

mikroElektronika

Development tools - Books - Compilers

www.mikroelektronika.co.yu

BASIC COMPILER FOR ATMEL AVR MICROCONTROLLERS

mikroBASIC

f o r A V R

M a k i n g i t s i m p l e

manual
User's



Develop your applications quickly and easily with the world's most intuitive BASIC compiler for AVR Microcontrollers.

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.

Copyright © mikroElektronika 2003 - 2005.

This manual covers mikroBasic version 2.0 and the related topics. New versions may contain changes without prior notice.

COMPILER BUG REPORTS:

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

E-mail: office@mikroelektronika.co.yu

AVR, AVRmicro and AVRStudio is a Registered trademark of Atmel 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

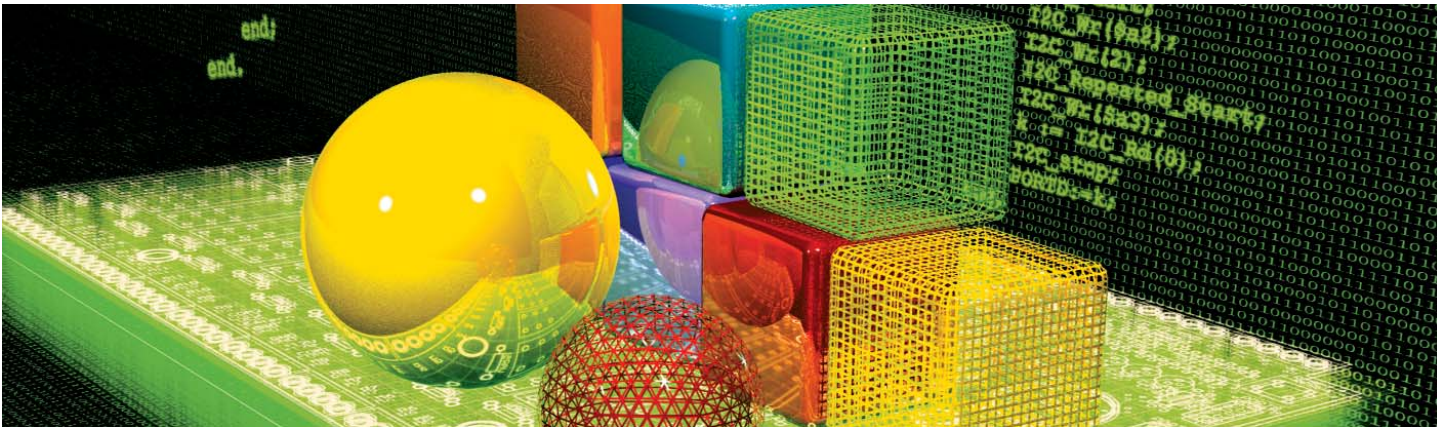


Table of Contents

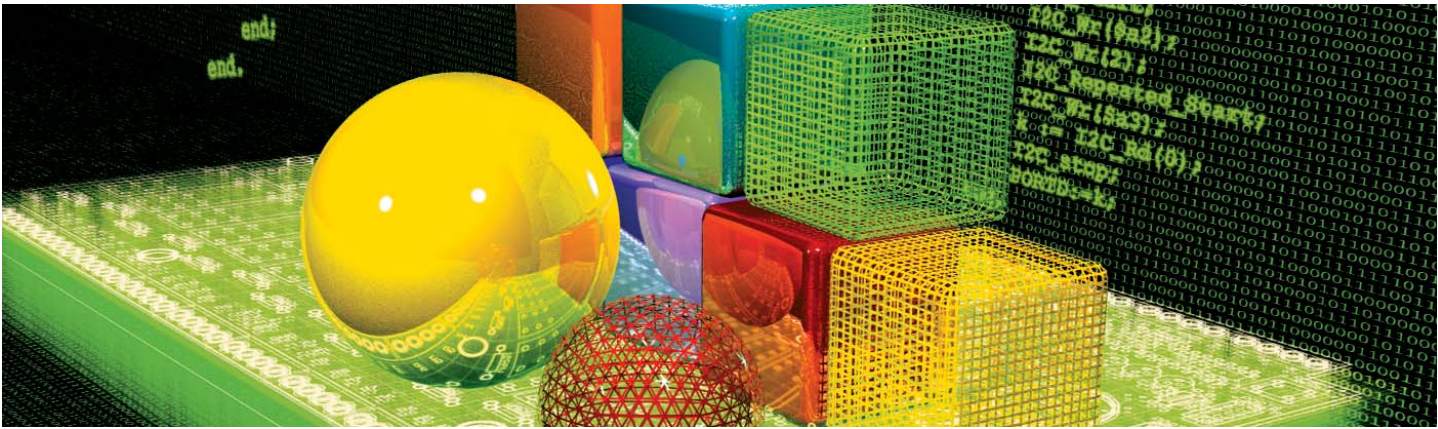
CHAPTER 1	mikroBasic IDE
CHAPTER 2	Building Applications
CHAPTER 3	mikroBasic Reference
CHAPTER 4	mikroBasic Libraries

