
Description	Sends `command` to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown on the page 140.
Requires	Port with LCD must be initialized. See `Lcd_Config` or `Lcd_Init`.
Example	`Lcd_Cmd(LCD_CLEAR)   ' Clear LCD display`

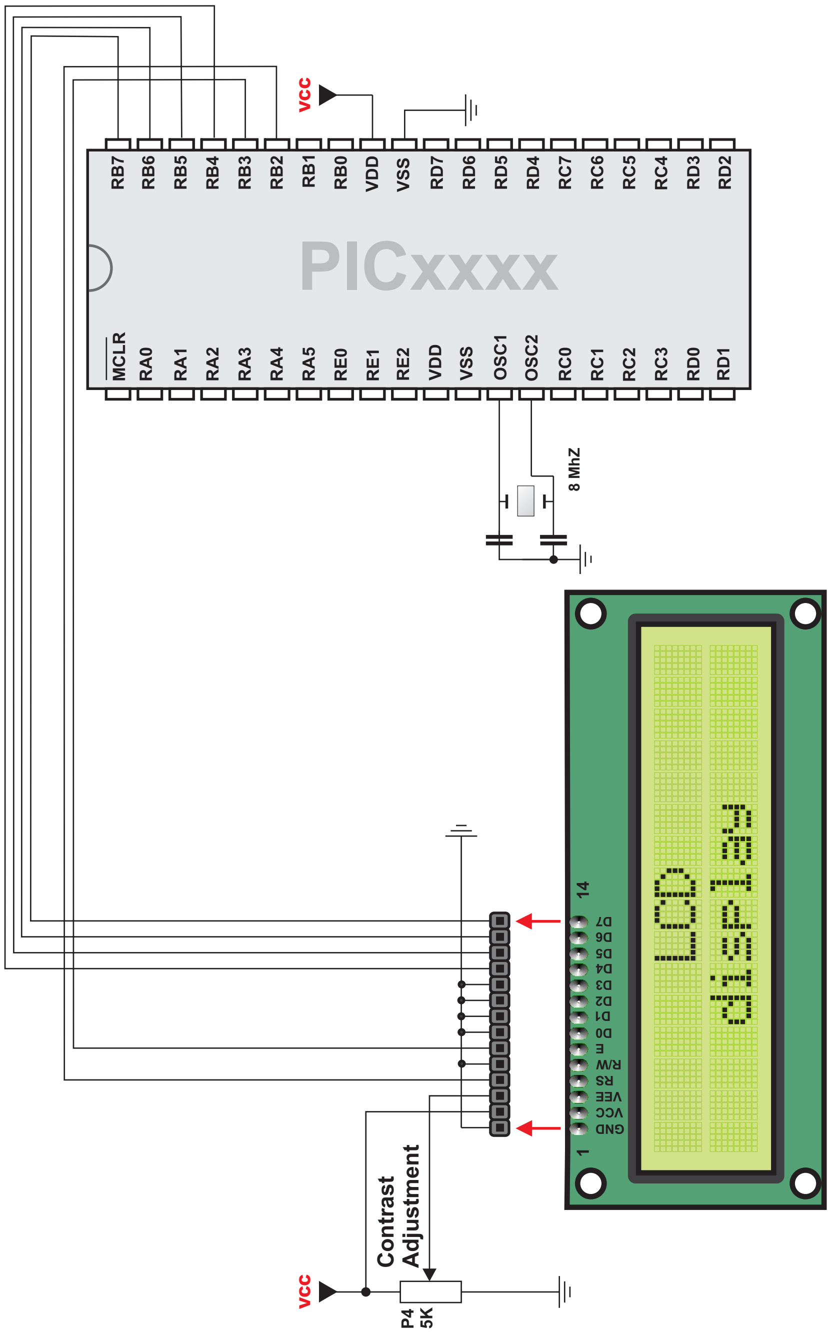
LCD Commands

LCD Command	Purpose
`LCD_FIRST_ROW`	Move cursor to 1st row
`LCD_SECOND_ROW`	Move cursor to 2nd row
`LCD_THIRD_ROW`	Move cursor to 3rd row
`LCD_FOURTH_ROW`	Move cursor to 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to original position. Display data RAM is unaffected.
`LCD_CURSOR_OFF`	Turn 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

Library Example (default pin settings)

```
program Lcd_default_test
dim text as char[ 20]

main:
  TRISB = 0                        ' PORTB is output
  Lcd_Init(PORTB)                  ' Initialize LCD on PORTB
  Lcd_Cmd(Lcd_CURSOR_OFF)          ' Turn off cursor
  text = "mikroElektronika"
  Lcd_Out(1, 1, text)              ' Print text at LCD
end.
```

Hardware Connection



LCD Library (8-bit interface)

mikroBasic provides a library for communicating with commonly used 8-bit interface LCD (with Hitachi HD44780 controller). Figures showing HW connection of PIC and LCD are given at the end of the chapter.

Library Routines

```
Lcd8_Init
Lcd8_Out
Lcd8_Out_Cp
Lcd8_Chr
Lcd8_Chr_Cp
Lcd8_Cmd
```

Lcd8_Init

Prototype	`sub procedure Lcd8_Init(dim byref ctrlport, dataport as byte)`
Description	Initializes LCD at Control port (ctrlport) and Data port (dataport) with default pin settings (see the connection scheme at the end of the chapter): E -> ctrlport.3, RS -> ctrlport.2, R/W -> ctrlport.0, D7 -> dataport.7, D6 -> dataport.6, D5 -> dataport.5, D4 -> dataport.4, D3 -> dataport.3, D2 -> dataport.2, D1 -> dataport.1, D0 -> dataport.0
Example	`Lcd8_Init(PORTB, PORTC)`

Lcd8_Out

Prototype	`sub procedure Lcd8_Out(dim row, col as byte, dim byref text as char[255])`
Description	Prints `text` on LCD at specified row and column (parameter `row` and `col`). Both string variables and literals can be passed as `text`.
Requires	Ports with LCD must be initialized. See `Lcd8_Config` or `Lcd8_Init`.
Example	`Lcd8_Out(1, 3, "Hello!")   ' Print "Hello!" at line 1, char 3`

Lcd8_Out_Cp

Prototype	`sub procedure Lcd8_Out_Cp(dim byref text as char[255])`
Description	Prints `text` on LCD at current cursor position. Both string variables and literals can be passed as `text`.
Requires	Ports with LCD must be initialized. See `Lcd8_Config` or `Lcd8_Init`.
Example	`Lcd8_Out_Cp("Here!")   ' Print "Here!" at current cursor position`

Lcd8_Chr

Prototype	`void Lcd8_Chr(char row, char col, char character);`
Description	Prints `character` on LCD at specified row and column (parameters `row` and `col`). Both variables and literals can be passed as `character`.
Requires	Ports with LCD must be initialized. See `Lcd8_Config` or `Lcd8_Init`.
Example	`Lcd8_Out(2, 3, "i")   ' Print "i" at line 2, char 3`

Lcd8_Chr_Cp

Prototype	**`sub procedure`** `Lcd8_Chr_Cp(`**`dim`** `character` **`as byte`**`)`
Description	Prints `character` on LCD at current cursor position. Both variables and literals can be passed as `character`.
Requires	Ports with LCD must be initialized. See `Lcd8_Config` or `Lcd8_Init`.
Example	`Lcd8_Chr_Cp("e")   ' Print "e" at current cursor position`

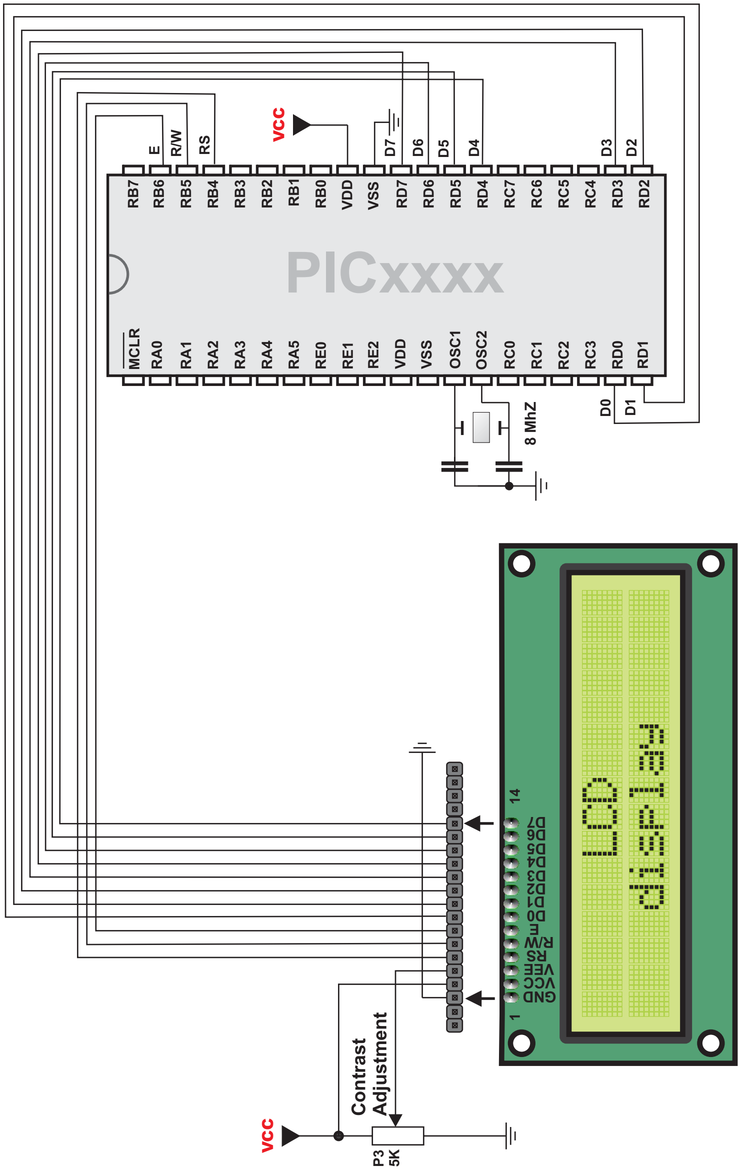
Lcd8_Cmd

Prototype	**`sub procedure`** `Lcd8_Cmd(`**`dim`** `command` **`as byte`**`)`
Description	Sends `command` to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is on the page 140.
Requires	Ports with LCD must be initialized. See `Lcd8_Config` or `Lcd8_Init`.
Example	`Lcd8_Cmd(LCD_CLEAR)   ' Clear LCD display`

Library Example (default pin settings)

```
program Lcd8_default_test
dim text as char[20]

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.
```

Hardware Connection



GLCD Library

mikroBasic provides a library for drawing and writing on Graphic LCD. These routines work with commonly used GLCD 128x64, and work only with the PIC18 family.

Library Routines

Basic routines:

```
Glcd_Init
Glcd_Set_Side
Glcd_Set_Page
Glcd_Set_X
Glcd_Read_Data
Glcd_Write_Data
```

Advanced routines:

```
Glcd_Fill
Glcd_Dot
Glcd_Line
Glcd_V_Line
Glcd_H_Line
Glcd_Rectangle
Glcd_Box
Glcd_Circle
Glcd_Set_Font
Glcd_Write_Char
Glcd_Write_Text
Glcd_Image
```

Glcd_Init

Prototype	**`sub procedure`** `Glcd_Init(`**`dim byref`** `ctrlport` **`as byte, dim`** `cs1, cs2, rs, rw, rst, en` **`as byte, dim byref`** `dataport` **`as byte`**`)`
Description	Initializes GLCD at lower byte of `data_port` with pin settings you specify. Parameters `cs1`, `cs2`, `rs`, `rw`, `rst`, and `en` can be pins of any available port. This procedure needs to be called befored using other routines of GLCD library.
Example	`Glcd_Init(PORTB, 2, 0, 3, 5, 7, 1, PORTC)`

Glcd_Set_Side

Prototype	**`sub procedure`** `Glcd_Set_Side(`**`dim`** `x` **`as byte`**`)`
Description	Selects side of GLCD, left or right. Parameter `x` specifies the side: values from 0 to 63 specify the left side, and values higher than 64 specify the right side. Use the functions `Glcd_Set_Side`, `Glcd_Set_X`, and `Glcd_Set_Page` to specify an exact position on GLCD. Then, you can use `Glcd_Write_Data` or `Glcd_Read_Data` on that location.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Set_Side(0)`

Glcd_Set_Page

Prototype	`sub procedure Glcd_Set_Page(dim page as byte)`
Description	Selects page of GLCD, technically a line on display; parameter `page` can be 0..7.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Set_Page(5)`

Glcd_Set_X

Prototype	`sub procedure Glcd_Set_X(dim x as byte)`
Description	Positions to `x` dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Set_X(25)`

Glcd_Read_Data

Prototype	`sub function Glcd_Read_Data as byte`
Returns	One word from the GLCD memory.
Description	Reads data from from the current location of GLCD memory. Use the functions `Glcd_Set_Side`, `Glcd_Set_X`, and `Glcd_Set_Page` to specify an exact position on GLCD. Then, you can use `Glcd_Write_Data` or `Glcd_Read_Data` on that location.
Requires	Reads data from from the current location of GLCD memory.
Example	`tmp = Glcd_Read_Data()`

Glcd_Write_Data

Prototype	`sub procedure Glcd_Write_Data(dim data as byte)`
Description	Writes `data` to the current location in GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Write_Data(data)`

Glcd_Fill

Prototype	`sub procedure Glcd_Fill(dim pattern as byte)`
Description	Fills the GLCD memory with byte pattern. To clear the GLCD screen, use `Glcd_Fill(0)`; to fill the screen completely, use `Glcd_Fill($FF)`.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Fill(0)   ' Clear screen`

Glcd_Dot

Prototype	`sub procedure Glcd_Dot(dim x, y, color as byte)`
Description	Draws a dot on the GLCD at coordinates `(x, y)`. Parameter color determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Dot(0, 0, 2)   ' Invert the dot in the upper left corner`

Glcd_Line

Prototype	`sub procedure Glcd_Line(dim x1, y1, x2, y2, color as byte)`
Description	Draws a line on the GLCD from `(x1, y1)` to `(x2, y2)`. Parameter `color` determines the dot state: 0 draws an empty line (clear dots), 1 draws a full line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Line(0, 63, 50, 0, 2)`

Glcd_V_Line

Prototype	`sub procedure Glcd_V_Line(dim y1, y2, x, color as byte)`
Description	Similar to `Glcd_Line`, draws a vertical line on the GLCD from `(x, y1)` to `(x, y2)`.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_V_Line(0, 63, 0, 1)`

Glcd_H_Line

Prototype	`sub procedure Glcd_H_Line(dim x1, x2, y, color as byte)`
Description	Similar to `Glcd_Line`, draws a horizontal line on the GLCD from `(x1, y)` to `(x2, y)`.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_H_Line(0, 127, 0, 1)`

Glcd_Rectangle

Prototype	`sub procedure Glcd_Rectangle(dim x1, y1, x2, y2, color as byte)`
Description	Draws a rectangle on the GLCD. Parameters `(x1, y1)` set the upper left corner, `(x2, y2)` set the bottom right corner. Parameter `color` defines the border: 0 draws an empty border (clear dots), 1 draws a solid border (put dots), and 2 draws a “smart” border (invert each dot).
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Rectangle(10, 0, 30, 35, 1)`

Glcd_Box

Prototype	`sub procedure Glcd_Box(dim x1, y1, x2, y2, color as byte)`
Description	Draws a box on the GLCD. Parameters `(x1, y1)` set the upper left corner, `(x2, y2)` set the bottom right corner. Parameter `color` defines the fill: 0 draws a white box (clear dots), 1 draws a full box (put dots), and 2 draws an inverted box (invert each dot).
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Box(10, 0, 30, 35, 1)`

Glcd_Circle

Prototype	`sub procedure Glcd_Circle(dim x, y, radius, color as integer)`
Description	Draws a circle on the GLCD, centered at (x, y) with radius. Parameter `color` defines the circle line: 0 draws an empty line (clear dots), 1 draws a solid line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Circle(63, 31, 25)`

Glcd_Set_Font

Prototype	`sub procedure Glcd_Set_Font(dim font_address as longint, dim font_width, font_height as byte, dim font_offset as word)`
Description	Sets the font for text display routines, `Glcd_Write_Char` and `Glcd_Write_Text`. Font needs to be formatted as an array of `byte`. Parameter `font_address` specifies the address of the font; you can pass a font name with the `@` operator. Parameters `font_width` and `font_height` specify the width and height of characters in dots. Font width should not exceed 128 dots, and font height should not exceed 8 dots. Parameter `font_offset` determines the ASCII character from which the supplied font starts. Demo fonts supplied with the library have an offset of 32, which means that they start with space. If no font is specified, `Glcd_Write_Char` and `Glcd_Write_Text` will use the default 5x8 font supplied with the library. You can create your own fonts by following the guidelines given in the file “GLCD_Fonts.ppas”. This file contains the default fonts for GLCD, and is located in your installation folder, “Extra Examples” > “GLCD”.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`' Use the custom 5x7 font "myfont" which starts with space (32):` `Glcd_Set_Font(@myfont, 5, 7, 32)`

Glcd_Write_Char

Prototype	`sub procedure Glcd_Write_Char(dim character, x, page, color as byte)`
Description	Prints character at `page` (one of 8 GLCD lines, 0..7), `x` dots away from the left border of display. Parameter color defines the “fill”: 0 writes a “white” letter (clear dots), 1 writes a solid letter (put dots), and 2 writes a “smart” letter (invert each dot). Use routine `Glcd_Set_Font` to specify font, or the default 5x7 font (included with the library) will be used.
Requires	GLCD needs to be initialized, see `Glcd_Init`. Use the `Glcd_Set_Font` to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	`Glcd_Write_Char("C", 0, 0, 1)`

Glcd_Write_Text

Prototype	`sub procedure Glcd_Write_Text(dim text as string[20], dim x, page, color as byte)`
Description	Prints `text` at `page` (one of 8 GLCD lines, 0..7), `x` dots away from the left border of display. Parameter `color` defines the "fill": 0 prints a "white" letters (clear dots), 1 prints solid letters (put dots), and 2 prints "smart" letters (invert each dot). Use routine `Glcd_Set_Font` to specify font, or the default 5x7 font (included with the library) will be used.
Requires	GLCD needs to be initialized, see `Glcd_Init`. Use the `Glcd_Set_Font` to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	`Glcd_Write_Text("Hello world!", 0, 0, 1)`

Glcd_Image

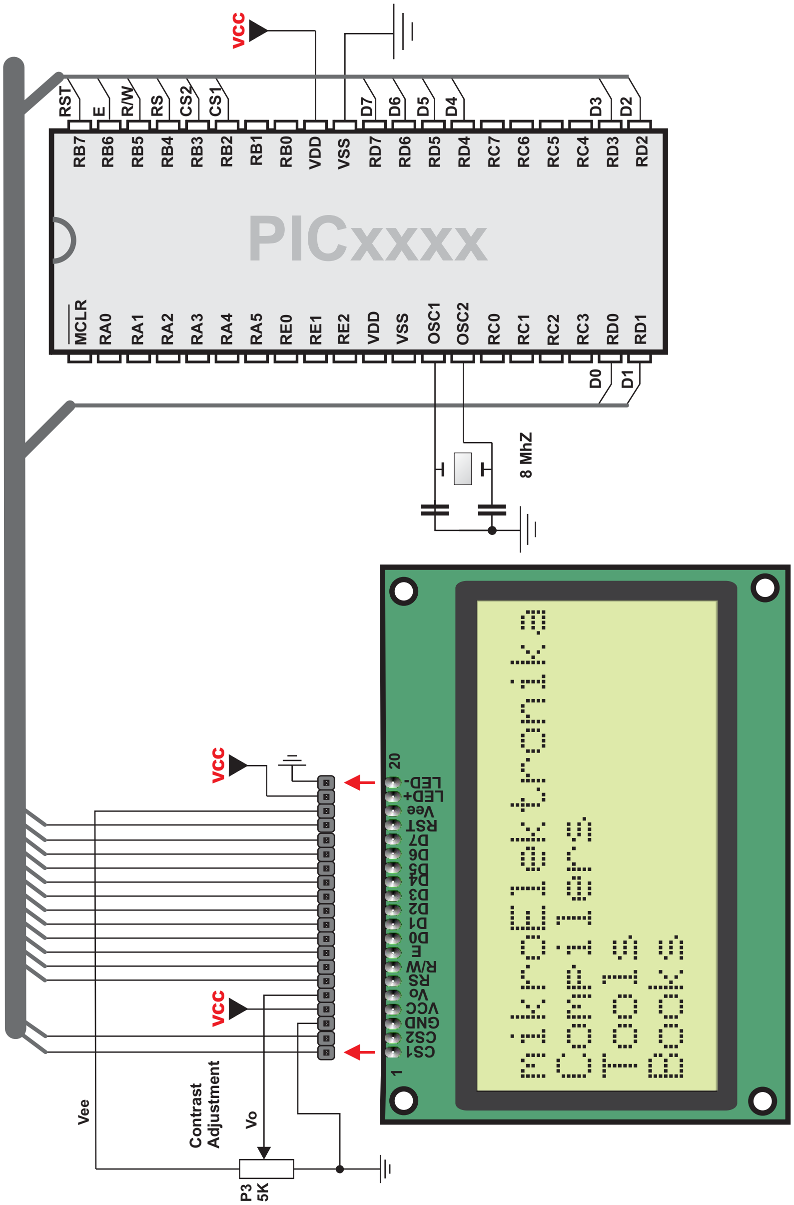
Prototype	`sub procedure Glcd_Image(dim image as byte[1024])`
Description	Displays bitmap image on the GLCD. Parameter `image` should be formatted as an array of 1024 bytes. Use the mikroBasic's integrated Bitmap-to-LCD editor (menu option Tools › Graphic LCD Editor) to convert image to a constant array suitable for display on GLCD.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Image(my_image)`

Library Example

The following drawing demo tests advanced routines of GLCD library.

```
program Glcd_Test

main:
  Glcd_Init(PORTB, 2, 0, 3, 5, 7, 1, PORTD)

  ' Set font for displaying text
  Glcd_Set_Font(@FontSystem5x8, 5, 8, 32)

  do

    ' Draw circles
    Glcd_Fill(0)   ' Clear screen
    Glcd_Write_Text("Circles", 0, 0, 1)
    j = 4
    while j < 31
      Glcd_Circle(63, 31, j, 2)
      j = j + 4
    wend
    Delay_ms(4000)

    ' Draw boxes
    Glcd_Fill(0)   ' Clear screen
    Glcd_Write_Text("Rectangles", 0, 0, 1)
    j = 0
    while j < 31
      Glcd_Box(j, 0, j + 20, j + 25, 2)
      j = j + 4
    wend
    Delay_ms(4000)

    ' Draw Lines
    Glcd_Fill(0)   ' Clear screen
    Glcd_Write_Text("Lines", 0, 0, 1)
    for j = 0 to 15
      k = j*4 + 3
      Glcd_Line(0, 0, 127, k, 2)
    next j
    Delay_ms(4000)
  loop until FALSE

end.
```

Hardware Connection



T6963C Graphic LCD Library

mikroBasic provides a library for drawing and writing on Toshiba T6963C Graphic LCD (changeable size).