CHAPTER 1: mikroBasic IDE	1
Quick Overview	1
Code Editor	3
Code Explorer	6
Debugger	7
Error Window	10
Statistics	11
Integrated Tools	14
Keyboard Shortcuts	17
CHAPTER 2: Building Applications	19
Projects	20
Source Files	21
Search Paths	21
Managing Source Files	21
Compilation	23
Output Files	23
Assembly View	23
Error Messages	24
CHAPTER 3: mikroBasic Language Reference	27
AVR Specifics	28
mikroBasic Specifics	29
Predefined Globals and Constants	29
Accessing Individual Bits	29
Interrupts	30
Linker Directives	31
Lexical Elements	31
Whitespace	32
Comments	33
Tokens	34
Literals	35
Integer Literals	35
Floating Point Literals	35
Character Literals	35
String Literals	36
Keywords	37
Identifiers	38

Punctuators	39
Program Organization	41
Scope and Visibility	44
Modules	45
Include Clause	45
Main Module	46
Other Modules	47
Variables	48
Constants	49
Labels	50
Symbols	51
Functions and Procedures	52
Functions	52
Procedures	53
Types	55
Simple Types	56
Arrays	57
Strings	58
Pointers	59
Structures	60
Types Conversions	62
Implicit Conversion	62
Explicit Conversion	63
Operators	64
Precedence and Associativity	64
Arithmetic Operators	65
Relational Operators	66
Bitwise Operators	67
Boolean Operators	70
Expressions	71
Statements	72
asm Statement	72
Assignment Statements	73
Conditional Statements	73
Iteration Statements	76
Jump Statements	78
Compiler Directives	81



CHAPTER 4: mikroBasic Libraries	85
Built-in Routines	86
Library Routines	92
ADC Library	93
Compact Flash Library	95
EEPROM Library	104
Flash Memory Library	106
TWI Library	108
Keypad Library	113
LCD Library (4-bit interface)	117
LCD Library (8-bit interface)	123
Graphic LCD Library	128
Multi Media Card Library	139
OneWire Library	149
PS/2 Library	153
PWM Library	157
Secure Digital Library	162
Software I2C Library	171
Software SPI Library	175
Software UART Library	178
Sound Library	181
SPI Library	185
USART Library	190
Util Library	194
Conversions Library	195
Delays Library	200
String Library	201

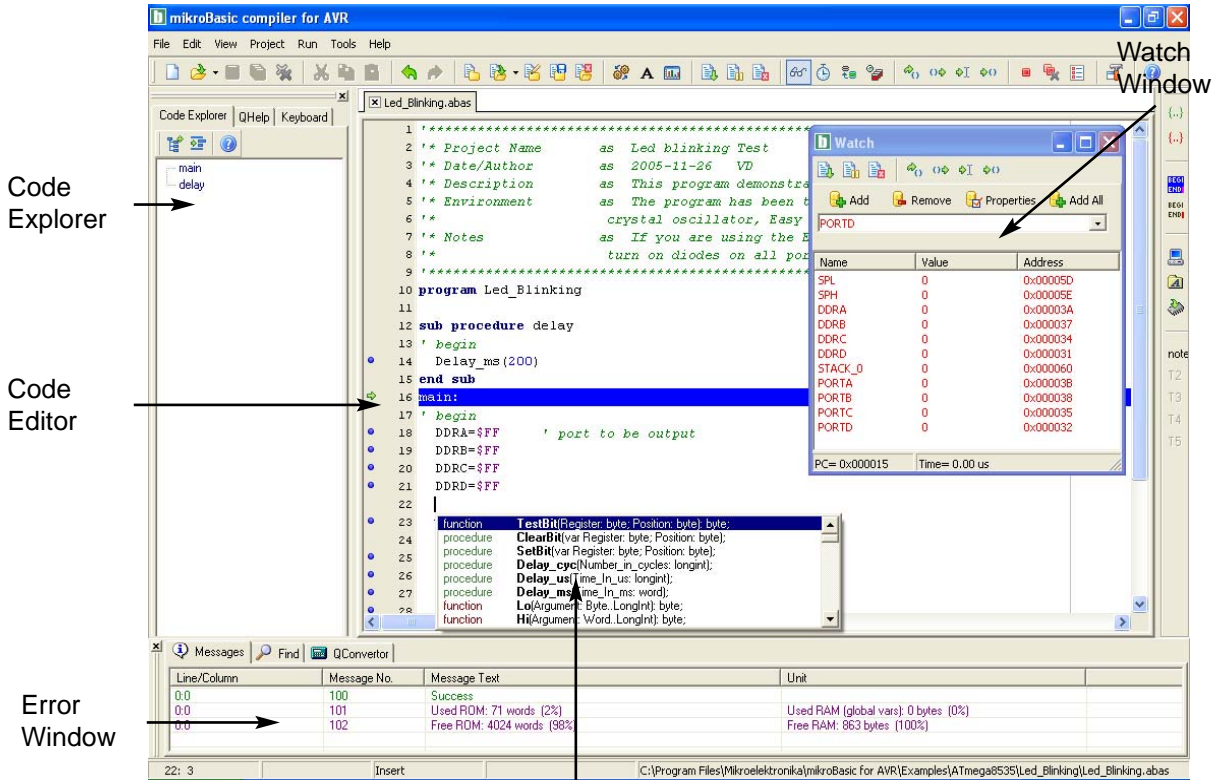


mikroBasic IDE

QUICK OVERVIEW

mikroBasic is a powerful, feature rich development tool for AVR 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.



Code Explorer

Code Editor

Error Window

Watch Window

Code Assistant

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.

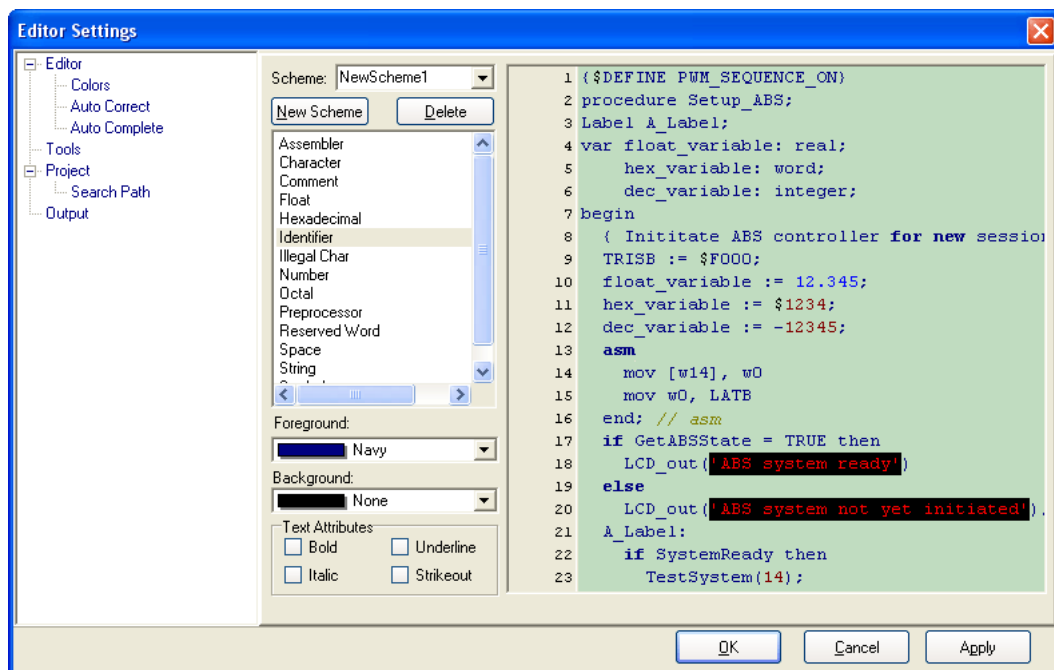
Advanced Editor features include:

- 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.



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.

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

Parameter Assistant [CTRL+SHIFT+SPACE]

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.

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

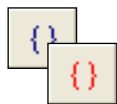
Code Template [CTR+J]

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.

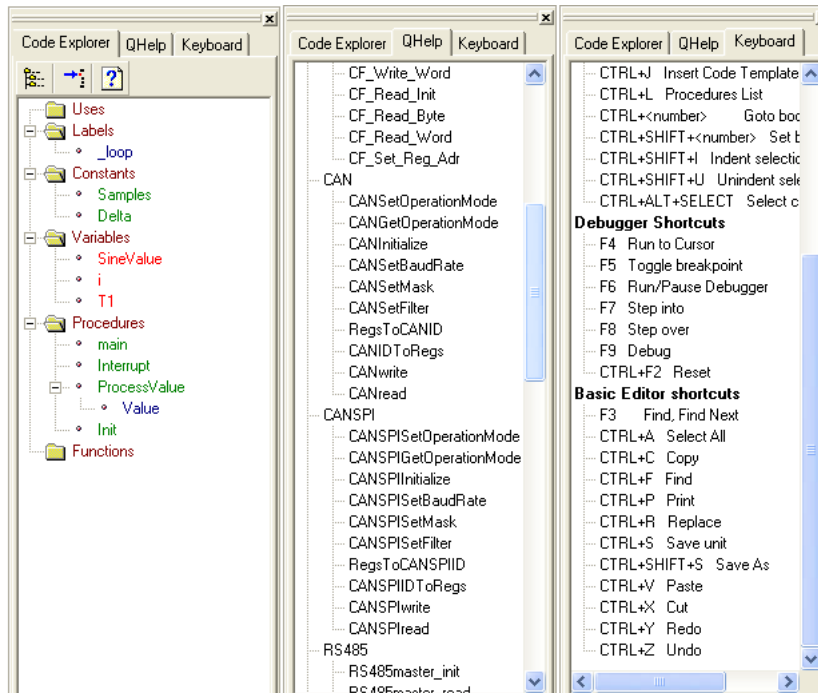
CODE EXPLORER

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.



DEBUGGER



Start Debugger.

Source-level Debugger is an integral component of mikroBasic development environment. It is designed to simulate operations of Atmel Technology's AVRmicros 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 AVR 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.