Library Routines

```
T6963C_Init
T6963C_writeData
T6963C_writeCommand
T6963C_setPtr
T6963C_waitReady
T6963C_fill
T6963C_dot
T6963C_write_char
T6963C_write_text
T6963C_line
T6963C_rectangle
T6963C_box
T6963C_circle
T6963C_image
T6963C_sprite
T6963C_set_cursor
T6963C_clearBit
T6963C_setBit
T6963C_negBit
T6963C_displayGrPanel
T6963C_displayTxtPanel
T6963C_setGrPanel
T6963C_setTxtPanel
T6963C_panelFill
T6963C_grFill
T6963C_txtFill
T6963C_cursor_height
T6963C_graphics
T6963C_text
T6963C_cursor
T6963C_cursor_blink
T6963C_Init_240x128
T6963C_Init_240x64
```

T6963C_init

Prototype	**sub procedure** `T6963C_init(`**dim** `width, height, fntW` **as word, dim byref** `data` **as word, dim byref** `cntrl` **as word, dim** `wr, rd, cd, rst` **as word**`)`
Description	Initalizes the Graphic Lcd controller. This function must be called before all T6963C Library Routines. width - Number of horizontal (x) pixels in the display. height - Number of vertical (y) pixels in the display. fntW - Font width, number of pixels in a text character, must be set accordingly to the hardware. `data` - Address of the port on which the Data Bus is connected. `cntrl` - Address of the port on which the Control Bus is connected. `wr` - !WR line bit number in the *cntrl port. `rd` - !RD line bit number in the *cntrl port. `cd` - !CD line bit number in the *cntrl port. `rst` - !RST line bit number in the *cntrl port. Display RAM : The library doesn't know the amount of available RAM. The library cuts the RAM into panels : a complete panel is one graphics panel followed by a text panel, The programer has to know his hardware to know how much panel he has.
Requires	Nothing.
Example	

```
T6963C_init(240, 128, 8, PORTD, PORTB, 3, 2, 1, 5)
'   {*
'    * init display for 240 pixel width and 128 pixel height
'    * 8 bits character width
'    * data bus on PORTF
'    * control bus on PORTD
'    * bit 3 is !WR
'    * bit 2 is !RD
'    * bit 1 is !CD
'    * bit 5 is RST
'    *}
```

T6963C_writeData

Prototype	`sub procedure T6963C_writeData(dim data as byte)`
Description	Routine that writes data to T6963C controller.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_writeData(AddrL)`

T6963C_writeCommand

Prototype	`sub procedure T6963C_writeCommand(dim data as byte)`
Description	Routine that writes command to T6963C controller.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_writeCommand(T6963C_CURSOR_POINTER_SET)`

T6963C_setPtr

Prototype	`sub procedure T6963C_setPtr(dim p as word, dim c as byte)`
Description	This routine sets the memory pointer p for command c.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_setPtr(T6963C_grHomeAddr + start, T6963C_ADDRESS_POINTER_SET)`

T6963C_waitReady

Prototype	`sub procedure T6963C_waitReady`
Description	This routine pools the status byte, and loops until ready.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_waitReady`

T6963C_fill

Prototype	`sub procedure T6963C_fill(dim v as byte, dim start, len as word)`
Description	This routine fills length with bytes to controller memory from start address.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_fill(0x33,0x00FF,0x000F)`

T6963C_dot

Prototype	`sub procedure T6963C_dot(dim x, y as integer, dim color as byte)`
Description	This sets current text work panel. It writes string str row x line y. mode = T6963C_ROM_MODE_[OR|EXOR|AND].
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_dot(x0, y0, pcolor)`

T6963C_write_char

Prototype	`sub procedure T6963C_write_char(dim c, x, y, mode as byte)`
Description	This routine sets current text work panel. It writes char c row x line y. mode = T6963C_ROM_MODE_[OR|EXOR|AND]
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_write_char('A',22,23,AND)`

T6963C_write_text

Prototype	`sub procedure T6963C_write_text(dim byref str as byte[10], dim x, y, mode as byte)`
Description	This sets current text work panel. It writes string str row x line y. mode = T6963C_ROM_MODE_[OR|EXOR|AND]
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_write_text("GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR)`

T6963C_line

Prototype	`sub procedure T6963C_line(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	This routine current graphic work panel. It's draw a line from (x0, y0) to (x1, y1). pcolor = T6963C_[WHITE[BLACK]
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_line(0, 0, 239, 127, T6963C_WHITE)`

T6963C_rectangle

Prototype	`sub procedure T6963C_rectangle(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	It sets current graphic work panel. It draws the border of the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE)`

T6963C_box

Prototype	`sub procedure T6963C_box(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	This routine sets current graphic work panel. It draws a solid box in the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_box(0, 119, 239, 127, T6963C_WHITE)`

T6963C_circle

Prototype	`sub procedure T6963C_circle(dim x, y as integer, dim r as longint, dim pcolor as word)`
Description	This routine sets current graphic work panel. It draws a circle, center is (x, y), diameter is r. pcolor = T6963C_[WHITE[BLACK]
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_circle(120, 64, 110, T6963C_WHITE)`

T6963C_image

Prototype	`sub procedure T6963C_image(const pic as ^byte)`
Description	This routine sets current graphic work panel : It fills graphic area with picture pointer by MCU. MCU must fit the display geometry. For example : for a 240x128 display, MCU must be an array of (240/8)*128 = 3840 bytes .
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_image(my_image)`

T6963C_sprite

Prototype	`sub procedure T6963C_sprite(dim px, py as byte, const pic as ^byte, dim sx, sy as byte)`
Description	This routine sets current graphic work panel. It fills graphic rectangle area (px, py)-(px + sx, py + sy) witch picture pointed by MCU. Sx and sy must be the size of the picture. MCU must be an array of sx*sy bytes.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_sprite(76, 4, einstein, 88, 119) ' draw a sprite`

T6963C_set_cursor

Prototype	`sub procedure T6963C_set_cursor(dim x, y as byte)`
Description	This routine sets cursor row x line y.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_set_cursor(cposx, cposy)`

T6963C_clearBit

Prototype	`sub procedure T6963C_clearBit(dim b as byte)`
Description	Clear control bit.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_clearBit(b)`

T6963C_setBit

Prototype	`sub procedure T6963C_setBit(dim b as byte)`
Description	Set control bit.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_setBit(b)`

T6963C_negBit

Prototype	`sub procedure T6963C_negBit(b as byte)`
Description	Neg control bit.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_negBit(b)`

T6963C_displayGrPanel

Prototype	`sub procedure T6963C_displayGrPanel(dim n as word)`
Description	Display graphic panel number n.
Requires	GLCD needs to be initialized, see `T6963C_init`.
Example	`T6963C_displayGrPanel(n)`

T6963C_displayTxtPanel

Prototype	`sub procedure T6963C_displayTxtPanel(dim n as word)`
Description	Display text panel number n.
Requires	GLCD needs to be initialized, see `T6963C_init`.
Example	`T6963C_displayTxtPanel(n)`

T6963C_setGrPanel

Prototype	`sub procedure T6963C_setGrPanel(dim n as word)`
Description	Compute graphic start address for panel number n.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_setGrPanel(n)`

T6963C_setTxtPanel

Prototype	`sub procedure T6963C_setTxtPanel(dim n as word)`
Description	Compute text start address for panel number n.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_setTxtPanel(n)`

T6963C_panelFill

Prototype	`sub procedure T6963C_panelFill(dim v as word)`
Description	Fill full #n panel with v bitmap (0 to clear).
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_panelFill(v)`

T6963C_grFill

Prototype	`sub procedure T6963C_grFill(dim v as word)`
Description	Fill graphic #n panel with v bitmap (0 to clear).
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_grFill(v)`

T6963C_txtFill

Prototype	`sub procedure T6963C_txtFill(dim v as word)`
Description	Fill text #n panel with char v + 32 (0 to clear).
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_txtFill(v)`

T6963C_cursor_height

Prototype	`sub procedure T6963C_cursor_height(dim n as word)`
Description	Set cursor size.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_cursor_height(n)`

T6963C_graphics

Prototype	`sub procedure T6963C_graphics(dim n as word)`
Description	Set graphics on/off.
Requires	GLCD needs to be initialized, see `T6963C_init`.
Example	`T6963C_graphics(1)`

T6963C_text

Prototype	`sub procedure T6963C_text(dim n as word)`
Description	Set text on/off.
Requires	GLCD needs to be initialized, see `T6963C_init`.
Example	`T6963C_text(1)`

T6963C_cursor

Prototype	`sub procedure T6963C_cursor(dim n as word)`
Description	Set cursor on/off.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_cursor(1)`

T6963C_cursor_blink

Prototype	`sub procedure T6963C_cursor_blink(dim n as word)`
Description	Set cursor blink on/off.
Requires	Ports must be initialized. See `T6963C_init`.
Example	`T6963C_cursor_blink(0)`

T6963C_Init_240x128

Prototype	`sub procedure T6963C_Init_240x128`
Description	Initialize T6963C based GLCD (240x128 pixels) with default settings for mE GLCD's.
Example	`T6963C_Init_240x128`

T6963C_Init_240x64

Prototype	`sub procedure T6963C_Init_240x64`
Description	Initialize T6963C based GLCD (240x64 pixels) with default settings for mE GLCD's.
Example	`T6963C_Init_240x64`

Library Example

The following drawing demo tests advanced routines of T6963C GLCD library.

```
program T6963C
include "__Lib_T6963c"
include "bitmap"
include "bitmap2"
dim   panel as byte            ' current panel
           i as word                ' general purpose register
        curs as byte             ' cursor visibility
       cposx,
       cposy as word   ' cursor x-y position
main:
  TRISC   = 0xFFFF
  TRISB   = 0x0000
  PORTD   = 0
  TRISD   = 0

   T6963C_init(240, 128, 8, PORTD, PORTB, 3, 2, 1, 5)

'   {*
'    * enable both graphics and text display at the same time
'    *}

  T6963C_graphics(1)
  T6963C_text(1)


//continues...
```

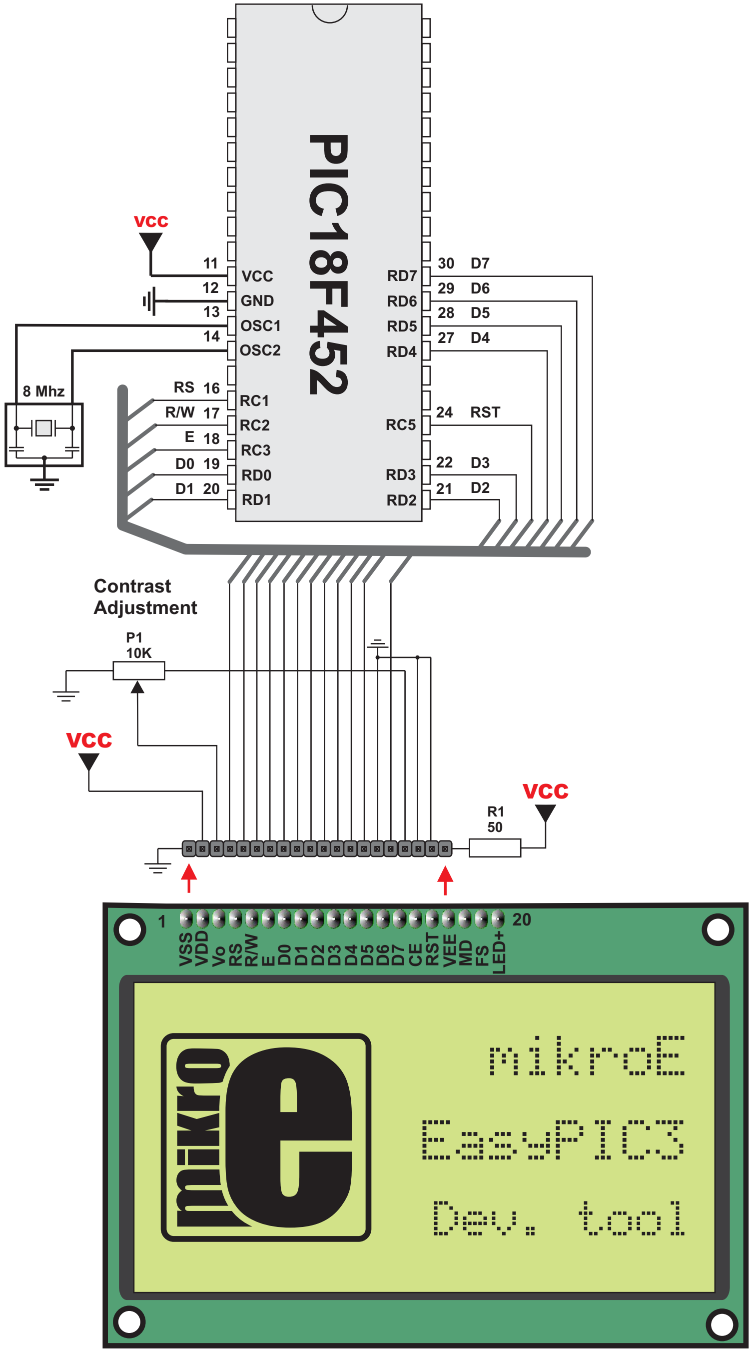
```
//continued...

  panel =0
  i =0
  curs =0
  cposy =0
  cposx =0

'  {*
'   * text messages
'   *}

  T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0, 0,
T6963C_ROM_MODE_XOR)
  T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mE", 0, 15,
T6963C_ROM_MODE_XOR)

' {*
'   * cursor
'   *}

  T6963C_cursor_height(8)        ' 8 pixel height
  T6963C_set_cursor(0, 0)        ' move cursor to top left
  T6963C_cursor(0)                ' cursor off

'  {*
'   * draw rectangles
'   *}

  T6963C_rectangle(0, 0, 239, 127, T6963C_WHITE)
  T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE)
  T6963C_rectangle(40, 40, 199, 87, T6963C_WHITE)
  T6963C_rectangle(60, 60, 179, 67, T6963C_WHITE)

'  {*
'   * draw a cross
'   *}

  T6963C_line(0, 0, 239, 127, T6963C_WHITE)
  T6963C_line(0, 127, 239, 0, T6963C_WHITE)

'  {*
'   * draw solid boxes
'   *}

  T6963C_box(0, 0, 239, 8, T6963C_WHITE)
  T6963C_box(0, 119, 239, 127, T6963C_WHITE)
```

```
//continued...

'  {*
'  * draw circles
'   *}

  T6963C_circle(120, 64, 10, T6963C_WHITE)
  T6963C_circle(120, 64, 30, T6963C_WHITE)
  T6963C_circle(120, 64, 50, T6963C_WHITE)
  T6963C_circle(120, 64, 70, T6963C_WHITE)
  T6963C_circle(120, 64, 90, T6963C_WHITE)
  T6963C_circle(120, 64, 110, T6963C_WHITE)
  T6963C_circle(120, 64, 130, T6963C_WHITE)

  T6963C_sprite(76, 4, einstein, 88, 119)
' draw a sprite

  T6963C_setGrPanel(1)            ' select other graphic panel

  T6963C_image(mikroPascal_logo_glcd_bmp)
' fill the graphic screen with a picture

  while true
'       {*
'        * if RC0 is pressed, toggle the display between positive
' and negative mode
'        *}
      if(PORTC.0 <> 0) then
          'PORTC.1 =PORTC.1 xor 1
          Delay_ms(300)
      end if

'       {*
'        * if RC1 is pressed, toggle the display between graphic
          panel 0 and graphic 1
'        *}
        if(PORTC.1 <> 0) then
            panel =panel + 1
            panel =panel and 1
            T6963C_displayGrPanel(panel)
            Delay_ms(300)
          end if


//continues...
```

```
//continued...

'       {*
'        * if RC2 is pressed, display only graphic panel
'        *}

          if(PORTC.2 <> 0) then
              T6963C_graphics(1)
              T6963C_text(0)
              Delay_ms(300)
          end if

'       {*
'        * if RC3 is pressed, display only text panel
'        *}
           if(PORTC.3 <> 0) then
                T6963C_graphics(0)
                T6963C_text(1)
                Delay_ms(300)
           end if

'       {*
'        * if RC4 is pressed, display text and graphic panels
'        *}
            if(PORTC.4 <> 0) then
                T6963C_graphics(1)
                T6963C_text(1)
                Delay_ms(300)
            end if

//continues...
```

```
//continued...

'       {*
'        * if RC5 is pressed, change cursor
'        *}
              if(PORTC.5 <> 0) then
                  curs =curs + 1
                  if(curs = 3) then
                    curs =0
                  end if
                  select case curs
                    case 0
                        T6963C_cursor(0)

                    case 1
                          T6963C_cursor(1)
                          T6963C_cursor_blink(1)

                    case 2
                          T6963C_cursor(1)
                          T6963C_cursor_blink(0)
                  end select
                  Delay_ms(300)
              end if
'       {*
'        * move cursor, even if not visible
'        *}
      cposx =cposx + 1
      if(cposx = T6963C_txtCols) then
          cposx =0
          cposy =cposy + 1
          if(cposy = (T6963C_grHeight div
T6963C_CHARACTER_HEIGHT)) then
            cposy =0
          end if
      end if
      T6963C_set_cursor(cposx, cposy)
      Delay_ms(100)
  wend
end.
```

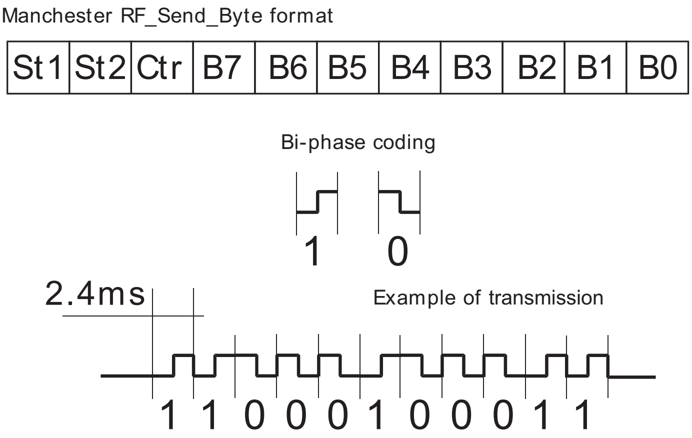
Hardware Connection



Toshiba T6963C Graphic LCD (240x128)

Manchester Code Library

mikroBasic provides a library for handling Manchester coded signals. 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).



Notes: 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.

Library Routines

```
Man_Receive_Config
Man_Receive_Init
Man_Receive
Man_Send_Config
Man_Send_Init
Man_Send
Man_Synchro
```

Man_Receive_Config

Prototype	`sub procedure Man_Receive_Config(dim byref port as byte, dim rxpin as byte)`
Description	The procedure prepares PIC for receiving signal. You need to specify the `port` and `rxpin` (0–7) of input signal. In case of multiple errors on reception, you should call `Man_Receive_Init` once again to enable synchronization.
Example	`Man_Receive_Config(PORTD, 6)`

Man_Receive_Init

Prototype	`sub procedure Man_Receive_Init(dim byref port as byte)`
Description	The procedure prepares PIC for receiving signal. You need to specify the `port`; `rxpin` is pin 6 by default. In case of multiple errors on reception, you should call `Man_Receive_Init` once again to enable synchronization.
Example	`Man_Receive_Init(PORTD)`

Man_Receive

Prototype	`sub function Man_Receive(dim byref error as byte) as byte`
Returns	Returns one byte from signal.
Description	procedure extracts one byte from signal. If signal format does not match the expected, `error` flag will be set to 255.
Requires	To use this function, you must first prepare the PIC for receiving. See `Man_Receive_Config` or `Man_Receive_Init`.
Example	`temp = Man_Receive(error)` `if error = true then ... ' error handling`

Man_Send_Config

Prototype	`sub procedure Man_Send_Config(dim byref port as byte, dim txpin as byte)`
Description	The function prepares PIC for sending signal. You need to specify `port` and `txpin` (0–7) for outgoing signal. Baud rate is const 500 bps.
Example	`Man_Send_Config(PORTD, 0)`

Man_Send_Init

Prototype	`sub procedure Man_Send_Init(dim byref port as byte)`
Description	The function prepares PIC for sending signal. You need to specify `port` for outgoing signal; `txpin` is pin 0 by default. Baud rate is const 500 bps.
Example	`Man_Send_Init(PORTD)`

Man_Send

Prototype	`sub procedure Man_Send(dim data as byte)`
Description	Sends one byte (`data`).
Requires	To use this function, you must first prepare the PIC for sending. See `Man_Send_Config` or `Man_Send_Init`.
Example	`Man_Send(msg)`

Man_Synchro

Prototype	`sub function Man_Synchro as byte`
Returns	Half of the manchester bit length, given in multiples of 10us.
Description	This function returns half of the manchester bit length. The length is given in multiples of 10us. It is assumed that one bit lasts no more than 255*10us = 2550 us.
Requires	To use this function, you must first prepare the PIC for sending. See `Man_Send_Config` or `Man_Send_Init`.
Example	`man_len = Man_Synchro`

Library Example

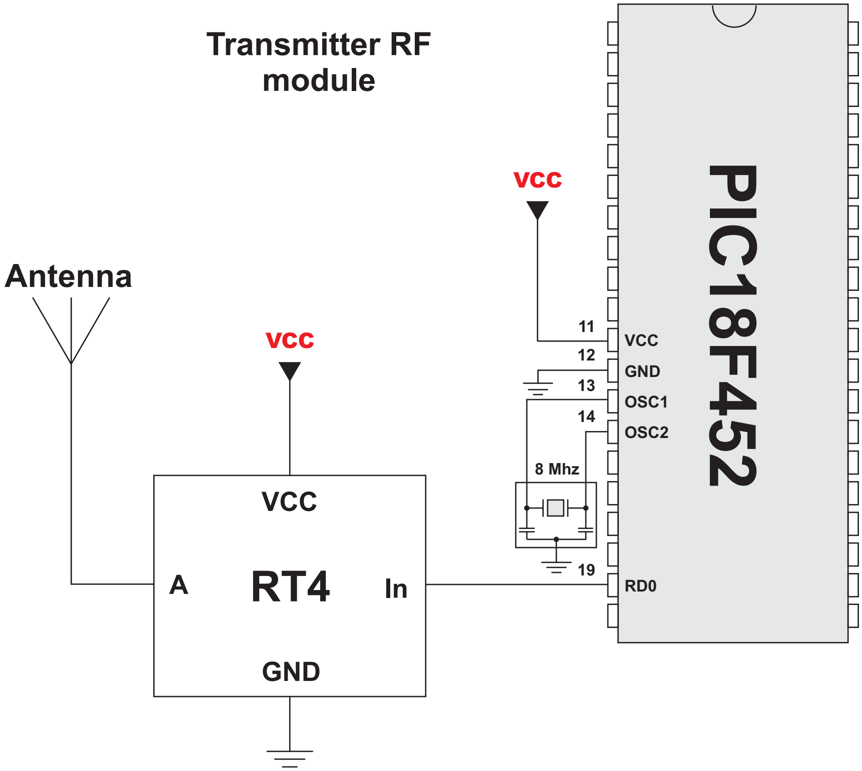
The following example transmits message in Manchester code. Message is delimited by markers `$0B` and `$0E`.

```
program RF_TX
dim i as byte
dim msg as string[ 20]

main:
  msg = "mikroElektronika"
  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
    for i = 1 to Strlen(msg)
      Man_Send(msg[ i] )      ' Send char
      Delay_ms(90)
    next i
    Man_Send($0E)             ' Send end marker
    Delay_ms(1000)
  wend
end.
```

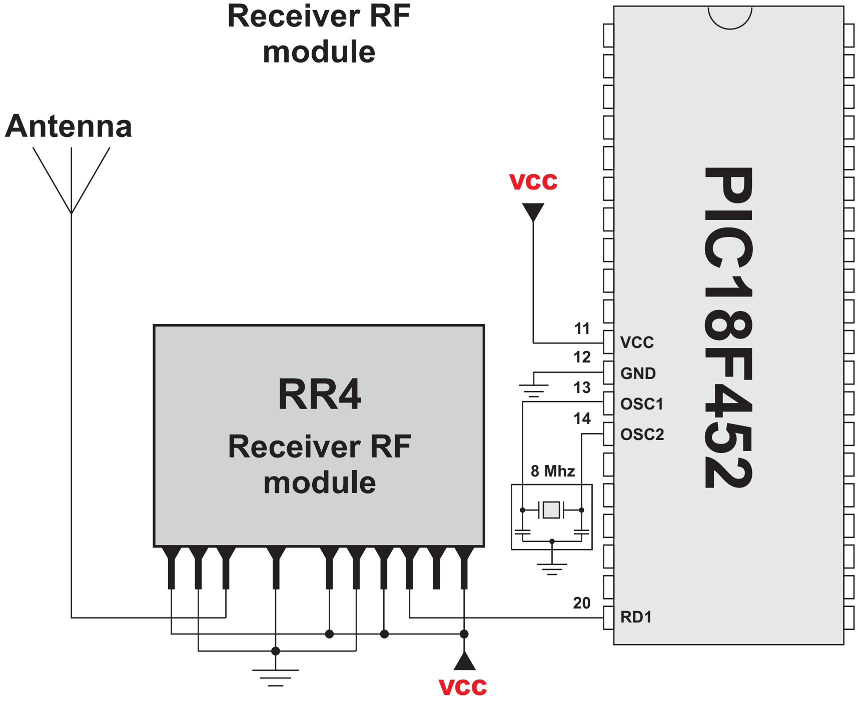
The following code receives messages sent by the previous example, and prints it on LCD. Each error in the received string will be indicated by a quotation mark.

```
program RRX
dim error1, ErrorCount, temp as byte

main:
  ErrorCount = 0
  TRISB      = 0
  CMCON = 7
  ' VRCON = 0                   ' Uncomment the line for PIC16
  Lcd_Init(PORTB)               ' Initialize LCD on PORTB
  Lcd_Cmd(LCD_CLEAR)
  Man_Receive_Config(PORTA,3)   ' Configure and synchronize receiver

  while true do
    Lcd_Cmd(Lcd_FIRST_ROW)

    while true do               ' Wait for the start marker
      temp = Man_Receive(error1)
      if temp = $0B then
        break
      end if                    ' We got the starting sequence
      if error1 then                  ' Exit so we do not loop forever
        break
      end if
    wend

    do
      temp = Man_Receive(error1) ' Attempt byte receive
      if error1 = true then
        Lcd_Chr_Cp(63)          ' ASCII for '?'
        Inc(ErrorCount)
        if ErrorCount > 20 then
          Man_Receive_Init(PORTA)
          ' alternative:
          ' temp = Man_Synchro
          ErrorCount = 0
        end if

      else
        if temp <> $0E then     ' Don't print the end marker on LCD
          Lcd_Chr_Cp(temp)
        end if
      Delay_ms(20)
      end if
    loop until temp = $0E
  wend
end.
```

Hardware Connection



Receiver RF module



Multi Media Card Library

The Multi Media Card (MMC) is a flash memory card standard. MMC cards are currently available in sizes up to and including 1 GB, and are used in cell phones, mp3 players, digital cameras, and PDA's.
mikroBasic provides a library for accessing data on Multi Media Card via SPI communication.This library also supports SD(Secure Digital) memory cards.