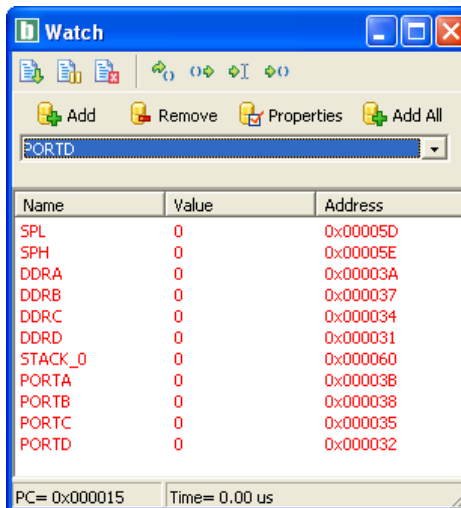
Toggle
Breakpoint.

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.



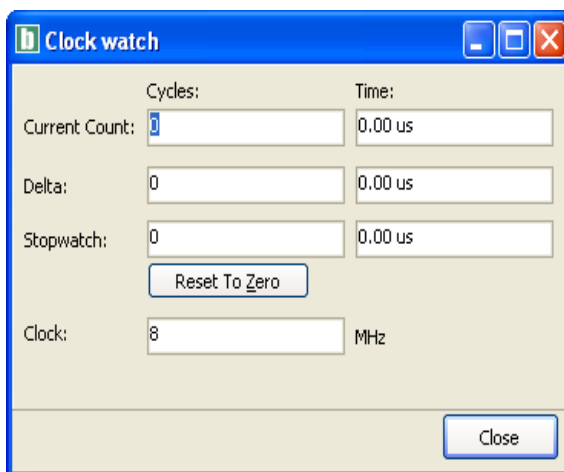
The Watch Window displays variables and registers of AVR, with their addresses and values. Values are updated as you go through the simulation. Use the drop-down menu to add and remove the items that you want to monitor. Recently changed items are colored red.

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.

Clockwatch Window

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

The Clockwatch Window displays the current count of cycles/time since the last Debugger action. Clockwatch 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).



Note: You can change the clock in the Clockwatch 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

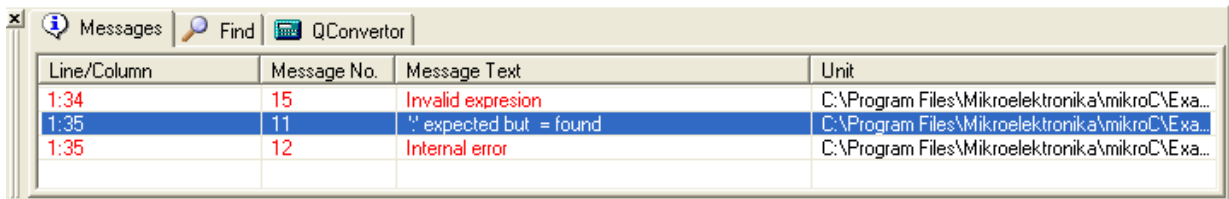
Debugger View RAM Window is available from the drop-down menu, View > Debug Windows > View RAM.

The View RAM Window displays the map of AVR'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 interfere with generation of hex.



Line/Column	Message No.	Message Text	Unit
1:34	15	Invalid expression	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

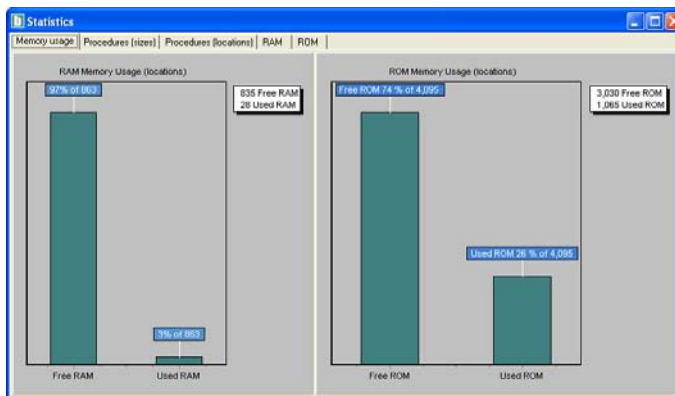


Statistics Icon.

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.

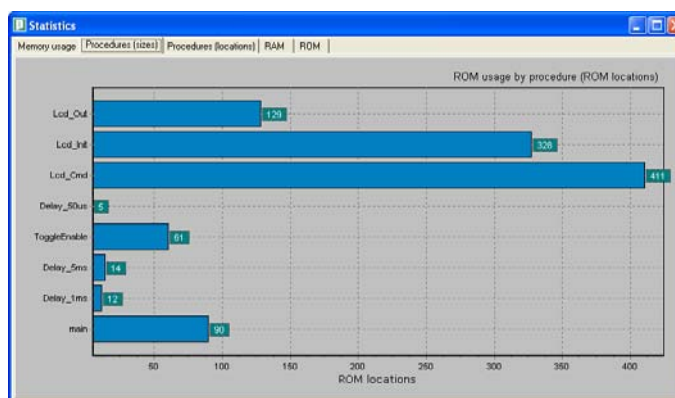


Procedures (Graph) Window

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

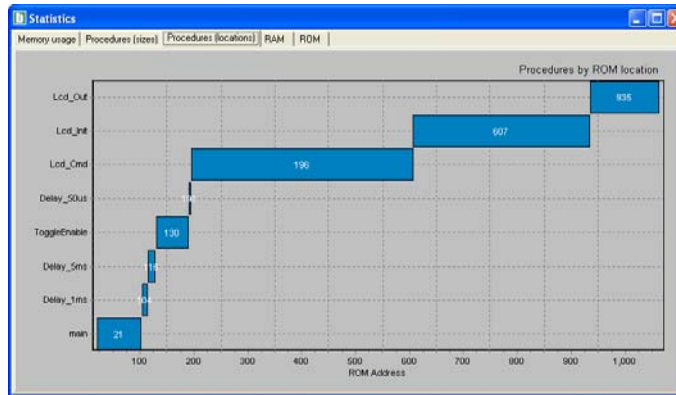
Procedures (Locations) Window

Displays how functions are distributed in microcontroller's memory.



Procedures (Details) Window

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



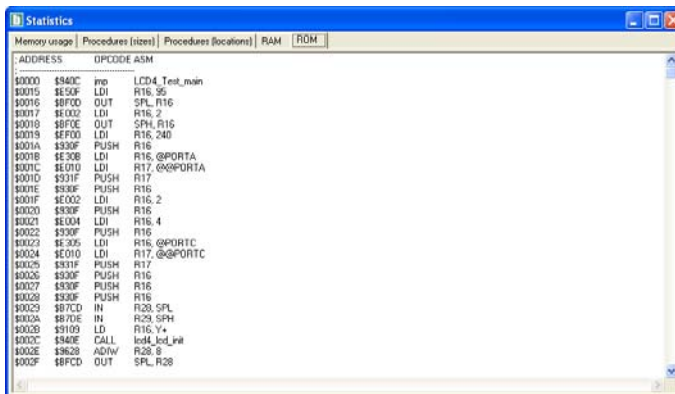
RAM Window

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

General purpose registers (GPR)		Special function registers (SFR)	
Address	Register	Address	Register
<SP> + 0x00		0x005D	SP
<SP> + 0x00	tmp	0x005E	SFH
<SP> + 0x00	tmp	0x005F	PORTC
<SP> + 0x00	tmpPh	0x0060	PORTA
<SP> + 0x00	a		
<SP> + 0x00	pr		
<SP> + 0x00	Flow		
<SP> + 0x00	control_post		
<SP> + 0x00	out_char		
<SP> + 0x00	Column		
<SP> + 0x00	test		
<SP> + 0x00	EH		
<SP> + 0x00	RS		
<SP> + 0x00	data_post		
<SP> + 0x00	nibble		
0x0060	STACK_0		
0x0061	STACK_1		
0x0062	STACK_2		

ROM Window

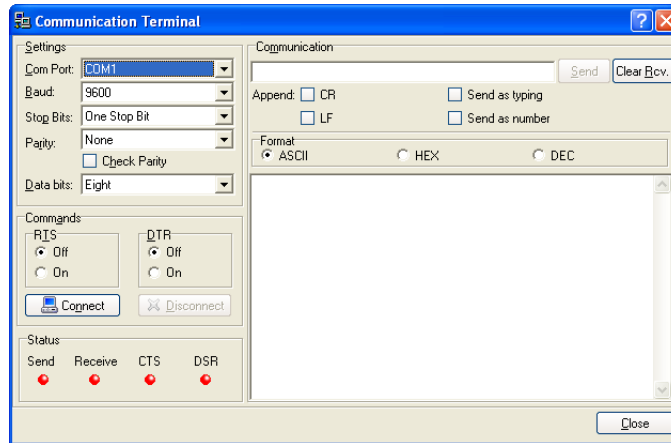
Lists op-codes and their addresses in form of a human readable hex code.



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.



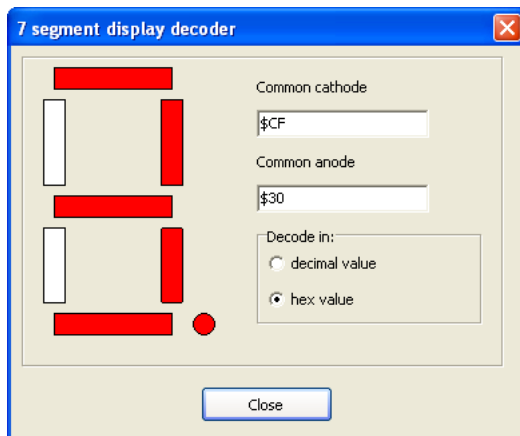
ASCII Chart

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.

ASCII chart							
0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7
10 LF	11 VT	12 FF	13 CR	14 SO	15 SI	16 DLE	17
20 DC4	21 NAK	22 SYN	23 ETB	24 CAN	25 EM	26 SUB	27
30 RS	31 US	32	33 !	34 "	35 #	36 \$	37
40 (41)	42 *	43 +	44 ,	45 -	46 .	47
50 2	51 3	52 4	53 5	54 6	55 7	56 8	57
60 <	61 =	62 >	63 ?	64 @	65 A	66 B	67
70 F	71 G	72 H	73 I	74 J	75 K	76 L	77
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87
90 Z	91 [92 \	93]	94 ^	95 _	96 `	97

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.