Secure Digital (SD) is a flash memory card standard, based on the older Multi Media Card (MMC) format. SD cards are currently available in sizes of up to and including 2 GB, and are used in cell phones, mp3 players, digital cameras.

Notes:

- Library works with PIC18 family only;
- Library functions create and read files from the root directory only;
- Library functions populate both FAT1 and FAT2 tables when writing to files, but the file data is being read from the FAT1 table only; i.e. there is no recovery if FAT1 table is corrupted.
- `Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH)` must be called before initializing `Mmc_Init` and `Mmc_Fat_Init`.

Library Routines

```
Mmc_Init
Mmc_Read_Sector
Mmc_Write_Sector
Mmc_Read_Cid
Mmc_Read_Csd

Mmc_Fat_Init
Mmc_Fat_Assign
Mmc_Fat_Reset
Mmc_Fat_Rewrite
Mmc_Fat_Append
Mmc_Fat_Read
Mmc_Fat_Write
Mmc_Fat_Set_File_Date
Mmc_Fat_Get_File_Date
Mmc_Fat_Get_File_Size
Mmc_Fat_Get_Swap_File
```

Mmc_Init

Prototype	`sub function Mmc_Init(dim byref port as byte, dim pin as byte) as byte`
Returns	Returns 0 if MMC card is present and successfully initialized, otherwise returns 1.
Description	Initializes hardware SPI communication; parameters `port` and `pin` designate the CS line used in the communication (parameter `pin` should be 0..7). The function returns 0 if MMC card is present and successfully initialized, otherwise returns 1. `Mmc_Init` needs to be called before using other functions of this library.
Requiress	`Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH)` must be called before initializing `Mmc_Init`.
Example	`error = Mmc_Init(PORTC, 2)   ' Init with CS line at RC2`

Mmc_Read_Sector

Prototype	`sub function Mmc_Read_Sector(dim sector as longint, dim byref data as byte[ 512] ) as byte`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads one sector (512 bytes) from MMC card at sector address `sector`. Read data is stored in the array `data`. Function returns 0 if read was successful, or 1 if an error occurred.
Requires	Library needs to be initialized, see `Mmc_Init`.
Example	`error = Mmc_Read_Sector(sector, data)`

Mmc_Write_Sector

Prototype	`sub function Mmc_Write_Sector(dim sector as longint, dim byref data as byte[512]) as byte`
Returns	Returns 0 if write was successful; returns 1 if there was an error in sending write command; returns 2 if there was an error in writing.
Description	Function writes 512 bytes of data to MMC card at sector address `sector`. Function returns 0 if write was successful, or 1 if there was an error in sending write command, or 2 if there was an error in writing.
Requires	Library needs to be initialized, see `Mmc_Init`.
Example	`error = Mmc_Write_Sector(sector, data)`

Mmc_Read_Cid

Prototype	`sub function Mmc_Read_Cid(dim byref data_for_registers as byte[512]) as byte`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CID register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see `Mmc_Init`.
Example	`error = Mmc_Read_Cid(data)`

Mmc_Read_Csd

Prototype	`sub function Mmc_Read_Csd(dim byref data_for_registers as byte[512]) as byte`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CSD register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see `Mmc_Init`.
Example	`error = Mmc_Read_Csd(data)`

Mmc_Fat_Init

Prototype	`sub function Mmc_Fat_Init(dim byref mmcport as byte, dim mmcpin as byte) as byte`
Returns	Returns non-zero value if MMC card is present and successfully initialized, otherwise returns 0.
Description	Initializes hardware SPI communication; designated CS line for communication is given by parameters `mmcport` and `mmcpin`. The function returns a non-zero value if MMC card is present and successfully initialized, otherwise it returns 0. This function needs to be called before using other functions of MMC FAT library.
Requires	`Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH)` must be called before initializing `Mmc_Fat_Init`.
Example	`success = Mmc_Fat_Init(PORTC, 2)`

Mmc_Fat_Assign

Prototype	`sub function Mmc_Fat_Assign(dim byref filename as char[ 11] ,` `dim create_file as byte) as byte`
Returns	The function returns non-zero value if the file that is specified by filename was been found or newly created, otherwise it returns 0.
Description	This function designates ("assigns") the file we'll be working with. The function looks for the file specified by the `filename` in the root directory. If the file is found, routine will initialize it by getting its start sector, size, etc. If the file is not found, an empty file will be created with the given name, if allowed. Whether the new file will be created or not is controlled by the parameter `create_file` - setting it to zero will prevent creation of new file, while giving it any non-zero value will do the opposite. The `filename` must be 8 + 3 characters in uppercase.
Requires	Library needs to be initialized; see `Mmc_Fat_Init`.

Example	`' Assign the file "EXAMPLE1.TXT" in the root directory of MMC.`
	`' If the file is not found, routine will create one.`
	`' In this case, function return value will allways be non-zero`
	`Mmc_Fat_Assign("EXAMPLE1TXT", 1)`
	
	`' Assign the file "EXAMPLE2.TXT" in the root directory of MMC.`
	`' If the file is not found, routine will NOT create new one.`
	`file_found = Mmc_Fat_Assign("EXAMPLE2TXT", 0)`

Mmc_Fat_Reset

Prototype	`sub procedure Mmc_Fat_Reset(dim byref size as longint)`
Description	Procedure resets the file pointer (moves it to the start of the file) of the assigned file, so that the file can be read. Parameter `size` stores the size of the assigned file, in bytes.
Requires	The file must be assigned, see `Mmc_Fat_Assign`.
Example	`Mmc_Fat_Reset(size)`

Mmc_Fat_Rewrite

Prototype	`sub procedure Mmc_Fat_Rewrite`
Description	Procedure resets the file pointer and clears the assigned file, so that new data can be written into the file.
Requires	The file must be assigned, see `Mmc_Fat_Assign`.
Example	`Mmc_Fat_Rewrite`

Mmc_Fat_Append

Prototype	`sub procedure Mmc_Fat_Append`
Description	The procedure moves the file pointer to the end of the assigned file, so that data can be appended to the file.
Requires	The file must be assigned, see `Mmc_Fat_Assign`.
Example	`Mmc_Fat_Append`

Mmc_Fat_Read

Prototype	`sub procedure Mmc_Fat_Read(dim byref data as byte)`
Description	Procedure reads the byte at which the file pointer points to and stores data into parameter `data`. The file pointer automatically increments with each call of `Mmc_Fat_Read`.
Requires	The file must be assigned, see `Mmc_Fat_Assign`. Also, file pointer must be initialized; see `Mmc_Fat_Reset`.
Example	`Mmc_Fat_Read(mydata)`

Mmc_Fat_Write

Prototype	`sub procedure Mmc_Fat_Write(dim byref fdata as char[ 512] , dim datalen as word)`
Description	Procedure writes a chunk of bytes (`fdata`) to the currently assigned file, at the position of the file pointer.
Requires	The file must be assigned, see `Mmc_Fat_Assign`. Also, file pointer must be initialized; see `Mmc_Fat_Append` or `Mmc_Fat_Rewrite`.
Example	`Mmc_Fat_Write(txt,255)` `Mmc_Fat_Write(“Hello world”,255)`

Mmc_Fat_Set_File_Date

Prototype	`sub procedure Mmc_Fat_Set_File_Date(dim year as word, dim month, day, hours, min, sec as byte)`
Description	Writes system timestamp to a file. Use this routine before each writing to file; otherwise, the file will be appended an unknown timestamp.
Requires	File pointer must be initialized; see `Mmc_Fat_Assign` and `Mmc_Fat_Reset`.
Example	`' April 1st 2005, 18:07:00` `Mmc_Fat_Set_File_Date(2005, 4, 1, 18, 7, 0)`

Mmc_Fat_Get_File_Date

Prototype	`sub procedure Mmc_Fat_Get_File_Date(dim byref year as word, dim byref month, day, hours, min, sec as byte)`
Description	Retrieves date and time for the currently selected file. Seconds are not being retrieved since they are written in 2-sec increments.
Requires	The file must be assigned, see `Mmc_Fat_Assign`.
Example	

```
' get Date/time of file
dim yr as word
dim mnth, dat, hrs, mins as byte
...
file_Name = "MYFILEABTXT"
Mmc_Fat_Assign(file_Name)
Mmc_Fat_Get_File_Date(yr, mnth, dat, hrs, mins)
```

Mmc_Fat_Get_File_Size

Prototype	`sub function Mmc_Fat_Get_File_Size as longint`
Returns	The size of active file (in bytes).
Description	Retrieves size for currently selected file.
Requires	The file must be assigned, see `Mmc_Fat_Assign`.
Example	

```
' get file size
dim yr as word
dim mnth, dat, hrs, mins as byte
...
file_name = "MYFILEXXTXT"
Mmc_Fat_Assign(file_name)
mmc_size = Mmc_Fat_Get_File_Size()
```

Mmc_Fat_Get_Swap_File

Prototype	**sub function** `Cf_Fat_Get_Swap_File(`**dim** `sectors_cnt` **as longint**`)`**as longint**
Returns	No. of start sector for the newly created swap file, if swap file was created; otherwise, the function returns zero.
Description	This function is used to create a swap file on the MMC/SD media. It accepts as sectors_cnt argument the number of consecutive sectors that user wants the swap file to have. During its execution, the function searches for the available consecutive sectors, their number being specified by the sectors_cnt argument. If there is such space on the media, the swap file named MIKROSWP.SYS is created, and that space is designated (in FAT tables) to it. The attributes of this file are: system, archive and hidden, in order to distinct it from other files. If a file named MIKROSWP.SYS already exists on the media, this function deletes it upon creating the new one. The purpose of the swap file is to make reading and writing to MMC/SD media as fast as possible, by using the Mmc_Read_Sector and Mmc_Write_Sector functions directly, without potentially damaging the FAT system. Swap file can be considered as a "window" on the media where user can freely write/read the data, in any way (s)he wants to. Its main purpose in mikroBasic library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a "normal" file, and formatted in the most suitable way.
Requires	Ports must be initialized for FAT operations with MMC. See Mmc_Fat_Init.
Example	see below

Example

```
'-------------- Tries to create a swap file, whose size will be
'at least 100 sectors.
'If it succeeds, it sends the No. of start sector over USART

sub procedure M_Create_Swap_File
  size = Mmc_Fat_Get_Swap_File(100)
  if (size) then
    Usart_Write($AA)
    Usart_Write(Lo(size))
    Usart_Write(Hi(size))
    Usart_Write(Higher(size))
    Usart_Write(Highest(size))
    Usart_Write($AA)
   end if
end sub
```

Library Example

The following code tests MMC library routines. First, we fill the buffer with 512 “M” characters and write it to sector 56; then, we repeat the sequence with character “E” at sector 56. Finally, we read the sectors 55 and 56 to check if the write was successful.

```
program mmc_test
dim tmp as byte
dim i as word
dim data as byte[ 512]

main:
  Usart_Init(9600)

  ' Initialize SPI
Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH)

  tmp = Mmc_Init(PORTC, 2)                ' Initialize ports

  for i = 0 to 512                        ' Fill the buffer with the "M" char
    data[ i] = "M"
  next i

  tmp = Mmc_Write_Sector(55, data)        ' Write it to MMC card, sector 55

  for i = 0 to 512                        ' Fill the buffer with the "E" char
    data[ i] = "E"
  next i

  tmp = Mmc_Write_Sector(56, data)        ' Write it to MMC card, sector 56

  '** Verify: **

  tmp = Mmc_Read_Sector(55, data)         ' Read from sector 55

  if tmp = 0 then                         ' Send 512 bytes from buffer to USART
    for i = 0 to 512
      Usart_Write(data[ i] )
    next i
  end if

  tmp = Mmc_Read_Sector(56, data)         ' Read from sector 56

  if tmp = 0 then                         ' Send 512 bytes from buffer to USART
    for i = 0 to 512
      Usart_Write(data[ i] )
    next i
  end if
end.
```

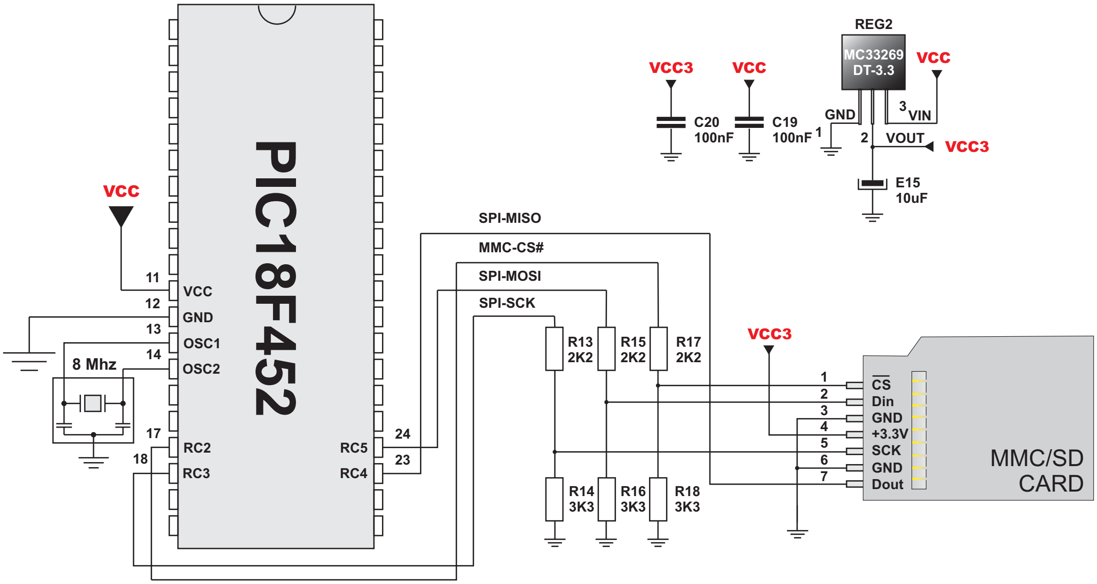
Library Example

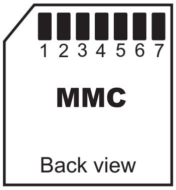
The next program tests MMC FAT routines. First, we create 5 different files in the root of MMC card, and fill with some information. Then, we read the files and send them via USART for a check.

```
program MMC_FAT_Test

const FAT_ERROR as string[ 20]  = "FAT16 not found"
dim filename as string[ 14]
dim tmp, character, j as byte
dim size, i as longint
dim aux as string[ 5]
dim msg as string[ 100]

main:

  Usart_Init(19200)               ' Set up USART for the read of the files
  ' Initialize SPI
Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH)

  tmp = Mmc_Fat_Init(PORTC, 2)    ' Try to locate the FAT

  if tmp <> 0 then
    for tmp = 0 to Strlen(FAT_ERROR) - 1
      Usart_Write(FAT_ERROR[ tmp] )
    next tmp
  end if

  j = 1
  '** Write test, 5 files **

  for j = 1 to 5                  ' We want 5 files on the MMC card
    filename = "MYFILE0xTXT"      ' File names, e.g. "MYFILE01.TXT"
    filename[ 7]  = j + 48        ' Set number 1, 2, 3, 4, or 5

    Mmc_Fat_Assign(filename, 1)   ' Create the file, if not found
    Mmc_Fat_Rewrite()             ' Clear the file and prepare for writing

    ' Form the text to be written
    aux = "   "
    aux[ 0]  = j + 48
    msg = "This is a test file, no." + aux

    Mmc_Fat_Write(msg)            ' Write data to the assigned file
  next j

' continues ..
```

```
' .. continued

  '** Append test **

  ' Now let's add more data to the same files
  for j = 1 to 5
    filename = "MYFILE0xTXT"
    filename[ 7]  = j + 48

    Mmc_Fat_Assign(filename, 1)                 ' Find the file and "assign" it
    Mmc_Fat_Append()                            ' Prepare for appending

    ' Form a text to be written
    aux = " "
    aux[ 0]  = j + 48
    msg = "Append test, try " + aux

    Mmc_Set_File_Date(2005,5,j,12,47,12)        ' Test the date function

    Mmc_Fat_Write(msg)                          ' Write data to the assigned file

    '** Read test **

    Mmc_Fat_Reset(size)                         ' Take the size of the file

    ' Send whole file to USART, char by char
    for i = 1 to size
      Mmc_Fat_Read(character)
      Usart_Write(character)
    next i

  next j

end.
```

Hardware Connection





OneWire Library

OneWire library provides routines for communication via OneWire bus, for example with DS1820 digital thermometer. This is a Master/Slave protocol, and all the cabling required is a single wire. Because of the hardware configuration it uses (single pullup and open collector drivers), it allows for the slaves even to get their power supply from that line.

Some basic characteristics of this protocol are:

- single master system,
- low cost,
- low transfer rates (up to 16 kbps),
- fairly long distances (up to 300 meters),
- small data transfer packages.

Each OneWire device also has a unique 64-bit registration number (8-bit device type, 48-bit serial number and 8-bit CRC), so multiple slaves can co-exist on the same bus.

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

Library Routines

```
Ow_Reset
Ow_Read
Ow_Write
```

Ow_Reset

Prototype	`sub function Ow_Reset(dim byref port as byte, pin as byte) as byte`
Returns	Returns 0 if DS1820 is present, 1 if not present.
Description	Issues OneWire reset signal for DS1820. Parameters `port` and `pin` specify the location of DS1820.
Requires	Works with Dallas DS1820 temperature sensor only.
Example	`Ow_Reset(PORTA, 5)   ' reset DS1820 connected to the RA5 pin`

Ow_Read

Prototype	`sub function Ow_Read(dim byref port as byte, dim pin as byte) as byte`
Returns	Data read from an external device over the OneWire bus.
Description	Reads one byte of data via the OneWire bus.
Example	`tmp = Ow_Read(PORTA, 5)`

Ow_Write

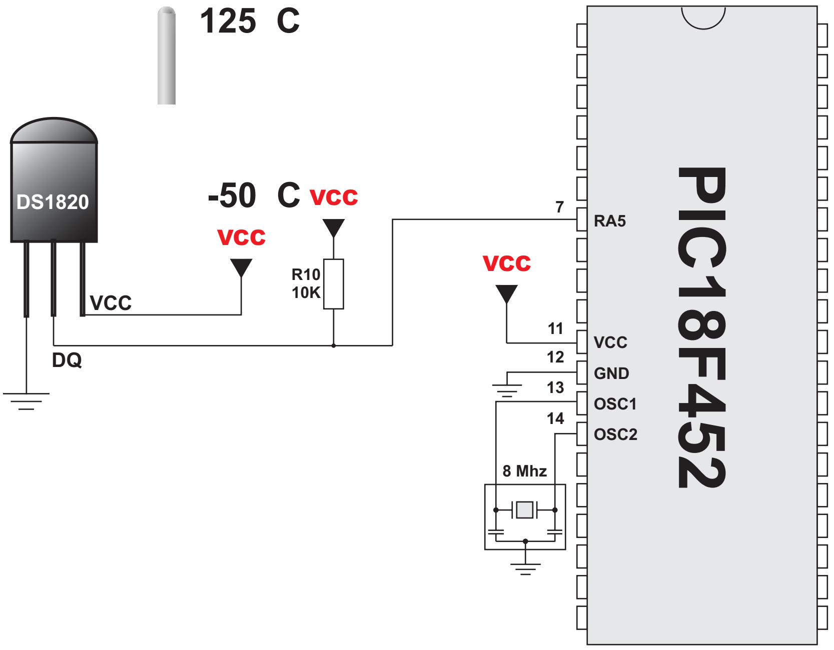
Prototype	`sub procedure Ow_Write(dim byref port as byte, dim pin, par as byte)`
Description	Writes one byte of data (argument `par`) via OneWire bus.
Example	`Ow_Write(PORTA, 5, $CC)`

Library Example

The example reads the temperature from DS1820 sensor connected to RA5. Temperature value is continually displayed on LCD.

```
program onewire

dim i, j1, j2 as byte
     text       as string[ 6]
     tmp_sign   as byte
main:
    text    = "Temp:"
    adcon1 = 255                       ' configure RA5 pin as digital I/O
    PORTA  = 255                       ' initialize porte to 255
    PORTD  =   0                       ' initialize portb to 255
    TRISA  = 255                       ' designate porte as input
    TRISD  =   0                       ' designate portb as output
    Lcd_Init(PORTD)
    lcd_cmd(LCD_CURSOR_OFF)
    lcd_out(1, 1, text)
    lcd_chr(2, 12, 223)               ' 'degree' character
    lcd_chr(2, 13, "C")
    while TRUE
      ow_reset(PORTA, 5)              ' onewire reset signal
      ow_write(PORTA, 5, $CC)        ' issue command to DS1820
      ow_write(PORTA, 5, $44)        ' issue command to DS1820
      delay_us(120)
      i = ow_reset(PORTA, 5)
      ow_write(PORTA, 5, $CC)        ' issue command to DS1820
      ow_write(PORTA, 5, $BE)        ' issue command to DS1820
      j1 = ow_read(PORTA, 5)          ' get result
      j2 = ow_read(PORTA, 5)' get result (assuming the temperature is positive)
      if j2 = $FF then
        tmp_sign = "-"                 ' temperature sign
        j1= j1 or $FF                  ' complement of two
        j1= j1 + $01
      else
        tmp_sign = "+"
      end if
      j2 = (j1 and $01) * 5           ' Get decimal value
      j1 = j1 >> 1                     ' Get temp value
      ByteToStr(j1, text)             ' whole number
      lcd_chr(2, 7, tmp_sign)
      lcd_chr(2, 8, text[ 1] )
      lcd_chr(2, 9, text[ 2] )
      lcd_chr(2, 10, 46)              ''.'
      ByteToStr(j2, text)             ' decimal
      lcd_chr(2, 11, text[ 2] )
      Delay_ms(500)
    wend                               'endless loop
end.
```

Hardware Connection



PS/2 Library

mikroBasic provides a library for communicating with common PS/2 keyboard. The library does not utilize interrupts for data retrieval, and requires oscillator clock to be 6MHz and above.

Please note:
- The pins to which a PS/2 keyboard is attached should be connected to pull-up resistors.
- Although PS/2 is a two-way communication bus, this library does not provide PIC-to-keyboard communication; e.g. the Caps Lock LED will not turn on if you press the Caps Lock key.