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

Basic Editor shortcuts

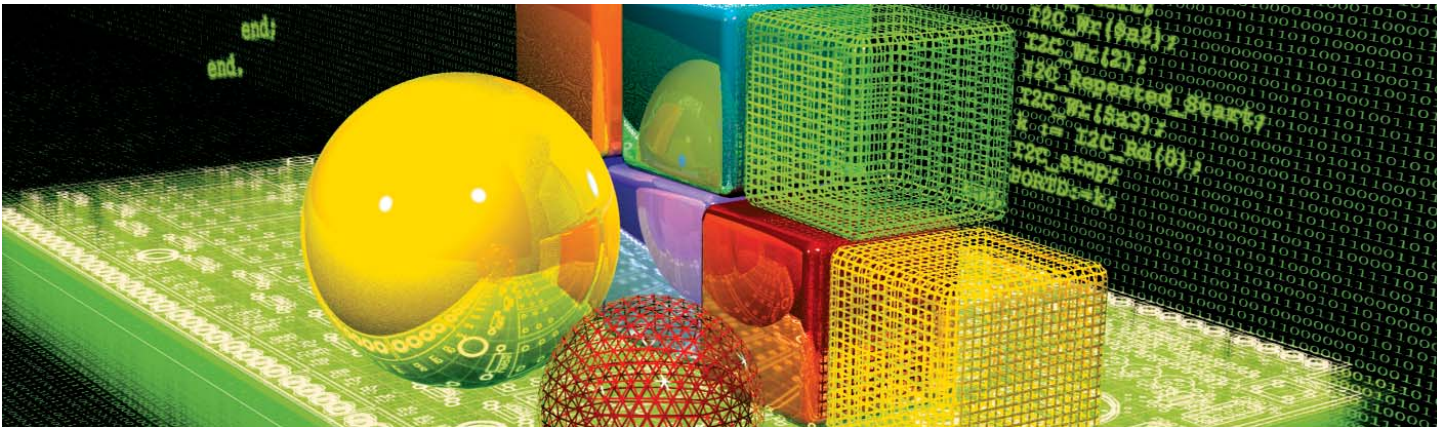
F3	Find, Find Next
CTRL+A	Select All
CTRL+C	Copy
CTRL+F	Find
CTRL+P	Print
CTRL+R	Replace
CTRL+S	Save unit
CTRL+SHIFT+S	Save As
CTRL+V	Paste
CTRL+X	Cut
CTRL+Y	Redo
CTRL+Z	Undo

Advanced Editor shortcuts

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

Debugger Shortcuts

F4	Run to Cursor
F5	Toggle breakpoint
F6	Run/Pause Debugger
F7	Step into
F8	Step over
CTRL+F8	Step out
F9	Debug
CTRL+F2	Reset
CTRL+F5	Add to Watch list
SHIFT+CTRL+F6	View RAM
CTRL+F6	View Clock
SHIFT+F6	Edit Registers



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 (.mc1 files) with other developers without disclosing the source code.

PROJECTS

mikroBasic organizes applications into *projects*, consisting of a single project file (extension `.abp`) and one or more source files (extension `.abas`). 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 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) 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, 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 `.abas`. List of source files relevant for the application is stored in project file with extension `.abp`, 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 `.abp`)

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 `.abas`, 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 `.abp`). Output files are summarized below:

Intel HEX file (`.hex`)

Intel style hex records. Use this file to program AVR MCU.

Binary mikro Compiled Library (`.mc1`)

Binary distribution of application that can be included in other projects.

List File (`.lst`)

Overview of AVR 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.

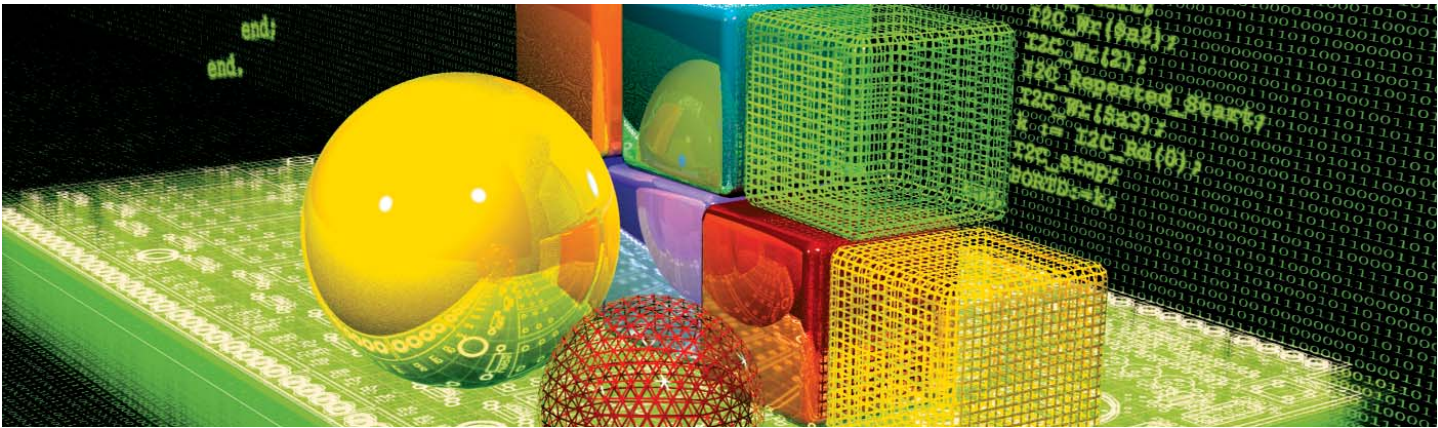
ERROR MESSAGES

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



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 AVR microcontrollers.

AVR SPECIFICS

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

Types Efficiency

First of all, you should know that AVR's ALU, which performs arithmetic operations, is optimized for working with bytes. Although mikroBasic is capable of handling very complex data types. 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.

Get to know your tool. When it comes down to calculus, not all AVRmicros are of equal performance. For example, ATtiny2313 lacks hardware resources to multiply two bytes, so it is compensated by a software algorithm. On the other hand, ATmega16 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.

mikroBasic limits the number of non-recursive nested calls to amount of available AVR stack memory (upper half of RAM memory).

On every function call number of used bytes by local variables, parameters and function results is limited to 64, due to AVR's hardware implementation. AVR's that don't have implemented instruction for indirect addressing with displacement (LDD Rd, Y+q; STD Y+q, Rd), are not allowed to use local variables, parameters and function. In that case you need to use only global variables.

mikroBASIC SPECIFICS

Predefined Globals and Constants

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

All AVR I/O 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 I/O registers and constants (such as `PORTB`, `DDRB`, etc). Identifiers are all in uppercase, identical to nomenclature in Atmel 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 PBO 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. `ADCSRA.ADEN`).

Interrupts

AVR chips have Interrupt Vector Table, and every interrupt source has one vector in table. (See datasheet). To make some procedure to be interrupt routine, you have to ORG her. Compiler will take care of putting vector of that procedure in Interrupt Vector Table - IVT. You can not use function, procedures with parameters or local variables like interrupt routines.

Routine Calls from Interrupt

You cannot call routines from within interrupt routine, but you can make a routine call from embedded assembly in interrupt. For this to work, called routine (func1 in further text) must fulfill the following conditions:

1. func1 does not use stack (or the stack is saved before call, and restored afterwards),
2. func1 must use global variables only.

The stated rules also apply to all the routines called from within func1.

Note: mikroBasic linker ignores calls to routines that occur only in interrupt assembler. For linker to recognize these routines, you need to make a call in Basic code, outside of interrupt body.

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. .

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 `.abas`.

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)

You can omit either the decimal integer or the decimal fraction (but not both). Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

mikroBasic limits floating-point constants to range
 $\pm 1.17549435082E38 \dots \pm 6.80564774407E38$.

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 I/O Registers are defined as global variables and represent reserved words that cannot be redefined (for example: PORTB, DDR, 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	float	or
abs	for	org
and	function	print
array	goto	procedure
asm	gosub	program
begin	if	read
boolean	include	select
case	in	sub
char	int	step
chr	integer	string
clear	interrupt	switch
const	is	then
dim	loop	to
div	label	until
do	mod	wend
double	module	while
else	new	with
end	next	xor
exit	not	

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 `.abp` contains information about the project, while modules, with extension `.abas`, 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 `.abas` files, in order specified by the search paths.

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

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 I/O Registers explicitly, as mikroBasic automatically declares relevant registers as global variables of `byte`. For example: `DDRB0`, `PORTB`, 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 I/O Registers
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 (Under Construction!!)	32-bit	$\pm 1.17549435082 * 10^{-38}$.. $\pm 6.80564774407 * 10^{38}$

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

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. 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)
```

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.

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
```

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.

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
<code>div</code>	division, rounds down to nearest integer	3
<code>mod</code>	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 (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:

$$a + 5 \geq c - 1.0 / e \quad \rightarrow \quad (a + 5) \geq (c - (1.0 / e))$$

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

Boolean Operators

Although mikroBasic does not support boolean type, you have Boolean operators at your disposal for building complex conditional expressions. These operators conform to standard Boolean logic, and return either TRUE (all ones) or FALSE (zero):

Operator	Operation
and	logical AND
or	logical OR
xor	logical exclusive OR
not	logical negation

Boolean operators associate from left to right. Negation operator not associates from right to left.

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.

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(Under Construction)

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
[#ELSEIF flag_1 THEN
    block of code 1
...
#ELSEIF flag_n THEN
    block of code n]
[#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`.