Library Routines

```
Ps2_Init
Ps2_Config
Ps2_Key_Read
```

Ps2_Init

Prototype	`sub procedure Ps2_Init(dim byref port as byte)`
Description	Initializes port for work with PS/2 keyboard, with default pin settings. Port pin 0 is Data line, and port pin 1 is Clock line. You need to call either Ps2_Init or Ps2_Config before using other routines of PS/2 library.
Requires	Both Data and Clock lines need to be in pull-up mode.
Example	`Ps2_Init(PORTB)`

Ps2_Config

Prototype	`sub procedure Ps2_Config(dim byref port as word, dim clock as word, dim data as word)`
Description	Initializes port for work with PS/2 keyboard, with custom pin settings. Parameters clock and data specify pins of port for Clock line and Data line, respectively. Clock and Data need to be in range 0..7 and cannot point to the same pin. You need to call either Ps2_Init or Ps2_Config before using other routines of PS/2 library.
Requires	Both Data and Clock lines need to be in pull-up mode.
Example	`Ps2_Config(PORTB, 2, 3)`

Ps2_Key_Read

Prototype	`sub function Ps2_Key_Read(dim byref value, special, pressed as byte) as byte`
Returns	Returns 1 if reading of a key from the keyboard was successful, otherwise returns 0.
Description	The procedure retrieves information about key pressed. Parameter `value` holds the value of the key pressed. For characters, numerals, punctuation marks, and space, value will store the appropriate ASCII value. Procedure "recognizes" the function of Shift and Caps Lock, and behaves appropriately. Parameter `special` is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of these, special will be set to 1, otherwise 0. Parameter `pressed` is set to 1 if the key is pressed, and 0 if released.
Requires	PS/2 keyboard needs to be initialized; see `Ps2_Init`.
Example	

```
' Press Enter to continue:
do
  if Ps2_Key_Read(val, spec, press) = 1 then
    if (val = 13) and (spec = 1) then
      break
    end if
  end if
loop until FALSE
```

Library Example

This simple example reads values of keys pressed on PS/2 keyboard and sends them via USART.

```
program ps2_test
dim keydata, special, down as byte

main:
  CMCON  = $07             ' Disable analog comparators (comment this for P18)
  INTCON = 0               ' Disable all interrupts
  Ps2_Init(PORTA)          ' Init PS/2 Keyboard on PORTA
  Delay_ms(100)            ' Wait for keyboard to finish

  do
    if Ps2_Key_Read(keydata, special, down) = 1 then
      if (down = 1) and (keydata = 16) then   ' Backspace
        ' ...do something with a backspace...
      else
        if (down = 1) and (keydata = 13) then   ' Enter
          Usart_Write(13)
        else
          if (down = 1) and (special = 0) and (keydata <> 1) then
            Usart_Write(keydata)
          end if
        end if
      end if
    end if
    Delay_ms(10)     ' debounce
  loop until FALSE
end.
```

PWM Library

CCP module is available with a number of PICmicros. mikroBasic provides library which simplifies using PWM HW Module.

Note: Certain PICmicros with two or more CCP modules, such as P18F8520, require you to specify the module you want to use. Simply append the number 1 or 2 to a Pwm. For example, `Pwm2_Start` Also, for the sake of backward compabitility with previous compiler versions and easier code management, MCU's with multiple PWM modules have PWM library which is identical to PWM1 (i.e. you can use `PWM_Init` instead of `PWM1_Init` to initialize CCP1).

Library Routines

```
Pwm_Init
Pwm_Change_Duty
Pwm_Start
Pwm_Stop
```

Pwm_Init

Prototype	`sub procedure Pwm_Init(dim freq as longint)`
Description	Initializes the PWM module with duty ratio 0. Parameter `freq` is a desired PWM frequency in Hz (refer to device data sheet for correct values in respect with Fosc). `Pwm_Init` needs to be called before using other functions from PWM Library.
Requires	You need a CCP module in order to use this library. Check mikroBasic installation folder, subfolder “Examples”, for alternate solutions.
Example	`Pwm_Init(5000)   ' Initialize PWM module at 5KHz`

Pwm_Change_Duty

Prototype	`sub procedure Pwm_Change_Duty(dim duty_ratio as byte)`
Description	Changes PWM duty ratio. Parameter `duty_ratio` takes values from 0 to 255, where 0 is 0%, 127 is 50%, and 255 is 100% duty ratio. Other specific values for duty ratio can be calculated as (Percent*255)/100.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initalized – see `Pwm_Init`.
Example	`Pwm_Change_Duty(192)   ' Set duty ratio to 75%`

Pwm_Start

Prototype	`sub procedure Pwm_Start`
Description	Starts PWM.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initalized – see `Pwm_Init`.
Example	`Pwm_Start`

Pwm_Stop

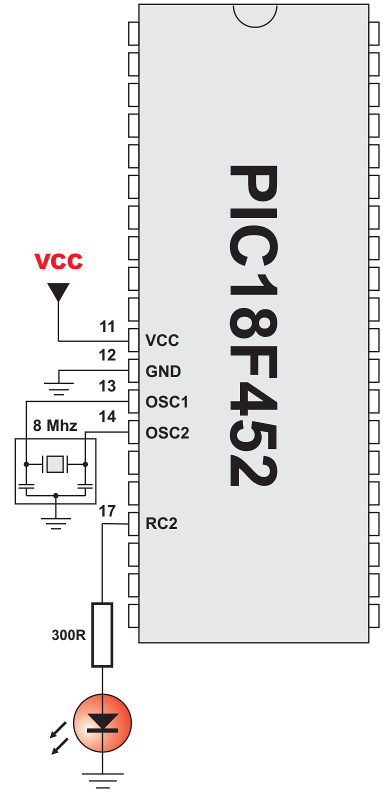
Prototype	`sub procedure Pwm_Stop`
Description	Stops PWM.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initalized – see `Pwm_Init`.
Example	`Pwm_Stop`

Library Example

The example changes PWM duty ratio on pin RC2 continually. If LED is connected to RC2, you can observe the gradual change of emitted light.

```
program Pwm_Test
dim j as byte
main:
  j = 0
  PORTC = $FF              ' Initialize PORTC
  Pwm_Init(5000)           ' Initialize PWM module, freq = 5kHz.
  Pwm_Start                ' Start PWM
  while true
    for i = 0 to 20
    Delay_us(500)
    Inc(j)
    Pwm_Change_Duty(j)     ' Change duty ratio
  wend
end.
```

Hardware Connection



RS-485 Library

RS-485 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 RS-485 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. Each Slave has its unique address and receives only the packets addressed to it. Slave can never initiate communication. It is programmer's responsibility to ensure that only one device transmits via 485 bus at a time.

RS-485 routines require USART module on PORTC. Pins of USART need to be attached to RS-485 interface transceiver, such as LTC485 or similar. Pins of transceiver (Receiver Output Enable and Driver Outputs Enable) should be connected to PORTC, pin 2 (check the figure at end of the chapter).

Note: Address 50 is the common address for all Slaves (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: Usart_Init must be called before initializing RS485.

Library Routines

```
RS485Master_Init
RS485Master_Receive
RS485Master_Send
RS485Slave_Init
RS485Slave_Receive
RS485Slave_Send
```

RS485Master_Init

Prototype	`sub procedure RS485Master_Init(dim byref port as byte, dim pin as byte)`
Description	Initializes PIC MCU as Master in RS-485 communication.
Requires	USART HW module needs to be initialized. See `USART_Init`.
Example	`RS485Master_Init(PORTC, 2)`

RS485Master_Receive

Prototype	`sub procedure RS485Master_Receive(dim byref data as byte[5])`
Description	Receives any message sent by Slaves. Messages are multi-byte, so 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 the message, `data[3]` is number of message bytes received, 1–3, `data[4]` is set to 255 when message is received, `data[5]` is set to 255 if error has occurred, `data[6]` is the address of the Slave which sent the message. Function automatically adjusts `data[4]` and `data[5]` upon every received message. These flags need to be cleared from the program.
Requires	MCU must be initialized as Master in RS-485 communication in order to be assigned an address. See `RS485Master_Init`.
Example	`RS485Master_Receive(msg)`

RS485Master_Send

Prototype	`sub procedure RS485Master_Send(dim byref data as byte[2], dim datalen, address as byte)`
Description	Sends `data` from buffer to Slave(s) specified by `address` via RS-485; `datalen` is a number of bytes in message (1 <= datalen <= 3).
Requires	MCU must be initialized as Master in RS-485 communication in order to be assigned an address. See `RS485Master_Init`. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.
Example	`RS485Master_Send(msg, 3, $12)`

RS485Slave_Init

Prototype	`sub procedure Rs485Slave_Init(dim byref port as byte, dim pin, address as byte)`
Description	Initializes MCU as Slave with a specified address in RS-485 communication. Slave `address` can take any value between 0 and 255, except 50, which is common address for all slaves.
Requires	USART HW module needs to be initialized. See `Usart_Init`.
Example	`RS485Slave_Init(PORTC, 2, 160)   ' Initialize MCU as Slave with address 160`

RS485Slave_Receive

Prototype	`sub procedure RS485Slave_Receive(dim byref data as byte[ 5] )`
Description	Receives message addressed to it. Messages are multi-byte, so 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 the message, `data[ 3]` is number of message bytes received, 1–3, `data[ 4]` is set to 255 when message is received, `data[ 5]` is set to 255 if error has occurred, `data[ 6]` is the address of the Slave which sent the message. Function automatically adjusts `data[ 4]` and `data[ 5]` upon every received message. These flags need to be cleared from the program.
Requires	MCU must be initialized as Slave in RS-485 communication in order to be assigned an address. See `RS485Slave_Init`.
Example	`RS485Slave_Read(msg)`

RS485Slave_Send

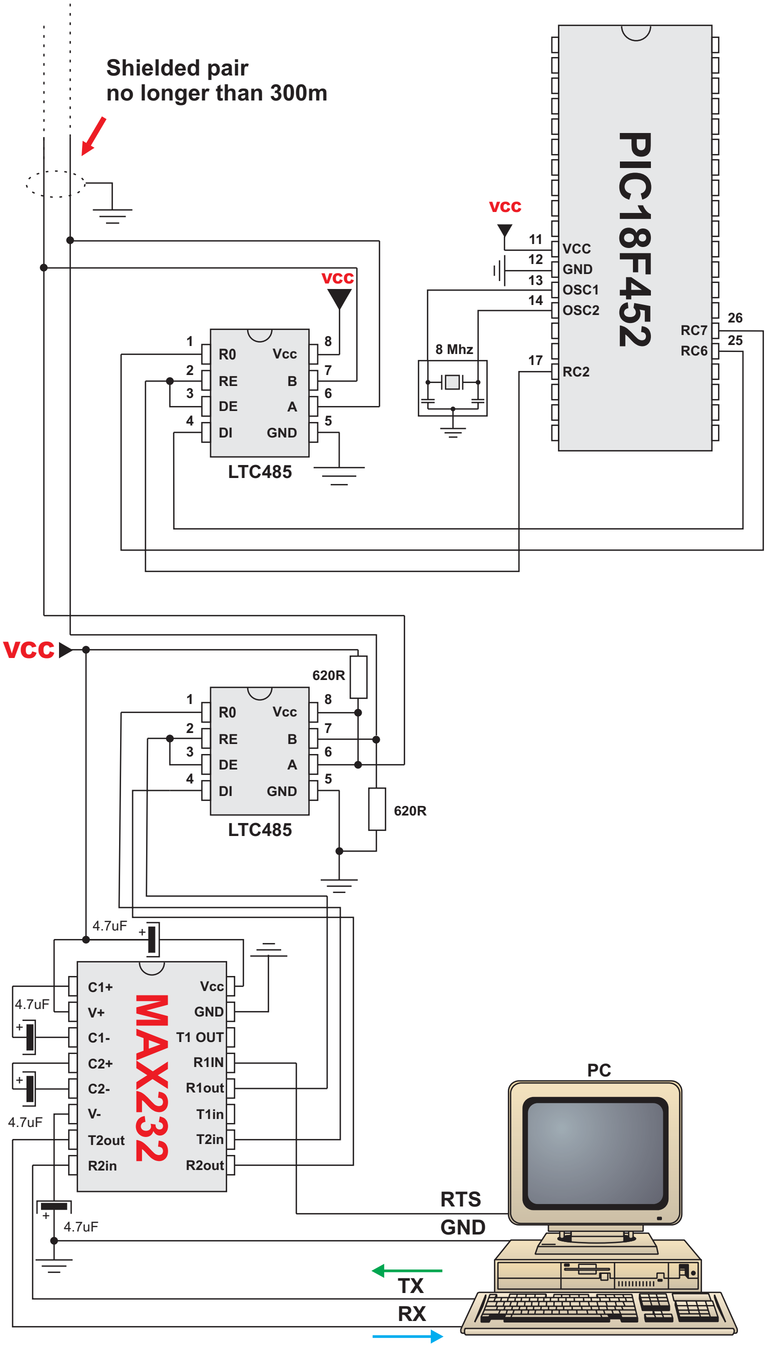
Prototype	`sub procedure RS485Slave_Write(dim byref data as byte[ 2] , dim datalen as byte)`
Description	Sends `data` from buffer to Master via RS-485; `datalen` is a number of bytes in message (1 <= datalen <= 3).
Requires	MCU must be initialized as Slave in RS-485 communication in order to be assigned an address. See `RS485Slave_Init`. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.
Example	`RS485Slave_Send(msg, 2)`

Library Example

The example demonstrates working with PIC as Slave nod in RS-485 communication. PIC receives only packets addressed to it (address 160 in our example), and general messsages with target address 50. The received data is forwarded to PORTB, and sent back to Master.

```
program rs485_test
dim i, j as byte
dim dat as byte[8]          ' Message buffer

sub procedure interrupt
  if TestBit(RCSTA, OERR) = 1 then
    PORTD = $81
  end if
  RS485Slave_Read(dat)
end sub

main:
  TRISB = 0
  TRISD = 0
  Usart_init(9600)              ' Initialize USART module
  RS485Slave_Init(PORTC, 2, 160) ' Initialize MCU as Slave, address 160
  SetBit(PIE1, RCIE)            ' Enable interrupt on byte received
  SetBit(INTCON, PEIE)          '   via USART (RS-485)
  ClearBit(PIE2, TXIE)
  SetBit(INTCON, GIE)
  PORTB  = 0
  PORTD  = 0
  dat[4] = 0                    ' Clear "msg received" flag
  dat[5] = 0                    ' Clear error flag

  while true
    if dat[5] = TRUE then       ' If there is error, set PORTD to $AA
      PORTD = $AA
    end if

    if dat[4] = TRUE 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_Write(dat, 1)  '   Send it back to Master
    end if
  wend
end.
```

Hardware Connection



Software I2C Library

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: This library implements time-based activities, so interrupts need to be disabled when using Soft I²C.

Library Routines

```
Soft_I2C_Config
Soft_I2C_Start
Soft_I2C_Read
Soft_I2C_Write
Soft_I2C_Stop
```

Soft_I2C_Config

Prototype	`sub procedure Soft_I2C_Config(dim byref port as byte, dim SDA, SCL as byte)`
Description	Configures software I²C. Parameter `port` specifies port of MCU on which `SDA` and `SCL` pins are located. Parameters `SCL` and `SDA` need to be in range 0–7 and cannot point at the same pin. `Soft_I2C_Config` needs to be called before using other functions from Soft I2C Library.
Example	`Soft_I2C_Config(PORTB, 1, 2)`

Soft_I2C_Start

Prototype	`sub procedure Soft_I2C_Start`
Description	Issues START signal. Needs to be called prior to sending and receiving data.
Requires	Soft I²C must be configured before using this function. See `Soft_I2C_Config`.
Example	`Soft_I2C_Start`

Soft_I2C_Read

Prototype	`sub function Soft_I2C_Read(dim ack as byte) as byte`
Returns	Returns one byte from the slave.
Description	Reads one byte from the slave, and sends not acknowledge signal if parameter ack is 0, otherwise it sends acknowledge.
Requires	START signal needs to be issued in order to use this function. See `Soft_I2C_Start`.
Example	`tmp = Soft_I2C_Read(0)   ' Read data, send not-acknowledge signal`

Soft_I2C_Write

Prototype	`sub function Soft_I2C_Write(dim data as byte) as byte`
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter `data`) via I²C bus.
Requires	START signal needs to be issued in order to use this function. See `Soft_I2C_Start`.
Example	`Soft_I2C_Write($A3)`

Soft_I2C_Stop

Prototype	`sub procedure Soft_I2C_Stop`
Description	Issues STOP signal.
Requires	START signal needs to be issued in order to use this function. See `Soft_I2C_Start`.
Example	`Soft_I2C_Stop`

Library Example

The example demonstrates use of Software I²C Library. 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. Check the hardware connection scheme at hardware I2C Library.

```
program soft_i2c_test

dim ee_adr, 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 = $FF                     ' Initialize PORTC
  Soft_I2C_Start()                ' Issue I2C signal: start
  Soft_I2C_Write($A2)             ' Send byte via I2C (command to 24cO2)
  ee_adr = 2
  Soft_I2C_Write(ee_adr)          ' Send byte (address for EEPROM)
  ee_data = $AA
  Soft_I2C_Write(ee_data)         ' Send data to be written
  Soft_I2C_Stop()                 ' Issue I2C signal: stop

  for jj = 0 to 65500             ' Pause while EEPROM writes data
    nop
  next jj

  Soft_I2C_Start()                ' Issue I2C start signal
  Soft_I2C_Write($A2)             ' Send byte via I2C
  ee_adr = 2
  Soft_I2C_Write(ee_adr)          ' Send byte (address for EEPROM)
  Soft_I2C_Start()                ' Issue I2C signal: repeated start
  Soft_I2C_Write($A3)             ' Send byte (request data from EEPROM)
  ee_data = Soft_I2C_Read(0)      ' Read the data
  Soft_I2C_Stop()                 ' Issue I2C signal: stop
  PORTC = ee_data                 ' Display data on PORTC

noend: goto noend
end.
```

Software SPI Library

mikroBasic provides library which implement software SPI. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

The library configures 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.

Note: These functions implement time-based activities, so interrupts need to be disabled when using the library.

Library Routines

```
Soft_Spi_Config
Soft_Spi_Read
Soft_Spi_Write
```

Soft_Spi_Config

Prototype	`sub procedure Soft_Spi_Config(dim byref port as byte, dim SDI, SDO, SCK as byte)`
Description	Configures and initializes software SPI. 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. `Soft_Spi_Config` needs to be called before using other functions from Soft SPI Library.
Example	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: `Soft_Spi_Config(PORTB, 1, 2, 3)`

Soft_Spi_Read

Prototype	`sub function Soft_Spi_Read(dim buffer as byte) as byte`
Returns	Returns the received data.
Description	Provides clock by sending `buffer` and receives data.
Requires	Soft SPI must be initialized and communication established before using this function. See `Soft_Spi_Config`.
Example	`tmp = Soft_Spi_Read(buffer)`

Soft_Spi_Write

Prototype	`sub procedure Soft_Spi_Write(dim data as byte)`
Description	Immediately transmits `data`.
Requires	Soft SPI must be initialized and communication established before using this function. See `Soft_Spi_Config`.
Example	`Soft_Spi_Write(1)`

Library Example

The example demonstrates using Software SPI library. Assumed HW configuration is: max7219 (chip select pin) is connected to RD1, and SDO, SDI, SCK pins are connected to corresponding pins of max7219. Hardware connection is given on page 186.

```
program soft_spi_test
include "m7219"

dim i as byte

main:
  ' Standard configuration
  Soft_Spi_Config(PORTD, 4, 5, 3)
  TRISC = TRISC and $FD
  max7219_init                  ' Initialize max7219
  SetBit(PORTD, 1)              ' Select max7219
  Soft_Spi_Write(1)             ' Send address (1) to max7219
  Soft_Spi_Write(7)             ' Send data (7) to max7219
  ClearBit(PORTD, 1)            ' Deselect max7219
end.
```

Software UART Library

mikroBasic provides library which implements 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 listed below.

Note: This library implements time-based activities, so interrupts need to be disabled when using Soft UART.

Library Routines

```
Soft_Uart_Init
Soft_Uart_Read
Soft_Uart_Write
```

Soft_Uart_Init

Prototype	**sub procedure** `Soft_Uart_Init(`**dim byref** `port` **as byte**, **dim** `rx, tx, baud_rate, inverted` **as byte**`)`
Description	Initalizes software UART. Parameter `port` specifies port of MCU on which RX and TX pins are located; parameters `rx` and `tx` need to be in range 0–7 and cannot point at the same pin; `baud_rate` is the desired baud rate. Maximum baud rate depends on PIC's clock and working conditions. Parameter `inverted`, if set to non-zero value, indicates inverted logic on output. `Soft_Uart_Init` needs to be called before using other functions from Soft UART Library.
Example	This will initialize software UART and establish the communication at 9600 bps: `Soft_Uart_Init(PORTB, 1, 2, 9600, 0)`

Soft_Uart_Read

Prototype	`sub function Soft_Uart_Read(dim byref error as byte) as byte`
Returns	Returns a received byte.
Description	Function receives a byte via software UART. Parameter `error` will be zero if the transfer was successful. This is a non-blocking function call, so you should test the `error` manually (check the example below).
Requires	Soft UART must be initialized and communication established before using this function. See `Soft_Uart_Init`.

Example

```
' Here's a loop which holds until data is received:
error = 1
do
  data = Soft_Uart_Read(error)
loop until error = 0
```

Soft_Uart_Write

Prototype	`sub procedure Soft_Uart_Write(dim data as byte)`
Description	Function transmits a byte (`data`) via UART.
Requires	Soft UART must be initialized and communication established before using this function. See `Soft_Uart_Init`. 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.
Example	`Soft_Uart_Write($0A)`

Library Example

The example demonstrates simple data exchange via software UART. When PIC MCU receives data, it immediately sends the same data back. If PIC is connected to the PC (see the figure below), you can test the example from mikroBasic terminal for RS232 communication, menu choice **Tools > Terminal**.

```
program soft_uart_test
dim received_byte, er as byte

main:
  Soft_Uart_Init(PORTB, 1, 2, 2400, 0)         ' Initialize soft UART
  er = 1
  while true
    do
      received_byte = Soft_Uart_Read(er)       ' Read received data
    loop until er = 0
    Soft_Uart_Write(received_byte)             ' Send data via UART
  wend
end.
```

Sound Library

mikroBasic provides a Sound Library which allows you to use sound signalization in your applications. You need a simple piezo speaker (or other hardware) on designated port.

Library Routines

```
Sound_Init
Sound_Play
```

Sound_Init

Prototype	`sub procedure Sound_Init(dim byref port as byte, dim pin as byte)`
Description	Prepares hardware for output at specified `port` and `pin`. Parameter `pin` needs to be within range 0–7.
Example	`Sound_Init(PORTB, 2)   ' Initialize sound at RB2`

Sound_Play

Prototype	`sub procedure Sound_Play(dim period_div_10 as byte, dim num_of_periods as word)`
Description	Plays the sound at the specified `port` and `pin` (see `Sound_Init`). Parameter `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`).
Requires	To hear the sound, you need a piezo speaker (or other hardware) on designated port. Also, you must call `Sound_Init` to prepare hardware for output.
Example	To play sound of 1KHz: T = 1/f = 1ms = 1000 cycles @ 4MHz. This gives us our first parameter: 1000/10 = 100. Play 150 periods like this: `Sound_Play(100, 150)`

Library Example

The example 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 sound_test
dim adcvalue as integer

main:
  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 at RB2

  while true                        ' Play in loop:
    adcvalue = ADC_Read(2)          '   Get lower byte from ADC
    Sound_Play(adcvalue, 200)       '   Play the sound
  wend
end.
```

SPI Library

SPI module is available with a number of PIC MCU models. mikroBasic provides a library for initializing Slave mode and comfortable work with Master mode. PIC 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).

Note: This library supports module on PORTB or PORTC, and will not work with modules on other ports. Examples for PICmicros with module on other ports can be found in your mikroBasic installation folder, subfolder “Examples”.

Library Routines

```
Spi_Init
Spi_Init_Advanced
Spi_Read
Spi_Write
```

Spi_Init

Prototype	**`sub procedure`** `Spi_Init`
Description	Configures and initializes SPI with default settings. `Spi_Init_Advanced` or `Spi_Init` needs to be called before using other functions from SPI Library. Default settings are: Master mode, clock Fosc/4, clock idle state low, data transmitted on low to high edge, and input data sampled at the middle of interval. For custom configuration, use `Spi_Init_Advanced`.
Requires	You need PIC MCU with hardware integrated SPI.
Example	`Spi_Init`

Spi_Init_Advanced

Prototype

```
sub procedure Spi_Init_Advanced(dim master, data_sample,
clock_idle, transmit_edge as byte)
```

Description

Configures and initializes SPI. `Spi_Init_Advanced` or `Spi_Init` needs to be called before using other functions of SPI Library.

Parameter `mast_slav` determines the work mode for SPI; can have the values:

```
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
```

The `data_sample` determines when data is sampled; can have the values:

```
DATA_SAMPLE_MIDDLE ' Input data sampled in middle of interval
DATA_SAMPLE_END    ' Input data sampled at the end of interval
```

Parameter `clock_idle` determines idle state for clock; can have the following values:

```
CLK_IDLE_HIGH      ' Clock idle HIGH
CLK_IDLE_LOW       ' Clock idle LOW
```

Parameter `transmit_edge` can have the following values:

```
LOW_2_HIGH         ' Data transmit on low to high edge
HIGH_2_LOW         ' Data transmit on high to low edge
```

Requires

You need PIC MCU with hardware integrated SPI.

Example

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:

```
Spi_Init_Advanced(MASTER_OSC_DIV4, DATA_SAMPLE_MIDDLE,
CLK_IDLE_LOW, LOW_2_HIGH)
```

Spi_Read

Prototype	`sub function Spi_Read(dim buffer as byte) as byte`
Returns	Returns the received data.
Description	Provides clock by sending `buffer` and receives data at the end of period.
Requires	SPI must be initialized and communication established before using this function. See `Spi_Init_Advanced` or `Spi_Init`.
Example	`take = Spi_Read(buffer)`

Spi_Write

Prototype	`sub procedure Spi_Write(dim data as byte) as byte`
Description	Writes byte `data` to SSPBUF, and immediately starts the transmission.
Requires	SPI must be initialized and communication established before using this function. See `Spi_Init_Advanced` or `Spi_Init`.
Example	`Spi_Write(1)`

Library Example

The code demonstrates how to use SPI library procedures and functions. Assumed configuration is: max7219 (chip select pin) connected to RC1, and SDO, SDI, SCK pins are connected to corresponding pins of max7219.

```
program spi_test
include "m7219"

main:
  Spi_Init                   ' Standard configuration
  TRISC = TRISC and $FD
  max7219_init               ' Initialize max7219
  ClearBit(PORTC, 1)         ' Select max7219
  Spi_Write(1)               ' Send address (1) to max7219
  Spi_Write(7)               ' Send data (7) to max7219
  SetBit(PORTC, 1)           ' Deselect max7219
end.
```

USART Library

USART hardware module is available with a number of PICmicros. mikroBasic USART Library provides 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 the topic – RS232 HW connection). You need a PIC MCU with hardware integrated USART, for example PIC16F877. Then, simply use the functions listed below.

Note: USART library functions support module on PORTB, PORTC, or PORTG, and will not work with modules on other ports. Examples for PICmicros with module on other ports can be found in “Examples” in mikroBasic installation folder.

Library Routines

```
Usart_Init
Usart_Data_Ready
Usart_Read
Usart_Write
```

Note: Certain PICmicros 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 a function name. For example, `Usart_Write2`.

Usart_Init

Prototype	`sub procedure Usart_Init(dim baud_rate as longint)`
Description	Initializes hardware USART module with the desired baud rate. 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. `Usart_Init` needs to be called before using other functions from USART Library.
Requires	You need PIC MCU with hardware USART.
Example	`Usart_Init(2400)   ' Establish communication at 2400 bps`

Usart_Data_Ready

Prototype	`sub function Usart_Data_Ready as byte`
Returns	Function returns 1 if data is ready or 0 if there is no data.
Description	Use the function to test if data in receive buffer is ready for reading.
Requires	USART HW module must be initialized and communication established before using this function. See `Usart_Init`.
Example	

```
' If data is ready, read it:
if Usart_Data_Ready = 1 then
  receive = Usart_Read
end if
```

Usart_Read

Prototype	`sub function Usart_Read as byte`
Returns	Returns the received byte. If byte is not received, returns 0.
Description	Function receives a byte via USART. Use the function `Usart_Data_Ready` to test if data is ready first.
Requires	USART HW module must be initialized and communication established before using this function. See `Usart_Init`.
Example	

```
' If data is ready, read it:
if Usart_Data_Ready = 1 then
  receive = Usart_Read
end if
```

Usart_Write

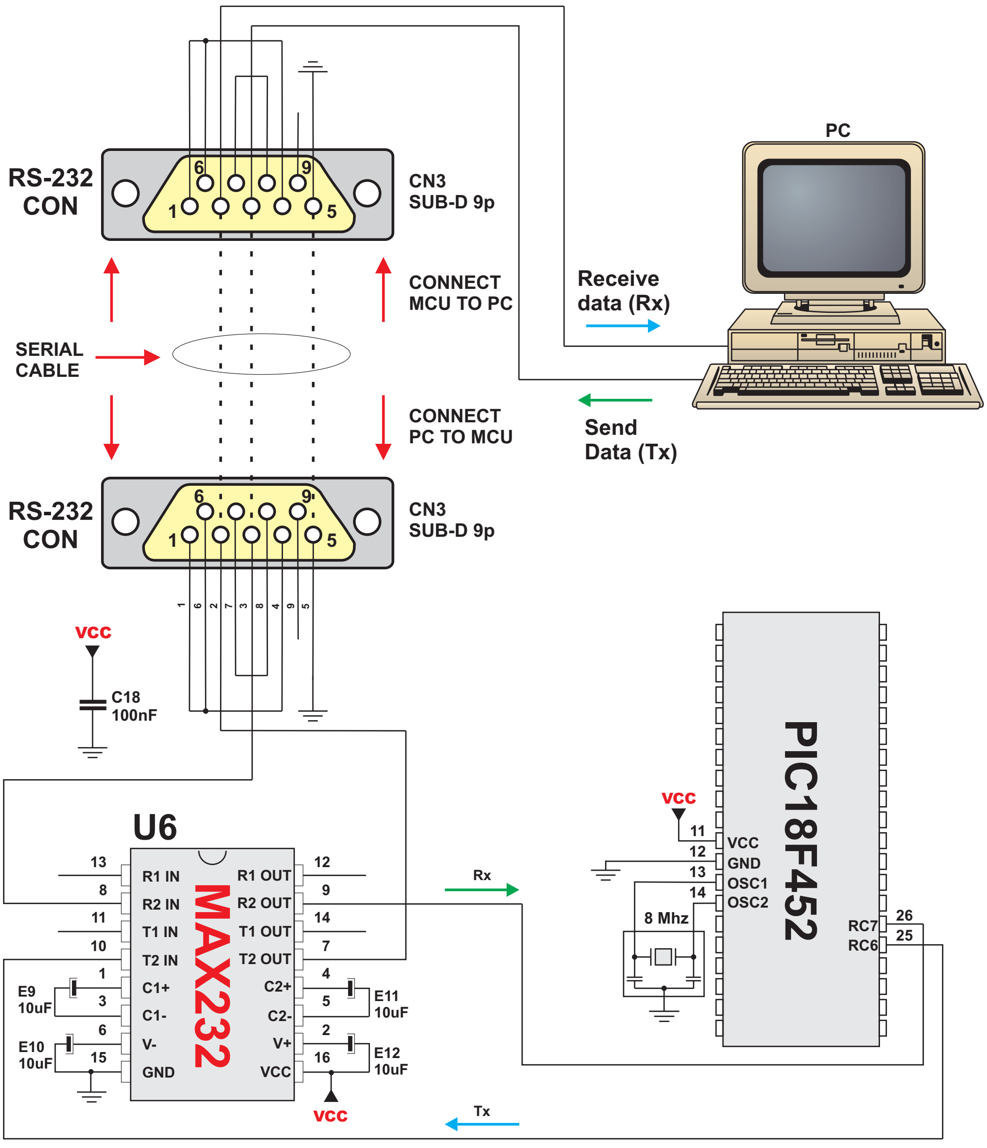
Prototype	`sub procedure Usart_Write(dim data as byte)`
Description	Function transmits a byte (data) via USART.
Requires	USART HW module must be initialized and communication established before using this function. See `Usart_Init`.
Example	`Usart_Write($1E)   ' send chunk via USART`

Library Example

The example demonstrates simple data exchange via USART. When PIC receives the data, it immediately sends it back. If PIC is connected to the PC (see the figure below), you can test the example from mikroBasic terminal for RS232 communication, menu choice Tools > Terminal.

```
program rs232_com_test
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.
```

Hardware Connection



USB HID Library

Universal Serial Bus (USB) provides a serial bus standard for connecting a wide variety of devices, including computers, cell phones, game consoles, PDAs, etc.

mikroBasic includes a library for working with human interface devices via Universal Serial Bus. A human interface device or HID is a type of computer device that interacts directly with and takes input from humans, such as the keyboard, mouse, graphics tablet, and the like.

Each project based on the USB HID library should include a descriptor source file which contains vendor id and name, product id and name, report length, and other relevant information. To create a descriptor file, use the integrated USB HID terminal of mikroBasic (Tools > USB HID Terminal). The default name for descriptor file is `USBdsc.pbas`, but you may rename it. The provided code in the "Examples" folder works at 48MHz, and the flags should not be modified without consulting the appropriate datasheet first.

Library Routines

```
Hid_Enable
Hid_Read
Hid_Write
Hid_Disable
```

Hid_Enable

Prototype	`sub procedure Hid_Enable(dim readbuff, writebuff as word)`
Description	Enables USB HID communication. Parameters readbuff and writebuff are the addresses of Read Buffer and the Write Buffer, respectively, which are used for HID communication. You can pass buffer names with the `@` operator. This function needs to be called before using other routines of USB HID Library.
Example	`Hid_Enable(@rd, @wr)`

Hid_Read

Prototype	`sub function Hid_Read as byte`
Returns	Number of characters in Read Buffer received from Host.
Description	Receives message from host and stores it in the Read Buffer. Function returns the number of characters received in Read Buffer.
Requires	USB HID needs to be enabled before using this function. See `Hid_Enable`.
Example	`length = Hid_Read`

Hid_Write

Prototype	`sub procedure Hid_Write(dim writebuff as word, dim len as byte`
Description	Function sends data from Write Buffer `writebuff` to host. Write Buffer is the address of the parameter used in initialization; see `Hid_Enable`. You can pass a buffer name with the `@` operator. Parameter `len` should specify a length of the data to be transmitted.
Requires	USB HID needs to be enabled before using this function. See `Hid_Enable`.
Example	`Hid_Write(@wr, len)`

Hid_Disable

Prototype	`sub procedure Hid_Disable`
Description	Disables USB HID communication.
Example	`Hid_Disable`

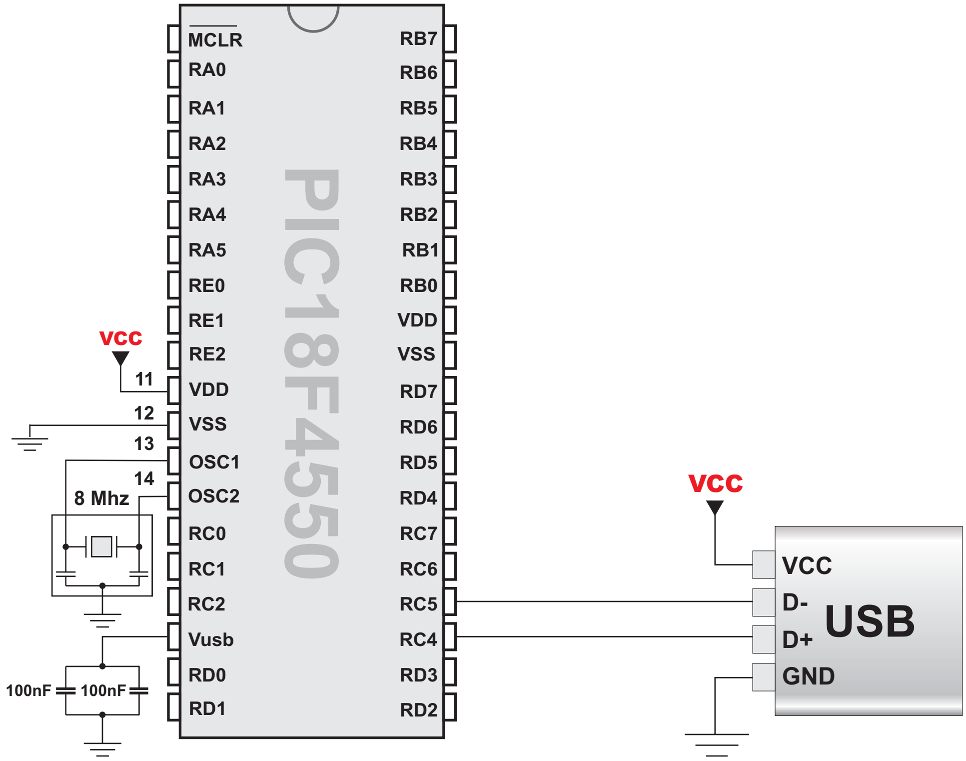
Library Example

The following example continually sends sequence of numbers 0..255 to the PC via Universal Serial Bus.

```
program hid_test

dim k as byte
dim userRD_buffer as byte[ 64]
dim userWR_buffer as byte[ 64]

sub procedure interrupt
  asm
    CALL _Hid_InterruptProc
    nop
  end asm
end sub

sub procedure Init_Main
  ' Disable all interrupts
  ' Disable GIE, PEIE, TMR0IE, INT0IE,RBIE
  INTCON  = 0
  INTCON2 = $F5
  INTCON3 = $C0
  ' Disable Priority Levels on interrupts
  RCON.IPEN = 0
  PIE1 = 0
  PIE2 = 0
  PIR1 = 0
  PIR2 = 0

  ' Configure all ports with analog function as digital
  ADCON1 = ADCON1 or $0F

  TRISA = 0
  TRISB = 0
  TRISC = $FF
  TRISD = $FF
  TRISE = $07

  LATA = 0
  LATB = 0
  LATC = 0
  LATD = 0
  LATE = 0

  ' continues ..
```

```
  ' .. continued

  ' Clear user RAM
  ' Banks [00 .. 07] ( 8 x 256 = 2048 Bytes )
  asm
    LFSR     FSR0, $000
    MOVLW    $08
    CLRF     POSTINC0, 0
    CPFSEQ   FSR0H, 0
    BRA      $ - 2
  end asm

  ' Timer 0
  T0CON = $07;
  TMR0H = (65536 - 156) >> 8
  TMR0L = (65536 - 156) and $FF
  INTCON.T0IE   = 1             ' Enable T0IE
  T0CON.TMR0ON = 1
end sub


'** Main Program **

main:
  Init_Main()
  Hid_Enable(@userRD_buffer, @userWR_buffer)

  do
    for k = 0 to 255
      ' Prepare send buffer
      userWR_buffer[ 0]  = k

      ' Send the number via USB
      Hid_Write(@userWR_buffer, 1)
    next k
  loop until FALSE

  Hid_Disable
end.
```

HW Connection



Util Library

Util library contains miscellaneous routines useful for project development.

Button

Prototype	`sub function Button(dim byref port as byte, dim pin, time, active_state as byte) as byte`
Returns	Returns 0 or 255.
Description	Function eliminates the influence of contact flickering upon pressing a button (debouncing). Parameter `port` specifies the location of the button; parameter `pin` is the pin number on designated `port` and goes from 0..7; parameter `time` is a debounce period in milliseconds; parameter `active_state` can be either 0 or 1, and it determines if the button is active upon logical zero or logical one.
Example	Example reads RB0, to which the button is connected; on transition from 1 to 0 (release of button), PORTD is inverted:

```
while true
  if Button(PORTB, 0, 1, 1) then
    oldstate = 255
  end if
  if oldstate and Button(PORTB, 0, 1, 0) then
    PORTD = not(PORTD)
    oldstate = 0
  end if
wend
```

Conversions Library

mikroBasic Conversions Library provides routines for converting numerals to strings, and routines for BCD/decimal conversions.

You can get text representation of numerical value by passing it to one of the following routines:

Library Routines

```
ByteToStr
ShortToStr
WordToStr
WordToStrWithZeros
IntToStr
LongintToStr
FloatToStr
```

Following functions convert decimal values to BCD (Binary Coded Decimal) and vice versa:

```
Bcd2Dec
Dec2Bcd
Bcd2Dec16
Dec2Bcd16
```

ByteToStr

Prototype	`sub procedure ByteToStr(dim number as byte, dim byref output as string[ 3] )`
Description	Procedure creates an `output` string out of a small unsigned `number` (numerical value less than $100). Output string has fixed width of 3 characters; remaining positions on the left (if any) are filled with blanks.

Example

```
dim t as word
dim txt as string[ 3]
'...
t = 24
ByteToStr(t, txt)   ' txt is " 24" (one blank here)
```

ShortToStr

Prototype	`sub procedure ShortToStr(dim number as short, dim byref output as string[4])`
Description	Procedure creates an `output` string out of a small signed `number` (numerical value less than $100). Output string has fixed width of 4 characters; remaining positions on the left (if any) are filled with blanks.

Example	`dim t as short`
	`dim txt as string[4];`
	`'...`
	`t = -24`
	`ShortToStr(t, txt)   ' txt is " -24" (one blank here)`

WordToStr

Prototype	`sub procedure WordToStr(dim number as word, dim byref output as string[5])`
Description	Procedure creates an `output` string out of an unsigned `number` (numerical value of `word` type). Output string has fixed width of 5 characters; remaining positions on the left (if any) are filled with blanks.

Example	`dim t as word`
	`dim txt as string[5]`
	`'...`
	`t = 437`
	`WordToStr(t, txt)   ' txt is "  437" (two blanks here)`

WordToStrWithZeros

Prototype	`sub procedure WordToStrWithZeros(dim number as word, dim byref output as string[5])`
Description	Procedure creates an output string out of an unsigned number (numerical value of word type). Output string has fixed width of 5 characters; remaining positions on the left (if any) are filled with zeros.
Requires	The output string should have the exact length as specified in the procedure prototype (5 characters).

Example	`dim t as word`
	`dim txt as string[5]`
	`'...`
	`t = 437`
	`WordToStr(t, txt)   ' txt is "  437" (two blanks here)`

IntToStr

Prototype	`sub procedure IntToStr(dim number as integer, dim byref output as string[6])`
Description	Procedure creates an `output` string out of a signed `number` (numerical value of `integer` type). Output string has fixed width of 6 characters; remaining positions on the left (if any) are filled with blanks.

Example	`dim j as integer`
	`dim txt as string[6]`
	`'...`
	`j = -4220`
	`IntToStr(j, txt)   ' txt is " -4220" (one blank here)`

LongintToStr

Prototype	`sub procedure LongintToStr(dim number as longint,` `dim byref output as string[11])`
Description	Procedure creates an `output` string out of a large signed `number` (numerical value of `longint` type). Output string has fixed width of 11 characters; remaining positions on the left (if any) are filled with blanks.

Example	`dim jj as longint`
	`dim txt as string[11]`
	`'...`
	`jj = -3700000`
	`LongintToStr(jj, txt)`
	`' txt is "   -3700000" (three blanks here)`

FloatToStr

Prototype	**sub procedure** `FloatToStr`(**dim** `input` **as float, dim byref** `output` **as string**[17])
Description	Procedure creates string out of the `input` parameter, which should be a floating point number in the `longint` range (±2147483648). Parameter `output` accepts the created string. The result is given in format "*`integer.fraction`*", left aligned. **Note:** Procedure won't return the correct result if `input` exceeds the `longint` range! You'll need to create a custom routine if you want to handle such large numbers. The *`integer`* part has flexible width of up to 11 characters (10 digits + sign). If the actual integer part is shorter than that, string will wrap to the integer's length. The *`fraction`* part is always 5 characters long. If the actual fraction is shorter than 5 digits, remaining chars on the right will be filled with zeroes; if the fraction exceeds 5 digits, the *`fraction`* part will be trimmed.
Requires	If you want to use the `FloatToStr` for printing on LCD, ensure that your program clears/refreshes the display with each printing of a string. Otherwise, LCD will display the remnants (rightmost digits) of the previous string, if it was longer than the presently displayed one.
Example	

```
// An example which prints value of a float variable on LCD:
dim input as float
dim output as string[17]
main:
input = -3.1415
FloatToStr(input, output)
Lcd_Out_Cp(output)   ' Print "-3.14150" on LCD
```

StrToInt

Prototype	`sub function StrToInt(dim byref input as string[6]) as integer`
Returns	Integer variable.
Description	Converts a string to integer.
Requires	The string is assumed to be a correct representation of a number.
Example	Here's an example which prints value of a `longint` variable on LCD:

```
dim ii as integer

main:
  ii = StrToInt('-1234')
```

StrToWord

Prototype	`sub function StrToWord(dim byref input as string[5]) as word`
Returns	Word variable.
Description	Converts a string to word.
Requires	`input` string with length of max 5 chars. The string is assumed to be a correct representation of a number.
Example	Here's an example which prints value of a `word` variable on LCD:

```
dim ww as word
main:
  ww = StrToword('65432')
```

Bcd2Dec

Prototype	`sub function Bcd2Dec(dim bcdnum as byte) as byte`
Returns	Returns converted decimal value.
Description	Converts 8-bit BCD numeral `bcdnum` to its decimal equivalent.
Example	

```
dim a, b as byte
a = $52
b = Bcd2Dec(a)   ' b equals 52
```

Dec2Bcd

Prototype	`sub function Dec2Bcd(dim decnum as byte) as byte`
Returns	Returns converted BCD value.
Description	Converts 8-bit decimal value `decnum` to BCD.

Example	`dim a, b as byte`
	`a = 52`
	`b = Dec2Bcd(a)   ' b equals $52`

Bcd2Dec16

Prototype	`sub function Bcd2Dec16(dim bcdnum as byte) as byte`
Returns	Returns converted decimal value.
Description	Converts 16-bit BCD numeral `bcdnum` to its decimal equivalent.

Example	`dim a, b as word`
	`a = 1234`
	`b = Bcd2Dec16(a)   ' b equals 4660`

Dec2Bcd16

Prototype	`sub function Dec2Bcd16(dim decnum as byte) as byte`
Returns	Returns converted BCD value.
Description	Converts 16-bit decimal value `decnum` to BCD.

Example	`dim a, b as word`
	`a = 4660`
	`b = Dec2Bcd16(a)   ' b equals 1234`

Delays Library

mikroBasic provides a basic utility routines for creating software delay. You can create more advanced and flexible versions based on this library.

Note: Routines do not provide an entirely accurate delay as it depends on clock specified in Project settings.

Delay_us

Prototype	`sub procedure Delay_us(const time_in_us as word)`
Description	Creates a software delay in duration of `time_in_us` microseconds (a constant). Range of applicable constants depends on the oscillator frequency.
Example	`Delay_us(10)   ' Ten microseconds pause`

Delay_ms

Prototype	`sub procedure Delay_ms(const time_in_ms as word)`
Description	Creates a software delay in duration of `time_in_ms` milliseconds (a constant). Range of applicable constants depends on the oscillator frequency.
Example	`Delay_ms(1000)   ' One second pause`

Vdelay_ms

Prototype	`sub procedure Vdelay_ms(dim time_in_ms as word)`
Description	Creates a software delay in duration of `time_in_ms` milliseconds (a variable). Generated delay is not as precise as the delay created by `Delay_ms`.

Example	`pause = 1000`
	`' ...`
	`Vdelay_ms(pause)   ' ~ one second pause`

Delay_Cyc

Prototype	`sub procedure Delay_Cyc(dim cycles_div_by_10 as byte)`
Description	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. Note that `Delay_Cyc` is library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Example	`Delay_Cyc(10)   ' Hundred MCU cycles pause`

Math Library

Math Library implements a number of common mathematical functions.

Library Routines

`Acos`
`Asin`
`Atan`
`Atan2`
`Ceil`
`Cos`
`CosE3`
`Cosh`
`Exp`
`Fabs`
`Floor`
`Frexp`
`Fmod`
`Ldexp`
`Log`
`Log10`
`Modf`
`Pow`
`Sin`
`SinE3`
`Sinh`
`Sqrt`
`Tan`
`Tanh`

Acos

Prototype	`sub function Acos(dim x as float) as float`
Description	Function returns the arc cosine of parameter `x`; that is, the value whose cosine is `x`. Input parameter `x` must be between -1 and 1 (inclusive). The return value is in radians, between 0 and pi (inclusive).

Asin

Prototype	`sub function Asin(dim x as float) as float`
Description	Function returns the arc sine of parameter `x`; that is, the value whose sine is `x`. Input parameter `x` must be between -1 and 1 (inclusive). The return value is in radians, between -pi/2 and pi/2 (inclusive).

Atan

Prototype	`sub function Atan(dim x as float) as float`
Description	Function computes the arc tangent of parameter `x`; that is, the value whose tangent is `x`. The return value is in radians, between -pi/2 and pi/2 (inclusive).

Atan2

Prototype	`sub function Atan2(dim x, y as float) as float`
Description	This is the two argument arc tangent function. It is similar to computing the arc tangent of `y/x`, except that the signs of both arguments are used to determine the quadrant of the result, and `x` is permitted to be zero. The return value is in radians, between -pi and pi (inclusive).

Ceil

Prototype	`sub function Ceil(dim x as float) as float`
Description	Function returns value of parameter `x` rounded up to the next whole number.

Cos

Prototype	`sub function Cos(dim x as float) as float`
Description	Function returns the cosine of `x` in radians. The return value is from -1 to 1.

CosE3

Prototype	`sub function CosE3(dim x as word) as integer`
Description	Function takes parameter `x` which represents angle in degrees, and returns its cosine multiplied by 1000 and rounded up to the nearest integer: `result := round_up(cos(x)*1000)` The function is implemented as a lookup table; maximum error obtained is ±1.

Cosh

Prototype	`sub function Cosh(dim x as float) as float`
Description	Function returns the hyperbolic cosine of `x`, defined mathematically as $(e^x+e^{-x})/2$. If the value of `x` is too large (if overflow occurs), the function fails.

Exp

Prototype	`sub function Exp(dim x as float) as float`
Description	Function returns the value of e — the base of natural logarithms — raised to the power of `x` (i.e. `e`^{x}).

Fabs

Prototype	`sub function Fabs(dim x as float) as float`
Description	Function returns the absolute (i.e. positive) value of `x`.

Floor

Prototype	`sub function Floor(dim x as float) as float`
Description	Function returns value of parameter `x` rounded down to the nearest integer.

Fmod

Prototype	`sub function Fmod(dim x, y as float) as float`
Description	Function computes the floating point remainder of `x/y`. Function returns the value `x - i * y` for some integer `i` such that, if `y` is nonzero, the result has the same sign as `x` and magnitude less then the magnitude of `y`. If v is zero, the `fmod` function returns zero.

Frexp

Prototype	`sub function Frexp(dim num as float, dim n as ^integer) as float`
Description	Function splits a floating-point value `num` into a normalized fraction and an integral power of 2. Return value is the normalized fraction, and the integer exponent is stored in the object pointed to by `n`.

Ldexp

Prototype	`sub function Ldexp(dim num as float, dim n as integer) as float`
Description	Function returns the result of multiplying the floating-point number `num` by 2 raised to the power `n` (i.e. returns $x * 2^n$).

Log

Prototype	`sub function Log(dim x as float) as float`
Description	Function returns the natural logarithm of `x` (i.e. $\log_e(x)$).

Log10

Prototype	`sub function Log10(dim x as float) as float`
Description	Function returns the base-10 logarithm of `x` (i.e. $\log_{10}(x)$).

Modf

Prototype	**sub function** `Modf(`**dim** `num` **as float, dim** `whole` **as ^float**`)` **as float**
Description	Function returns the signed fractional component of `num`, placing its whole number component into the variable pointed to by `whole`.

Pow

Prototype	**sub function** `Pow(`**dim** `x, y` **as float**`)` **as float**
Description	Function returns the value of `x` raised to the power of y (i.e. x^y). If the `x` is negative, function will automatically cast the `y` into `longint`.

Sin

Prototype	**sub function** `Sin(`**dim** `x` **as float**`)` **as float**
Description	Function returns the sine of `x` in radians. The return value is from -1 to 1.

SinE3

Prototype	**sub function** `SinE3(`**dim** `x` **as word**`)` **as integer**
Description	Function takes parameter `x` which represents angle in degrees, and returns its sine multiplied by 1000 and rounded up to the nearest integer: *`result := round_up(sin(x)*1000)`* The function is implemented as a lookup table; maximum error obtained is ±1.

Sinh

Prototype	`sub function Sinh(dim x as float) as float`
Description	Function returns the hyperbolic sine of `x`, defined mathematically as $(e^x-e^{-x})/2$. If the value of `x` is too large (if overflow occurs), the function fails.

Sqrt

Prototype	`sub function Sqrt(dim x as float) as float`
Description	Function returns the non negative square root of `num`.

Tan

Prototype	`sub function Tan(dim x as float) as float`
Description	Function returns the tangent of `x` in radians. The return value spans the allowed range of floating point in mikroPascal.

Tanh

Prototype	`sub function Tanh(dim x as float) as float`
Description	Function returns the hyperbolic tangent of `x`, defined mathematically as `sinh(x)/cosh(x)`.

String Library

The String Library provides a number of routines for string handling.

Library Routines

```
Memchr
Memcmp
Memcpy
Memmove
Memset
Strcat
Strchr
Strcmp
Strcpy
Strcspn
Strlen
Strncat
Strncmp
Strncpy
Strpbrk
Strrchr
Strspn
Strstr
strAppendSuf
strAppendPre
```

Memchr

Prototype	`sub function Memchr(dim p as word, dim ch as char, dim n as word) as word`
Description	Function locates the first occurrence of byte `ch` in the initial `n` bytes of memory area starting at the address `p`. Function returns the offset of this occurrence from the memory address p or `$FFFF` if the character was not found. For parameter `p` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Memcmp

Prototype	`sub function Memcmp(dim p1, p2, n as word) as integer`
Description	Function returns a positive, negative, or zero value indicating the relationship of first `n` bytes of memory areas starting at addresses `p1` and `p2`. The `Memcmp` function compares two memory areas starting at addresses `p1` and `p2` for `n` bytes and returns a value indicating their relationship as follows: **Value** **Meaning** < 0 `p1` "less than" `p2` = 0 `p1` "equal to" `p2` > 0 `p1` "greater than" `p2` The value returned by function is determined by the difference between the values of the first pair of bytes that differ in the strings being compared. For parameters `p1` and `p2` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Memcpy

Prototype	**`sub procedure`** `Memcpy(`**`dim`** `p1, p2, n` **`as word`**`)`
Description	Function copies n bytes from the memory area starting at the address `p2` to the memory area starting at `p1`. If these memory buffers overlap, the memcpy function cannot guarantee that bytes are copied before being overwritten. If these buffers do overlap, use the `Memmove` function. For parameters `p1` and `p2` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Memmove

Prototype	**`sub procedure`** `Memmove(`**`dim`** `p1, p2, n` **`as word`**`)`
Description	Function copies n bytes from the memory area starting at the address `p2` to the memory area starting at `p1`. If these memory buffers overlap, the Memmove function ensures that bytes in `p2` are copied to `p1` before being overwritten. For parameters `p1` and `p2` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Memset

Prototype	**`sub procedure`** `Memset(`**`dim`** `p` **`as word`**`,` **`dim`** `ch` **`as char`**`,` **`dim`** `n` **`as word`**`)`
Description	Function fills the first `n` bytes in the memory area starting at the address `p` with the value of byte `ch`. For parameter `p` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring` or `@PORTB`.

Strcat

Prototype	`sub procedure Strcat(dim byref s1, s2 as string[100])`
Description	Function appends the value of string `s2` to string `s1` and terminates s1 with a null character.

Strchr

Prototype	`sub function Strchr(dim byref s as string[100], dim ch as char) as byte`
Description	Function searches the string `s` for the first occurrence of the character `ch`. The null character terminating `s` is not included in the search. Function returns the position (index) of the first character `ch` found in `s`; if no matching character was found, function returns `$FF`.

Strcmp

Prototype	`sub function Strcmp(dim byref s1, s2 as string[100]) as byte`
Description	Function lexicographically compares the contents of strings `s1` and `s2` and returns a value indicating their relationship: **Value Meaning** < 0 `s1` "less than" `s2` = 0 `s1` "equal to" `s2` > 0 `s1` "greater than" `s2` The value returned by function is determined by the difference between the values of the first pair of bytes that differ in the strings being compared.

Strcpy

Prototype	`sub procedure Strcpy(dim byref s1, s2 as string[100])`
Description	Function copies the value of string `s2` to the string `s1` and appends a null character to the end of `s1`.

Strcspn

Prototype	`sub function Strcspn(dim byref s1, s2 as string[100]) as byte`
Description	The strcspn function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of characters not from the string pointed to by s2.Function returns the length of the segment.

Strlen

Prototype	`sub function Strlen(dim byref s as string[100]) as byte`
Description	Function returns the length, in bytes, of the string s. The length does not include the null terminating character.

Strncat

Prototype	`sub procedure Strncat(dim byref s1, s2 as string[100], dim n as byte)`
Description	Function appends at most n characters from the string `s2` to the string `s1` and terminates `s1` with a null character. If `s2` is shorter than `n` characters, `s2` is copied up to and including the null terminating character.

Strncmp

Prototype	`sub function Strncmp(dim byref s1, s2 as string[100], dim n as byte) as integer`
Description	Function lexicographically compares the first `n` bytes of the strings `s1` and `s2` and returns a value indicating their relationship: **Value** **Meaning** < 0 `s1` "less than" `s2` = 0 `s1` "equal to" `s2` > 0 `s1` "greater than" `s2` The value returned by function is determined by the difference between the values of the first pair of bytes that differ in the strings being compared (within first `n` bytes).

Strncpy

Prototype	`sub procedure Strncpy(dim byref s1, s2 as string[100], dim n as byte)`
Description	Function copies at most n characters from the string `s2` to the string `s1`. If `s2` contains fewer characters than n, `s1` is padded out with null characters up to the total length of n characters.

Strpbrk

Prototype	**`sub procedure`** `Strpbrk(`**`dim byref`** `s1, s2` **`as string`**`[100])`
Description	Function searches `s1` for the first occurrence of any character from the string `s2`. The null terminator is not included in the search. Function returns an index of the matching character in `s1`. If `s1` contains no characters from `s2`, function returns `$FF`.

Strrchr

Prototype	**`sub procedure`** `Strrchr(`**`dim byref`** `s` **`as string`**`[100],` **`dim`** `ch` **`as byte`**`)`
Description	Function searches the string `s` for the last occurrence of the character `ch`. The null character terminating `s` is not included in the search. Function returns an index of the last `ch` found in `s`; if no matching character was found, function returns `$FF`.

Strspn

Prototype	**`sub function`** `Strspn(`**`dim byref`** `s1, s2` **`as string`**`[100])` **`as byte`**
Description	The strspn function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of the characters from the string pointed to by s2.Function returns the length of the segment.

Strstr

Prototype	`sub function Strstr(dim byref s1, s2 as string[100]) as byte`
Description	Function locates the first occurrence of the string `s2` in the string `s1` (excluding the terminating null character). Function returns a number indicating the position of the first occurrence of `s2` in `s1`; if no string was found, function returns `$FF`. If `s2` is a null string, the function returns 0.

strAppendSuf

Prototype	`sub procedure strAppendSuf(dim byref s1 as string[100], dim letter as char)`
Description	Adds suffix(letter) to string (s1).
Example	txt = "123" strAppendSuf(txt, "4"); *' txt = "1234"*

strAppendPre

Prototype	`sub procedure strAppendPre(dim letter as char, dim byref s1 as string[100])`
Description	Adds prefix(letter) to string (s1).
Example	txt = "123" strAppendPre("0", txt) *' txt = "0123"*

SPI Graphic LCD Library

mikroBasic provides a library for operating the Graphic LCD 128x64 via SPI. These routines work with the common GLCD 128x64 (samsung ks0108).

Note: Be sure to designate port with GLCD as output, before using any of the following library procedures or functions.

Note: `Spi_Init` must be called before initializing SPI GLCD.

Library Routines

Basic routines:

```
Spi_Glcd_Init
Spi_Glcd_Set_Side
Spi_Glcd_Set_Page
Spi_Glcd_Set_X
Spi_Glcd_Read_Data
Spi_Glcd_Write_Data
```

Advanced routines:

```
Spi_Glcd_Fill
Spi_Glcd_Dot
Spi_Glcd_Line
Spi_Glcd_V_Line
Spi_Glcd_H_Line
Spi_Glcd_Rectangle
Spi_Glcd_Box
Spi_Glcd_Circle
Spi_Glcd_Set_Font
Spi_Glcd_Write_Char
Spi_Glcd_Write_Text
Spi_Glcd_Image
```

Spi_Glcd_Init

Prototype	**sub procedure** `Spi_Glcd_Init(`**dim byref** `RstPort` **as byte,** **dim** `RstPin` **as byte,** **dim byref** `CSPort` **as byte,** **dim** `CSPin, DeviceAddress` **as byte**`)`
Description	Initializes Graphic LCD 128x64 via SPI. RstPort and RstPin - Sets pin connected on reset pin of spi expander. `CSPort` and `CSPin` - Sets pin connected on CS pin of spi expander. `device address` - address of spi expander (hardware setting of A0, A1 and A2 pins (connected on VCC or GND) on spi expander).
Requires	`Spi_Init` must be called before initializing SPI GLCD. This procedure needs to be called before using other routines of SPI GLCD library.
Example	`Spi_Glcd_Init(PORTC, 0, PORTC, 1, 0)`

Spi_Glcd_Set_Side

Prototype	**sub procedure** `Spi_Glcd_Set_Side(`**dim** `x` **as byte**`)`
Description	Selects side of GLCD, left or right. Parameter `x` specifies the side: values from 0 to 63 specify the left side, and values higher than 64 specify the right side. Use the functions `Spi_Glcd_Set_Side`, `Spi_Glcd_Set_X`, and `Spi_Glcd_Set_Page` to specify an exact position on GLCD. Then, you can use `Spi_Glcd_Write_Data` or `Spi_Glcd_Read_Data` on that location.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Select_Side(0)` `Spi_Glcd_Select_Side(10)`

Spi_Glcd_Set_Page

Prototype	`sub procedure Spi_Glcd_Set_Page(dim page as byte)`
Description	Selects page of GLCD, technically a line on display; parameter `page` can be 0..7.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Set_Page(5)`

Spi_Glcd_Set_X

Prototype	`sub procedure Spi_Glcd_Set_X(dim x as byte)`
Description	Positions to x dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Set_X(25)`

Spi_Glcd_Read_Data

Prototype	`sub function Spi_Glcd_Read_Data as byte`
Returns	One word from the GLCD memory.
Description	Reads data from from the current location of GLCD memory. Use the functions `Spi_Glcd_Set_Side`, `Spi_Glcd_Set_X`, and `Spi_Glcd_Set_Page` to specify an exact position on GLCD. Then, you can use `Spi_Glcd_Write_Data` or `Spi_Glcd_Read_Data` on that location.
Requires	Reads data from from the current location of GLCD memory.
Example	`tmp = Spi_Glcd_Read_Data`

Spi_Glcd_Write_Data

Prototype	`sub procedure Spi_Glcd_Write_Data(dim data as byte)`
Description	Writes `data` to the current location in GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Write_Data(data)`

Spi_Glcd_Fill

Prototype	`sub procedure Spi_Glcd_Fill(dim pattern as byte)`
Description	Fills the GLCD memory with byte pattern. To clear the GLCD screen, use `Spi_Glcd_Fill(0)`; to fill the screen completely, use `Spi_Glcd_Fill($FF)`.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Fill(0)   ' Clear screen`

Spi_Glcd_Dot

Prototype	`sub procedure Spi_Glcd_Dot(dim x, y, color as byte)`
Description	Draws a dot on the GLCD at coordinates `(x, y)`. Parameter color determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Dot(0, 0, 2)`

Spi_Glcd_Line

Prototype	`sub procedure Spi_Glcd_Line(dim x1, y1, x2, y2, color as byte)`
Description	Draws a line on the GLCD from `(x1, y1)` to `(x2, y2)`. Parameter `color` determines the dot state: 0 draws an empty line (clear dots), 1 draws a full line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Line(0, 63, 50, 0, 2)`

Spi_Glcd_V_Line

Prototype	`sub procedure Spi_Glcd_V_Line(dim y1, y2, x, color as byte)`
Description	Similar to GLcd_Line, draws a vertical line on the GLCD from `(x, y1)` to `(x, y2)`.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_V_Line(0, 63, 0, 1)`

Spi_Glcd_H_Line

Prototype	`sub procedure Spi_Glcd_H_Line(dim x1, x2, y, color as byte)`
Description	Similar to GLcd_Line, draws a horizontal line on the GLCD from `(x1, y)` to `(x2, y)`.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_H_Line(0, 127, 0, 1)`

Spi_Glcd_Rectangle

Prototype	`sub procedure Spi_Glcd_Rectangle(dim x1, y1, x2, y2, color as byte)`
Description	Draws a rectangle on the GLCD. Parameters `(x1, y1)` set the upper left corner, `(x2, y2)` set the bottom right corner. Parameter `color` defines the border: 0 draws an empty border (clear dots), 1 draws a solid border (put dots), and 2 draws a “smart” border (invert each dot).
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Rectangle(10, 0, 30, 35, 1)`

Spi_Glcd_Box

Prototype	`sub procedure Spi_Glcd_Box(dim x1, y1, x2, y2, color as byte)`
Description	Draws a box on the GLCD. Parameters `(x1, y1)` set the upper left corner, `(x2, y2)` set the bottom right corner. Parameter `color` defines the fill: 0 draws a white box (clear dots), 1 draws a full box (put dots), and 2 draws an inverted box (invert each dot).
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Box(10, 0, 30, 35, 1)`

Spi_Glcd_Circle

Prototype	`sub procedure Spi_Glcd_Circle(dim x, y, radius, color as integer)`
Description	Draws a circle on the GLCD, centered at (x, y) with radius. Parameter color defines the circle line: 0 draws an empty line (clear dots), 1 draws a solid line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Circle(63, 31, 25, 1)`

Spi_Glcd_Set_Font

Prototype	`sub procedure Spi_Glcd_Set_Font(dim font_address as longint, dim font_width, font_height as byte, dim offset as word)`
Description	Sets the font for text display routines, `Spi_Glcd_Write_Char` and `Spi_Glcd_Write_Text`. Font needs to be formatted as an array of `byte`. Parameter `font_address` specifies the address of the font; you can pass a font name with the `@` operator. Parameters `font_width` and `font_height` specify the width and height of characters in dots. Font width should not exceed 128 dots, and font height should not exceed 8 dots. Parameter `font_offset` determines the ASCII character from which the supplied font starts. Demo fonts supplied with the library have an offset of 32, which means that they start with space. If no font is specified, `Spi_Glcd_Write_Char` and `Spi_Glcd_Write_Text` will use the default 5x8 font supplied with the library. You can create your own fonts by following the guidelines given in the file "GLCD_Fonts.dpas". This file contains the default fonts for GLCD, and is located in your installation folder, "Extra Examples" > "GLCD".
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`' Use the custom 5x7 font "myfont" which starts with space (32):` `Spi_Glcd_Set_Font(@myfont, 5, 7, 32)`

Spi_Glcd_Write_Char

Prototype	`sub procedure Spi_Glcd_Write_Char(dim character, x, page, color as byte)`
Description	Prints character at page (one of 8 GLCD lines, 0..7), `x` dots away from the left border of display. Parameter color defines the "fill": 0 writes a "white" letter (clear dots), 1 writes a solid letter (put dots), and 2 writes a "smart" letter (invert each dot). Use routine `Spi_Glcd_Set_Font` to specify font, or the default 5x7 font (included with the library) will be used.
Requires	GLCD needs to be initialized, see `Spi_Glcd_Init`. Use the `Spi_Glcd_Set_Font` to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	`Spi_Glcd_Write_Char('C', 0, 0, 1)`

Spi_Glcd_Write_Text

Prototype	`sub procedure Spi_Glcd_Write_Text(dim text as string[20], dim x, page, color as byte)`
Description	Prints `text` at page (one of 8 GLCD lines, 0..7), `x` dots away from the left border of display. Parameter `color` defines the "fill": 0 prints a "white" letters (clear dots), 1 prints solid letters (put dots), and 2 prints "smart" letters (invert each dot). Use routine `Spi_Glcd_Set_Font` to specify font, or the default 5x7 font (included with the library) will be used.
Requires	GLCD needs to be initialized, see `Spi_Glcd_Init`. Use the `Spi_Glcd_Set_Font` to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	`Spi_Glcd_Write_Text('Hello world!', 0, 0, 1)`

Spi_Glcd_Image

Prototype	`sub procedure Spi_Glcd_Image(dim image as byte[1024])`
Description	Displays bitmap image on the GLCD. Parameter image should be formatted as an array of 1024 bytes. Use the mikroPascal's integrated Bitmap-to-LCD editor (menu option Tools > Graphic LCD Editor) to convert image to a constant array suitable for display on GLCD.
Requires	GLCD needs to be initialized. See `Spi_Glcd_Init`.
Example	`Spi_Glcd_Image(my_image)`

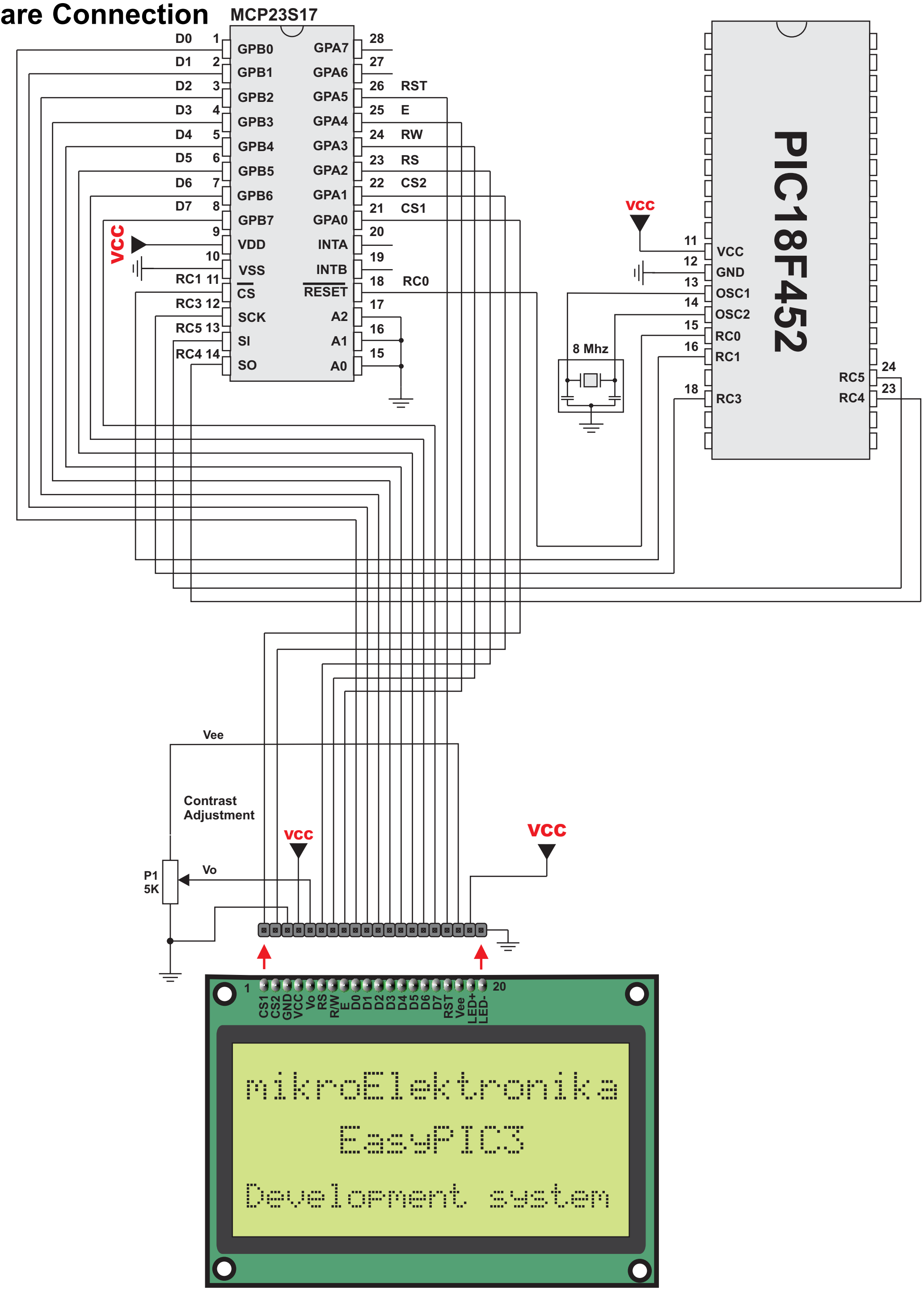
Library Example

The example demonstrates how to communicate to KS0108 GLCD via SPI module, using serial to parallel convertor MCP23S17.

```
program SerialGLCD

include "__Lib_SerialGlcd"
include "images"

dim
  ii as byte
  someText as string[20]

sub procedure delay2S
  delay_ms(2000)
end sub

main:
  delay2S
  Spi_Init ' initialize SPI
  Spi_Glcd_Init(PORTC, 2, PORTC, 1, 0)
  Spi_Glcd_Fill(0xAA)
  delay2S
  while TRUE
      Spi_Glcd_Fill(0x00)
      Spi_Glcd_Image(truck_bmp)
      delay2S

      Spi_Glcd_Fill(0x00)
      for ii  =  1 to 40
        Spi_Glcd_Dot(ii, ii, 1)
      next ii
      delay2S

      Spi_Glcd_Fill(0x00)
      Spi_Glcd_Line(120, 1, 5,60, 1)
      delay2S
      Spi_Glcd_Line(12, 42, 5,60, 1)
      delay2S

      Spi_Glcd_Rectangle(12, 20, 93,57, 1)
      delay2S


      //continues..
```

```
          Spi_Glcd_Line(120, 12, 12,60, 1)
          delay2S

          Spi_Glcd_H_Line(5, 15, 6, 1)
          Spi_Glcd_Line(0, 12, 120, 60, 1)
          Spi_Glcd_V_Line(7, 63, 127, 1)
          delay2S

          for ii  =  1 to 10
            Spi_Glcd_Circle(63, 32, 3*ii, 1)
          next ii

          delay2S
          Spi_Glcd_Box(12, 20, 70, 57, 2)
          delay2S

          Spi_Glcd_Fill(0x00)

          Spi_Glcd_Set_Font(@System3x6, 3, 6, 32)
          someText  =  "SMALL FONT: 3X6"
          Spi_Glcd_Write_Text(someText, 20, 5, 1)

          Spi_Glcd_Set_Font(@FontSystem5x8, 5, 8, 32)
          someText  =  "Large Font 5x8"
          Spi_Glcd_Write_Text(someText, 3, 4, 1)
          delay2S
        wend
    end.
```

Hardware Connection



Port Expander Library

The SPI Expander Library facilitates working with MCP23S17, Microchip's SPI port expander. The chip connects to the PIC according to the scheme presented below.

Note: PIC need to have a hardware SPI module.

Note: `SPI_Init` must be called before initializing Port Expander.

Library Routines

```
Expander_Init
PortExpanderSelect
PortExpanderUnSelect
Expander_Read_Byte
Expander_Write_Byte
Expander_Set_Mode
Expander_Read_Array
Expander_Write_Array
Expander_Read_PortA
Expander_Read_PortB
Expander_Read_ArrayPortA
Expander_Read_ArrayPortB
Expander_Write_PortA
Expander_Write_PortB
Expander_Set_DirectionPortA
Expander_Set_DirectionPortB
Expander_Set_PullUpsPortA
Expander_Set_PullUpsPortB
```

Expander_Init

Prototype	`sub procedure Expander_Init(dim byref RstPort as byte, dim RstPin as byte, dim byref CSPort as byte, dim CSPin, ModuleAddress as byte)`
Description	Establishes SPI communication with the expander and initializes the expander. `RstPort` and `RstPin` - Sets pin connected on reset pin of spi expander. `CSPort` and `CSPin` - Sets pin connected on CS pin of spi expander. `moduleaddress` - address of spi expander (hardware setting of A0, A1 and A2 pins (connected on VCC or GND) on spi expander).
Requires	`SPI_Init` must be called before initializing Port Expander. This procedure needs to be called before using other routines of PORT Expander library.
Example	`Expander_Init(PORTC, 0, PORTC, 1, 0)`

PortExpanderSelect

Prototype	`sub procedure PortExpanderSelect`
Description	Selects current port expander.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`PortExpanderSelect`

PortExpanderUnSelect

Prototype	`sub procedure PortExpanderUnSelect`
Description	Un-Selects current port expader.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`PortExpanderUnSelect`

Expander_Read_Byte

Prototype	`sub function Expander_Read_Byte(dim ModuleAddress, RegAddress as byte) as byte`
Returns	Byte read from port expander.
Description	Function reads byte from port expander on `ModuleAddress` and port on `RegAddress`.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Read_Byte(0,1)`

Expander_Write_Byte

Prototype	`sub procedure Expander_Write_Byte(dim ModuleAddress, RegAddress, Data as byte)`
Returns	Nothing.
Description	This routine writes data to port expander on `ModuleAddress` and port on `RegAddress`.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Write_Byte(0,1,$FF)`

Expander_Set_Mode

Prototype	`sub procedure Expander_Set_Mode(dim ModuleAddress, Mode as byte)`
Returns	Nothing.
Description	Sets port expander `mode` on `ModuleAddress`.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Set_Mode(1,0)`

Expander_Read_ArrayPortA

Prototype	`sub procedure Expander_Read_ArrayPortA(dim ModuleAddress, NoBytes as byte, dim byref DestArray as byte[ 100] )`
Returns	Nothing.
Description	This routine reads array of bytes (`DestArray`) from port expander on `ModuleAddress` and portA. `NoBytes` represents number of read bytes.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Read_PortA(0,1,data)`

Expander_Read_Array

Prototype	`sub procedure Expander_Read_Array(dim ModuleAddress, StartAddress, NoBytes as byte, dim byref DestArray as byte[ 100] )`
Returns	Nothing.
Description	This routine reads array of bytes (`DestArray`) from port expander on `ModuleAddress` and `StartAddress`. `NoBytes` represents number of read bytes.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Read_Array(1,1,5,data)`

Expander_Write_Array

Prototype	`sub procedure Expander_Write_Array(dim ModuleAddress, StartAddress, NoBytes as byte, dim byref SourceArray as byte[100])`
Returns	Nothing.
Description	This routine writes array of bytes (`DestArray`) to port expander on `ModuleAddress` and `StartAddress`. `NoBytes` represents number of read bytes.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Write_Array(1,1,5,data)`

Expander_Read_PortA

Prototype	`sub function Expander_Read_PortA(dim Address as byte) as byte`
Returns	Read byte.
Description	This routine reads byte from port expander on `Address` and PortA.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Read_PortA(1)`

Expander_Read_Array

Prototype	`sub procedure Expander_Read_Array(dim ModuleAddress, StartAddress, NoBytes as byte, dim byref DestArray as byte[100])`
Returns	Read byte.
Description	This routine reads byte from port expander on `Address` and PortB.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Read_Array(1,1,5,data)`

Expander_Read_ArrayPortB

Prototype	`sub procedure Expander_Read_ArrayPortB(dim ModuleAddress, NoBytes as byte, dim byref DestArray as byte[100])`
Returns	Nothing.
Description	This routine reads array of bytes (`DestArray`) from port expander on `ModuleAddress` and portB. `NoBytes` represents number of read bytes.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Read_PortB(0,8,data)`

Expander_Write_PortA

Prototype	`sub procedure Expander_Write_PortA(dim ModuleAddress, Data as byte)`
Returns	Nothing.
Description	This routine writes byte to port expander on `ModuleAddress` and portA.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_write_PortA(3,$FF)`

Expander_Write_PortB

Prototype	`sub procedure Expander_Write_PortB(dim ModuleAddress, Data as byte)`
Returns	Nothing.
Description	This routine writes byte to port expander on `ModuleAddress` and portB.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_write_PortB(2,$FF)`

Expander_Set_DirectionPortA

Prototype	**`sub procedure`** `Expander_Set_DirectionPortA(`**`dim`** `ModuleAddress, Data` **`as byte`**`)`
Description	Set port expander PortA pin as input or output.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Set_DirectionPortA(0,$FF)`

Expander_Set_DirectionPortB

Prototype	**`sub procedure`** `Expander_Set_DirectionPortB(`**`dim`** `ModuleAddress, Data` **`as byte`**`)`
Description	Set port expander PortB pin as input or output.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Set_DirectionPortB(0,$FF)`

Expander_Set_PullUpsPortA

Prototype	**`sub procedure`** `Expander_Set_PullUpsPortA(`**`dim`** `ModuleAddress, Data` **`as byte`**`)`
Description	This routine sets port expander PortA pin as pullup or pulldown.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Set_PullUpsPortA(0,$FF)`

Expander_Set_PullUpsPortB

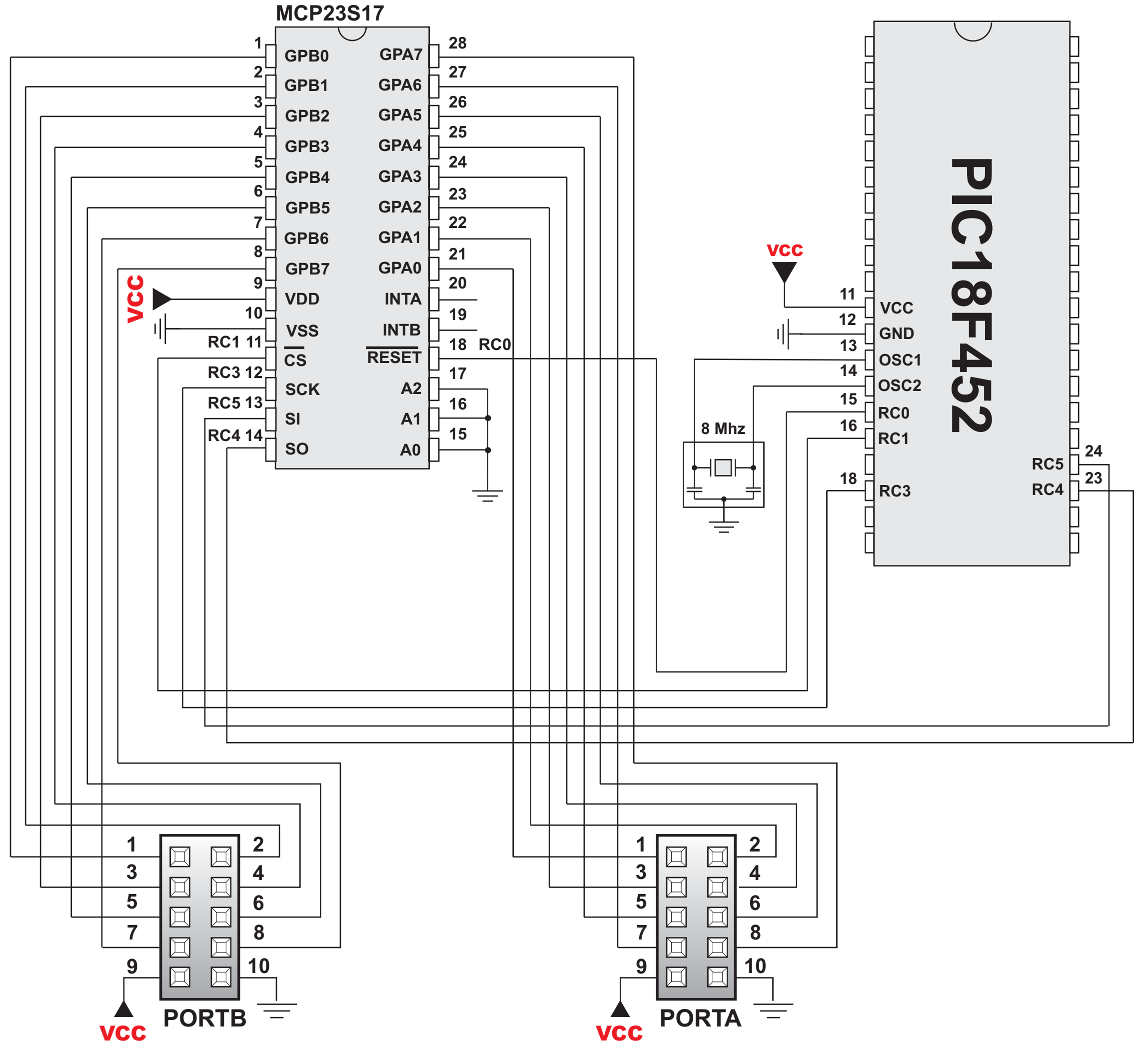
Prototype	**`sub procedure`** `Expander_Set_PullUpsPortB(`**`dim`** `ModuleAddress, Data` **`as byte`**`)`
Description	This routine sets port expander PortB pin as pullup or pulldown.
Requires	PORT Expander must be initialized. See `Expander_Init`.
Example	`Expander_Set_PullUpsPortB(0,$FF)`

Library Example

The example demonstrates how to communicate to port expander MCP23S17.

```
program PortExpander

dim i as byte

main:
  ADCON1 = ADCON1 or 0x0f
  TRISB  = 0x00
  PORTB  = 0xFF
  Delay_ms(200)

  Spi_Init ' initialize SPI

  Expander_Init(PORTC, 0, PORTC, 1, 0)
' initialize port expander

  Expander_Set_DirectionPortA(0, 0)
' set expander's porta to be output

  Expander_Set_DirectionPortB(0,0xFF)
' set expander's porta to be input

  Expander_Set_PullUpsPortB(0,0xFF)
' set pull ups to all of the expander's portb pins

  i = 0
  while 1
      Expander_Write_PortA(0, i)
      ' write i to expander's porta
      i = i + 1
      PORTB = Expander_Read_PortB(0)
      ' read expander's portb and write it to PIC's PORTB
      Delay_ms(20)
  wend
end.
```

Hardware Connection



SPI LCD Library (4-bit interface)

mikroBasic provides a library for communicating with commonly used LCD (4-bit interface) via SPI interface. Figures showing HW connection of PIC and SPI LCD are given at the end of the chapter.

Note: Spi_Init must be called before initializing SPI LCD.

Library Routines

```
Spi_Lcd_Config
Spi_Lcd_Init
Spi_Lcd_Out
Spi_Lcd_Out_Cp
Spi_Lcd_Chr
Spi_Lcd_Chr_Cp
Spi_Lcd_Cmd
```

Spi_Lcd_Config

Prototype	`sub procedure Spi_Lcd_Config(dim DeviceAddress as byte, dim byref rstport as byte, dim rstpin as byte, dim byref csport as byte, dim cspin as byte)`
Description	Initializes LCD via SPI interface with pin settings (Reset pin and Chip Select pin) you specify.
Requires	`Spi_Init` must be called before initializing SPI LCD.
Example	`Spi_Lcd_Config(0,PORTB, 1, PORTB, 0)`

Spi_Lcd_Init

Prototype	`sub procedure Spi_Lcd_Init`
Description	Initializes LCD at port with default pin settings (see the connection scheme at the end of the chapter).
Requires	`Spi_Init` must be called before initializing SPI LCD.
Example	`Spi_Lcd_Init`

Spi_Lcd_Out

Prototype	`sub procedure Spi_Lcd_Out(dim row, column as byte, dim byref text as string[20])`
Description	Prints text on LCD at specified row and column (parameters row and col). Both string variables and literals can be passed as text.
Requires	Port with LCD must be initialized. See `Spi_Lcd_Config` or `Spi_Lcd_Init`.
Example	`Spi_Lcd_Out(1, 3, "Hello!")`

Spi_Lcd_Out_Cp

Prototype	`sub procedure Spi_Lcd_Out_CP(dim byref text as string[40])`
Description	Prints `text` on LCD at current cursor position. Both string variables and literals can be passed as `text`.
Requires	Port with LCD must be initialized. See `Spi_Lcd_Config` or `Spi_Lcd_Init`.
Example	`Spi_Lcd_Out_Cp("Here!") ' Print "Here!" at current cursor position`

Spi_Lcd_Chr

Prototype	`sub procedure Spi_Lcd_Chr(dim Row, Column, Out_Char as byte)`
Description	Prints `character` on LCD at specified row and column (parameters `row` and `col`). Both variables and literals can be passed as `character`.
Requires	Port with LCD must be initialized. See `Spi_Lcd_Config` or `Spi_Lcd_Init`.
Example	`Spi_Lcd_Chr(2, 3, "i")`

Spi_Lcd_Chr_Cp

Prototype	`sub procedure Spi_Lcd_Chr_CP(dim Out_Char as byte)`
Description	Prints `character` on LCD at current cursor position. Both variables and literals can be passed as `character`.
Requires	Port with LCD must be initialized. See `Spi_Lcd_Config` or `Spi_Lcd_Init`.
Example	`Spi_Lcd_Chr_Cp("e")   ' Print "e" at current cursor position`

Spi_Lcd_Cmd

Prototype	`sub procedure Spi_Lcd_Cmd(dim out_char as byte)`
Description	Sends `command` to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown below.
Requires	Port with LCD must be initialized. See `Spi_Lcd_Config` or `Spi_Lcd_Init`.
Example	`Spi_Lcd_Cmd(Spi_Lcd_Clear)   ' Clear LCD display`

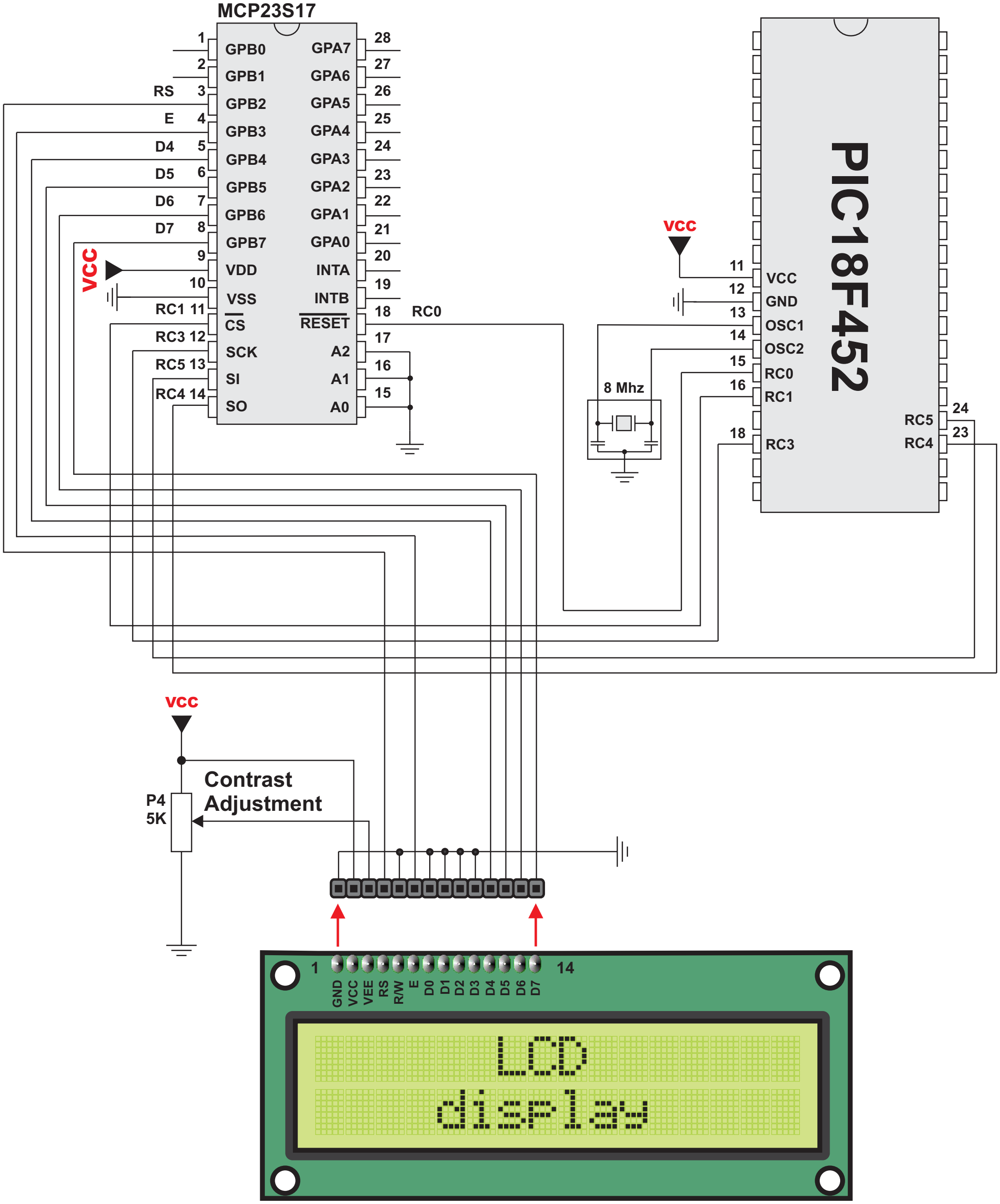
LCD Commands

LCD Command	Purpose
`LCD_FIRST_ROW`	Move cursor to 1st row
`LCD_SECOND_ROW`	Move cursor to 2nd row
`LCD_THIRD_ROW`	Move cursor to 3rd row
`LCD_FOURTH_ROW`	Move cursor to 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to original position. Display data RAM is unaffected.
`LCD_CURSOR_OFF`	Turn 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

Library Example (default pin settings)

```
program Spi_Lcd_default_test

dim text as char[ 20]

main:
   SPI_init                                 ' Initialize SPI communication
   Spi_Lcd_Init                             ' Initialize lcd via SPI interface
   Spi_Lcd_Cmd(LCD_CLEAR)                   ' Clear display
   Spi_Lcd_Cmd(LCD_CURSOR_OFF)              ' Turn cursor OFF
   Spi_Lcd_Out(1, 5, "mikroE")              ' Write Txt to LCD
   Spi_Lcd_Out(2, 1, "mikroElektronika")  ' Write Txt to LCD
   Spi_Lcd_Out(3, 1, "mikroE")  ' Write Txt to LCD, for lcd's with more than two rows
   Spi_Lcd_Out(4, 10, "mikroE")' Write Txt to LCD, for lcd's with more than two rows
end.
```

Hardware Connection



SPI LCD8 (8-bit interface) Library

mikroBasic provides a library for communicating with commonly used 8-bit interface LCD (with Hitachi HD44780 controller) via SPI Interface. Figures showing HW connection of PIC and SPI LCD are given at the end of the chapter.

Note: Spi_Init must be called before initializing SPI LCD8.

Library Routines

```
Spi_Lcd8_Config
Spi_Lcd8_Init
Spi_Lcd8_Out
Spi_Lcd8_Out_Cp
Spi_Lcd8_Chr
Spi_Lcd8_Chr_Cp
Spi_Lcd8_Cmd
```

Spi_Lcd8_Config

Prototype	`sub procedure Spi_LCD8_Config(dim DeviceAddress as byte, dim byref rstport as byte, dim rstpin as byte, dim byref csport as byte, dim cspin as byte)`
Description	Initializes LCD via SPI interface with pin settings (Reset pin and Chip Select pin) you specify.
Requires	`Spi_Init` must be called before initializing SPI LCD8.
Example	`Spi_Lcd8_Config(0, PORTB, 1, PORTB, 0)`

Spi_Lcd8_Init

Prototype	`sub procedure Spi_Lcd8_Init`
Description	Initializes LCD at Control port (ctrlport) and Data port (dataport) with default pin settings (see the connection scheme at the end of the chapter).
Requires	`Spi_Init` must be called before initializing SPI LCD8.
Example	`Spi_Lcd8_Init`

Spi_Lcd8_Out

Prototype	`sub procedure Spi_LCD8_Out(dim row, column as byte, dim byref Text as string[20])`
Description	Prints text on LCD at specified row and column (parameters row and col). Both string variables and literals can be passed as text.
Requires	Ports with LCD must be initialized. See `Spi_Lcd8_Config` or `Spi_Lcd8_Init`.
Example	`Spi_Lcd8_Out(1, 3, "Hello!")  ' Print "Hello!" at line 1, char 3`

Spi_Lcd8_Out_Cp

Prototype	`sub procedure Spi_LCD8_Out_CP(dim byref text as string[20])`
Description	Prints `text` on LCD at current cursor position. Both string variables and literals can be passed as `text`.
Requires	Ports with LCD must be initialized. See `Spi_Lcd8_Config` or `Spi_Lcd8_Init`.
Example	`Spi_Lcd8_Out_Cp("Here!")  ' Print "Here!" at current cursor position`

Spi_Lcd8_Chr

Prototype	`sub procedure Spi_LCD8_Chr(dim row, column, out_char as byte)`
Description	Prints `character` on LCD at specified row and column (parameters `row` and `col`). Both variables and literals can be passed as `character`.
Requires	Ports with LCD must be initialized. See `Spi_Lcd8_Config` or `Spi_Lcd8_Init`.
Example	`Spi_LCD8_Chr(1,1,"e")   ' Print "e" at line 1, char 1`

Spi_Lcd8_Chr_Cp

Prototype	`sub procedure Spi_LCD8_Chr_CP(dim out_char as byte)`
Description	Prints `character` on LCD at current cursor position. Both variables and literals can be passed as `character`.
Requires	Ports with LCD must be initialized. See `Spi_Lcd8_Config` or `Spi_Lcd8_Init`.
Example	`Spi_Lcd8_Chr_Cp("e");   ' Print "e" at current cursor position`

Spi_Lcd8_Cmd

Prototype	`sub procedure Spi_LCD8_Cmd(dim out_char as byte)`
Description	Sends `command` to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown below.
Requires	Ports with LCD must be initialized. See `Spi_Lcd8_Config` or `Spi_Lcd8_Init`.
Example	`Spi_Lcd8_Cmd(LCD_Clear)   ' Clear LCD display`

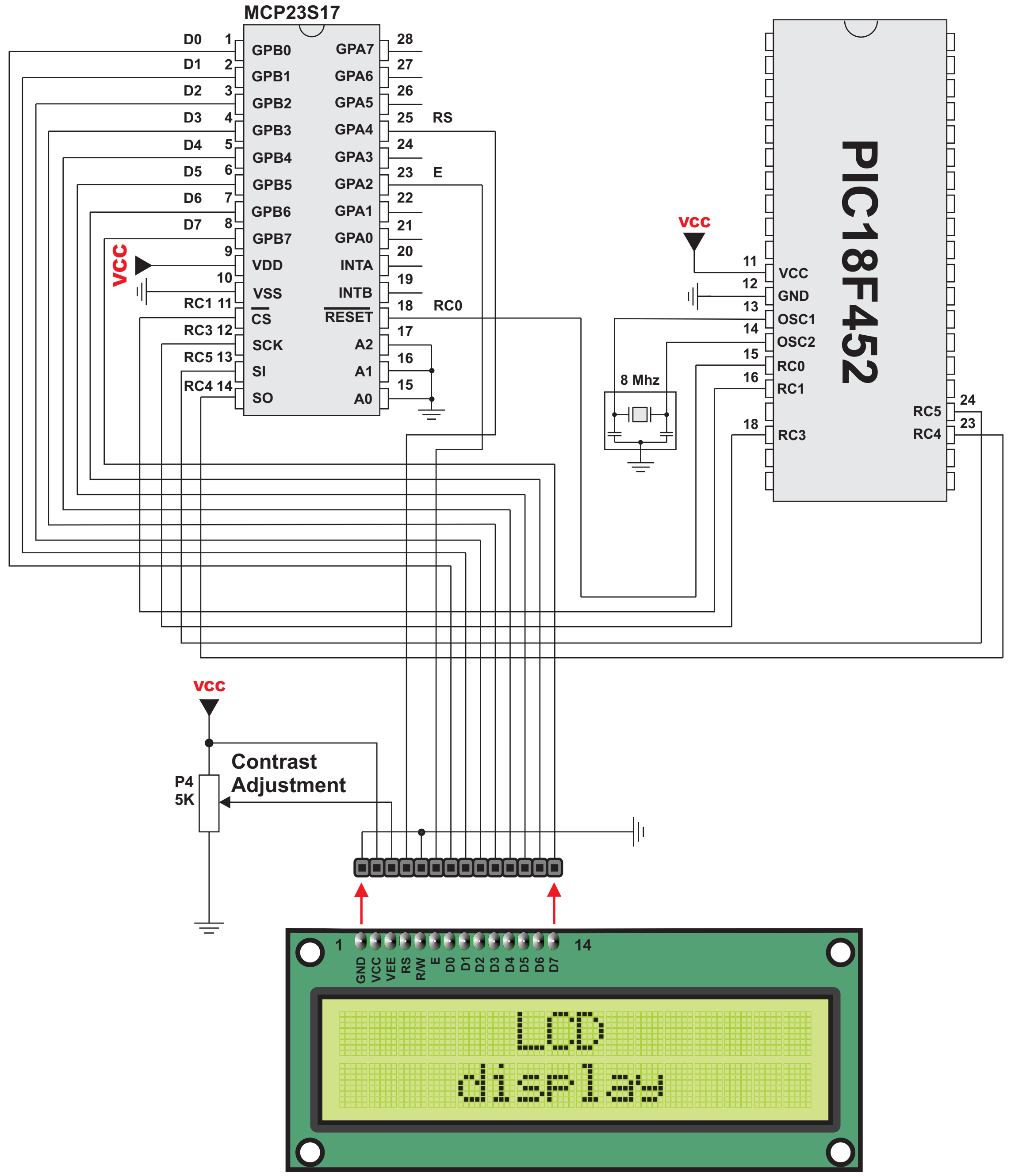
LCD Commands

LCD Command	Purpose
`LCD_FIRST_ROW`	Move cursor to 1st row
`LCD_SECOND_ROW`	Move cursor to 2nd row
`LCD_THIRD_ROW`	Move cursor to 3rd row
`LCD_FOURTH_ROW`	Move cursor to 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to original position. Display data RAM is unaffected.
`LCD_CURSOR_OFF`	Turn 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

Library Example (default pin settings)

```
program Spi_Lcd8_default_test
dim text as char[ 20]
main:
  SPI_init                                  ' Initialize SPI communication
  Spi_Lcd8_Init                            ' Initialize lcd via SPI interface
  Spi_Lcd8_Cmd(LCD_CLEAR)                 ' Clear display
  Spi_Lcd8_Cmd(LCD_CURSOR_OFF)           ' Turn cursor OFF
  Spi_Lcd8_Out(1, 5, "mikroE")           ' Write Txt to LCD
  Spi_Lcd8_Out(2, 1, "mikroElektronika")  ' Write Txt to LCD
  Spi_Lcd8_Out(3, 1, "mikroE")' Write Txt to LCD, for lcd's with more than two rows
  Spi_Lcd8_Out(4, 10, "mikroE")' Write Txt to LCD, for lcd's with more than two rows
end.
```

Hardware Connection



SPI T6963C Graphic LCD Library

mikroPascal for PIC provides a library for drawing and writing on Toshiba T6963C Graphic LCD (various sizes) via SPI interface.

Note: Spi_Init; must be called before initializing SPI LCD.

Library Routines

```
Spi_T6963C_Config
Spi_T6963C_writeData
Spi_T6963C_writeCommand
Spi_T6963C_setPtr
Spi_T6963C_waitReady
Spi_T6963C_fill
Spi_T6963C_dot
Spi_T6963C_write_char
Spi_T6963C_write_text
Spi_T6963C_line
Spi_T6963C_rectangle
Spi_T6963C_box
Spi_T6963C_circle
Spi_T6963C_image
Spi_T6963C_sprite
Spi_T6963C_set_cursor
Spi_T6963C_clearBit
Spi_T6963C_setBit
Spi_T6963C_negBit
Spi_T6963C_displayGrPanel
Spi_T6963C_displayTxtPanel
Spi_T6963C_setGrPanel
Spi_T6963C_setTxtPanel
Spi_T6963C_panelFill
Spi_T6963C_grFill
Spi_T6963C_txtFill
Spi_T6963C_cursor_height
Spi_T6963C_graphics
Spi_T6963C_text
Spi_T6963C_cursor
Spi_T6963C_cursor_blink
Spi_T6963C_Config_240x128
Spi_T6963C_Config_240x64
```

Spi_T6963C_Config

Prototype	`sub procedure Spi_T6963C_Config(dim width, height, fntW as word, dim byref rstport as byte, dim rstpin as byte, dim byref csport as byte, dim cspin as byte, dim wr, rd, cd, rst as byte, dim DeviceAddress as byte)`
Description	Initalizes the Graphic Lcd controller. This function must be called before all Spi T6963C Library Routines. `width` - Number of horizontal (x) pixels in the display. `height` - Number of vertical (y) pixels in the display. `fntW` - Font width, number of pixels in a text character, must be set accordingly to the hardware. `data` - Address of the port on which the Data Bus is connected. `cntrl` - Address of the port on which the Control Bus is connected. `wr` - !WR line bit number in the *cntrl port. `rd` - !RD line bit number in the *cntrl port. `cd` - !CD line bit number in the *cntrl port. `rst` - !RST line bit number in the *cntrl port. `DeviceAddress` - Device Address. Display RAM : The library doesn't know the amount of available RAM. The library cuts the RAM into panels : a complete panel is one graphics panel followed by a text panel, The programer has to know his hardware to know how much panel he has.
Requires	`Spi_Init` must be called before initializing SPI Toshiba T6963C Graphic LCD.
Example	see below

Example

```
Spi_T6963C_Config(240, 64, 8, PORTB, 1, PORTB, 0, 0, 1, 3, 4, 0)

'* init display for 240 pixel width and 64 pixel height
'* 8 bits character width
'* reset pin on PORTB.1
'* chip select pin on PORTB.0
'* bit 0 is !WR
'* bit 1 is !RD
'* bit 3 is !CD
'* bit 4 is RST
'* chip enable, reverse on, 8x8 font internaly set in library
'* device address is 0
```

Spi_T6963C_writeData

Prototype	**sub procedure** `Spi_T6963C_writeData(`**dim** `data` **as byte**`)`
Description	Routine that writes data to Spi T6963C controller.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	`Spi_T6963C_writeData(AddrL)`

Spi_T6963C_writeCommand

Prototype	**sub procedure** `Spi_T6963C_writeCommand(`**dim** `data` **as byte**`)`
Description	Routine that writes command to Spi T6963C controller
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	`Spi_T6963C_writeCommand(T6963C_CURSOR_POINTER_SET)`

Spi_T6963C_setPtr

Prototype	**sub procedure** `Spi_T6963C_setPtr(`**dim** `p` **as word, dim** `c` **as byte**`)`
Description	This routine sets the memory pointer p for command c.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_setPtr(T6963C_grHomeAddr + start, T6963C_ADDRESS_POINTER_SET)`

Spi_T6963C_waitReady

Prototype	**sub procedure** `Spi_T6963C_waitReady`
Description	This routine pools the status byte, and loops until ready.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_waitReady`

Spi_T6963C_fill

Prototype	`sub procedure Spi_T6963C_fill(dim v as byte, dim start, len as word)`
Description	This routine fills length with bytes to controller memory from start address.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_fill(0x33,0x00FF,0x000F)`

Spi_T6963C_dot

Prototype	`sub procedure Spi_T6963C_dot(dim x, y as integer, dim color as byte)`
Description	This routine sets current graphic work panel. It sets the pixel dot (x0, y0). pcolor = T6963C_[WHITE[BLACK].
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_dot(x0, y0, pcolor)`

Spi_T6963C_write_char

Prototype	`sub procedure Spi_T6963C_write_char(dim c, x, y, mode as byte)`
Description	This routine sets current text work panel. It writes char c row x line y. mode = T6963C_ROM_MODE_[OR|EXOR|AND]
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_write_char('A',22,23,AND)`

Spi_T6963C_write_text

Prototype	`sub procedure Spi_T6963C_write_text(dim byref str as byte[ 10] , dim x, y, mode as byte)`
Description	This sets current text work panel. It writes string str row x line y. mode = T6963C_ROM_MODE_[OR|EXOR|AND]
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_write_text("GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR)`

Spi_T6963C_line

Prototype	`sub procedure Spi_T6963C_line(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	This routine current graphic work panel. It's draw a line from (x0, y0) to (x1, y1). pcolor = T6963C_[WHITE[BLACK]
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_line(0, 0, 239, 127, T6963C_WHITE)`

Spi_T6963C_rectangle

Prototype	`sub procedure Spi_T6963C_rectangle(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	It sets current graphic work panel. It draws the border of the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE)`

Spi_T6963C_box

Prototype	`sub procedure Spi_T6963C_box(dim x0, y0, x1, y1 as integer, dim pcolor as byte)`
Description	This routine sets current graphic work panel. It draws a solid box in the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_box(0, 119, 239, 127, T6963C_WHITE)`

Spi_T6963C_circle

Prototype	`sub procedure Spi_T6963C_circle(dim x, y as integer, dim r as longint, dim pcolor as word)`
Description	This routine sets current graphic work panel. It draws a circle, center is (x, y), diameter is r. pcolor = T6963C_[WHITE[BLACK]
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_circle(120, 64, 110, T6963C_WHITE)`

Spi_T6963C_image

Prototype	`sub procedure Spi_T6963C_image(const pic as ^byte)`
Description	This routine sets current graphic work panel : It fills graphic area with picture pointer by MCU. MCU must fit the display geometry. For example : for a 240x128 display, MCU must be an array of (240/8)*128 = 3840 bytes .
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_image(my_image)`

Spi_T6963C_sprite

Prototype	`sub procedure Spi_T6963C_sprite(dim px, py as byte, const pic as ^byte, dim sx, sy as byte)`
Description	This routine sets current graphic work panel. It fills graphic rectangle area (px, py)-(px + sx, py + sy) witch picture pointed by MCU. Sx and sy must be the size of the picture. MCU must be an array of sx*sy bytes.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_sprite(76, 4, einstein, 88, 119) ' draw a sprite`

Spi_T6963C_set_cursor

Prototype	`sub procedure Spi_T6963C_set_cursor(dim x, y as byte)`
Description	This routine sets cursor row x line y.
Requires	Ports must be initialized. See Spi_`T6963C_init`.
Example	`Spi_T6963C_set_cursor(cposx, cposy)`

Spi_T6963C_clearBit

Prototype	`sub procedure Spi_T6963C_clearBit(b as byte)`
Description	Clear control bit.
Requires	Ports must be initialized. See Spi_`T6963C_init`.
Example	`Spi_T6963C_clearBit(b)`

Spi_T6963C_setBit

Prototype	`sub procedure Spi_T6963C_setBit(dim b as byte)`
Description	Set control bit.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_setBit(b)`

Spi_T6963C_negBit

Prototype	`sub procedure Spi_T6963C_negBit(dim b as byte)`
Description	Neg control bit.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_negBit(b)`

Spi_T6963C_displayGrPanel

Prototype	`sub procedure Spi_T6963C_displayGrPanel(dim n as word)`
Description	Display graphic panel number n.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_displayGrPanel(n)`

Spi_T6963C_displayTxtPanel

Prototype	`sub procedure Spi_T6963C_displayTxtPanel(dim n as word)`
Description	Display text panel number n.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_displayTxtPanel(n)`

Spi_T6963C_setGrPanel

Prototype	`sub procedure Spi_T6963C_setGrPanel(dim n as word)`
Description	Compute graphic start address for panel number n.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_setGrPanel(n)`

Spi_T6963C_setTxtPanel

Prototype	`sub procedure Spi_T6963C_setTxtPanel(dim n as word)`
Description	Compute text start address for panel number n.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_setTxtPanel(n)`

Spi_T6963C_panelFill

Prototype	`sub procedure Spi_T6963C_panelFill(dim v as word)`
Description	Fill full #n panel with v bitmap (0 to clear).
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_panelFill(v)`

Spi_T6963C_grFill

Prototype	`sub procedure Spi_T6963C_grFill(dim v as word)`
Description	Fill graphic #n panel with v bitmap (0 to clear).
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_grFill(v)`

Spi_T6963C_txtFill

Prototype	`sub procedure Spi_T6963C_txtFill(dim v as word)`
Description	Fill text #n panel with char v + 32 (0 to clear).
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_txtFill(v)`

Spi_T6963C_cursor_height

Prototype	`sub procedure Spi_T6963C_cursor_height(dim n as word)`
Description	Set cursor size.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_cursor_height(n)`

Spi_T6963C_graphics

Prototype	`sub procedure Spi_T6963C_graphics(dim n as word)`
Description	Set graphics on/off.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_graphics(1)`

Spi_T6963C_text

Prototype	`sub procedure Spi_T6963C_text(dim n as word)`
Description	Set text on/off.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_text(1)`

Spi_T6963C_cursor

Prototype	`sub procedure Spi_T6963C_cursor(dim n as word)`
Description	Set cursor on/off.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_cursor(1)`

Spi_T6963C_cursor_blink

Prototype	`sub procedure Spi_T6963C_cursor_blink(dim n as word)`
Description	Set cursor blink on/off.
Requires	GLCD needs to be initialized, see `Spi_T6963C_Config`.
Example	`Spi_T6963C_cursor_blink(0)`

Spi_T6963C_Config_240x128

Prototype	**sub procedure** `Spi_T6963C_Config_240x128`
Description	Initialize T6963C based GLCD (240x128 pixels) with default settings for mE GLCD's.
Requires	Spi_Init; must be called before initializing SPI Toshiba T6963C Graphic LCD.
Example	`Spi_T6963C_Config_240x128`

Spi_T6963C_Config_240x64

Prototype	**sub procedure** `Spi_T6963C_Config_240x64`
Description	Set graphics on/off.
Requires	Initialize T6963C based GLCD (240x64 pixels) with default settings for mE GLCD's.
Example	`Spi_T6963C_Config_240x64`

Library Example

The following drawing demo tests advanced routines of SPI T6963C GLCD library.

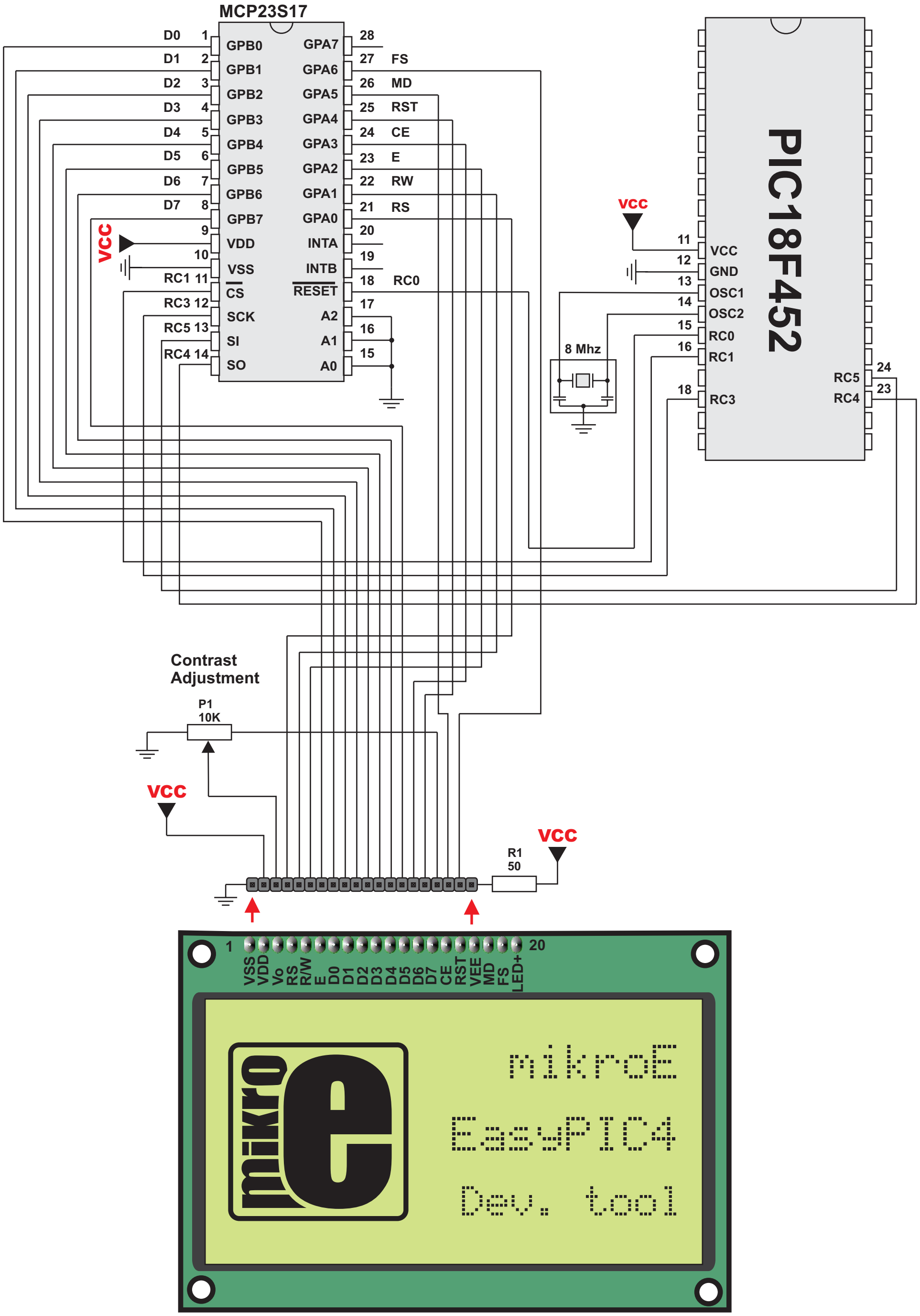
```
program Spi_T6963C_240x128

include "T6963C_Consts"
include "bitmap"
include "bitmap2"

dim   panel as byte          ' current panel
            i as word         ' general purpose register
         curs as byte        ' cursor visibility
        cposx,
        cposy as word        ' cursor x-y position
        txtcols as byte      ' number of text colons

main:
  PORTB   = 0
  TRISB   = 0xFF


//continues...
```

```
//continued...

'    * init display for 240 pixel width and 128 pixel height
'    * 8 bits character width
'    * data bus on PORTF
'    * control bus on PORTD
'    * bit 2 is !WR
'    * bit 1 is !RD
'    * bit 0 is !CD
'    * bit 4 is RST
'    *
'    * chip enable, reverse on, 8x8 font internaly set in library
  SPI_Init()
  Spi_T6963C_Init_240x128()
  Spi_T6963C_panelFill(0)
  ' enable both graphics and text display at the same time
  Spi_T6963C_graphics(1)
  Spi_T6963C_text(1)

  panel = 0
  i = 0
  curs = 0
  cposy = 0
  cposx = 0
  txtcols = 240 div 8          ' calculate number of text colomns
                   ' (grafic display width divided by font width)
   ' text messages
  Spi_T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0, 0,
T6963C_ROM_MODE_XOR)
  Spi_T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mE", 0, 15,
T6963C_ROM_MODE_XOR)

  ' cursor
  Spi_T6963C_cursor_height(8)        ' 8 pixel height
  Spi_T6963C_set_cursor(0, 0)        ' move cursor to top left
  Spi_T6963C_cursor(0)                ' cursor off

  ' draw rectangles
  Spi_T6963C_rectangle(0, 0, 239, 127, T6963C_WHITE)
  Spi_T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE)
  Spi_T6963C_rectangle(40, 40, 199, 87, T6963C_WHITE)
  Spi_T6963C_rectangle(60, 60, 179, 67, T6963C_WHITE)


//continues...
```

```
 //continued...

 'draw a cross
 Spi_T6963C_line(0, 0, 239, 127, T6963C_WHITE)
 Spi_T6963C_line(0, 127, 239, 0, T6963C_WHITE)

 ' draw solid boxes
 Spi_T6963C_box(0, 0, 239, 8, T6963C_WHITE)
 Spi_T6963C_box(0, 119, 239, 127, T6963C_WHITE)

 ' draw cicles
 Spi_T6963C_circle(120, 64, 10, T6963C_WHITE)
 Spi_T6963C_circle(120, 64, 30, T6963C_WHITE)
 Spi_T6963C_circle(120, 64, 50, T6963C_WHITE)
 Spi_T6963C_circle(120, 64, 70, T6963C_WHITE)
 Spi_T6963C_circle(120, 64, 90, T6963C_WHITE)
 Spi_T6963C_circle(120, 64, 110, T6963C_WHITE)
 Spi_T6963C_circle(120, 64, 130, T6963C_WHITE)

 Spi_T6963C_sprite(76, 4, einstein, 88, 119) ' draw a sprite

 Spi_T6963C_setGrPanel(1)           ' select other graphic panel

 Spi_T6963C_image(mikroPascal_logo_glcd_bmp)
 ' fill the graphic screen with a picture

 while true
' if RB1 is pressed, toggle the display between graphic panel 0 and
' graphic 1
        if(PORTB.1 <> 0) then
          panel = panel + 1
          panel = panel and 1
          Spi_T6963C_displayGrPanel(panel)
          Delay_ms(300)
        end if
' if RB2 is pressed, display only graphic panel
        if(PORTB.2 <> 0) then
          Spi_T6963C_graphics(1)
          Spi_T6963C_text(0)
          Delay_ms(300)
        end if


//continues...
```

```
//continued...

         ' if RB4 is pressed, display text and graphic panels
         if(PORTB.4 <> 0) then
           Spi_T6963C_graphics(1)
           Spi_T6963C_text(1)
           Delay_ms(300)
         end if

         ' if RB5 is pressed, change cursor
         if(PORTB.5 <> 0) then
             curs = curs + 1
             if(curs = 3) then
               curs = 0
             end if
             select case curs
               case 0
                   Spi_T6963C_cursor(0)
               case 1
                   Spi_T6963C_cursor(1)
                   Spi_T6963C_cursor_blink(1)
               case 2
                   Spi_T6963C_cursor(1)
                   Spi_T6963C_cursor_blink(0)
             end select
             Delay_ms(300)
           end if
       ' move cursor, even if not visible
       cposx = cposx + 1
       if(cposx = txtcols) then
         cposx = 0
         cposy = cposy + 1
         if(cposy = (128 div T6963C_CHARACTER_HEIGHT)) then
           ' if y end
           cposy = 0                                                 '
           ' graphic height (128) div character height
         end if
       end if
       Spi_T6963C_set_cursor(cposx, cposy)
       Delay_ms(100)
  wend
end.
```

Hardware Connection



Toshiba T6963C Graphic LCD (240x128)

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
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@mikroe.com***
Web: ***www.mikroe.com***