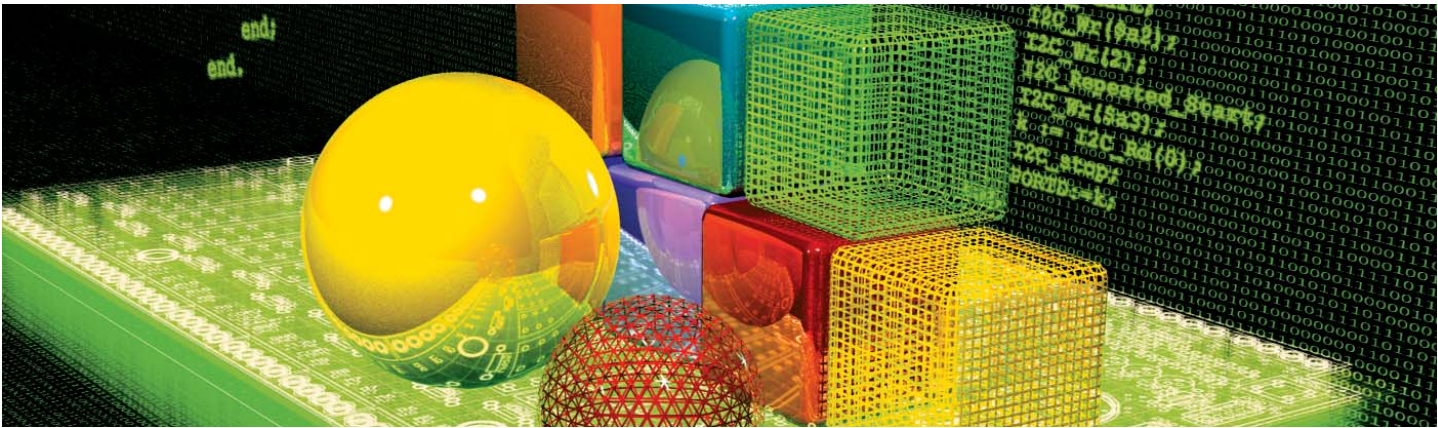
Here is a simple example:

```
' Uncomment the appropriate flag for your application:
'#DEFINE resolution8
'#DEFINE resolution10
'#DEFINE resolution12

#IFDEF resolution8 THEN
    ... ' code specific to 8-bit resolution
#ELSEIF resolution10 THEN
    ... ' code specific to 10-bit resolution
#ELSEIF resolution12 THEN
    ... ' code specific to 12-bit resolution
#ELSE
    ... ' default code
#endif
```

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).



mikroBasic Libraries

mikroBasic provides a number of built-in and library routines which help you develop your application faster and easier. Libraries for ADC, USART, SPI, I2C, 1-Wire, LCD, PWM, 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 (Under construction!!!)
- Clock_Khz
- Clock_Mhz

Inc

Prototype	<code>sub function Inc(dim byref par as longint) as longint</code>
Description	Increases parameter <code>par</code> 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	<code>sub function Dec(dim byref par as longint) as longint</code>
Description	Decreases parameter <code>par</code> 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	<code>sub function Chr(dim code as byte) as char</code>
Returns	Returns a character associated with the specified character code.
Description	Function returns a character associated with the specified character <code>code</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	<code>c = Chr(10) ' returns a linefeed character</code>

Ord

Prototype	<code>sub function Ord(dim character as char) as byte</code>
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	<code>c = Ord("A") ' returns 65</code>

SetBit

Prototype	<code>sub procedure SetBit(dim byref register as byte, dim rbit as byte)</code>
Description	Function sets the bit <code>rbit</code> of <code>register</code> . Parameter <code>rbit</code> 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	<code>SetBit(PORTB, 2) ' Set RB2</code>

ClearBit

Prototype	<code>sub procedure ClearBit(dim byref register as byte, dim rbit as byte)</code>
Description	Function clears the bit <code>rbit</code> of <code>register</code> . Parameter <code>rbit</code> 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	<code>ClearBit(PORTC, 7) ' Clear RC7</code>

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	<code>flag = TestBit(PORTE, 2) ' 1 if RE2 is set, otherwise 0</code>

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	<code>= Lo(0x1AC30F4) ' Equals 0xF4</code>

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	<code>a = Hi(0x1AC30F4) ' Equals 0x30</code>

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 (Under Construction !!!)

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

Clock_Mhz

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

LIBRARY ROUTINES

mikroBasic provides a set of libraries which simplifies the initialization and use of AVR 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
- Compact Flash Library
- EEPROM Library
- Flash Memory Library
- I2C Library
- Keypad Library
- LCD Library
- LCD8 Library
- Graphic LCD Library
- Multi Media Card Library
- OneWire Library
- PS/2 Library
- PWM Library
- Secure Digital Library
- Software I2C Library
- Software SPI Library
- Software UART Library
- Sound Library
- SPI Library
- USART Library
- Util Library

- Conversions Library
- Delays Library
- String Library

ADC Library

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

Adc_Read

Prototype	sub function <code>Adc_Read(dim channel as byte) as word</code>
Returns	10-bit unsigned value read from the specified ADC channel.
Description	<p>Initializes AVR'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 T_{ad} 4μS.</p> <p>Parameter <code>channel</code> represents the channel from which the analog value is to be acquired. For channel-to-pin mapping please refer to documentation for the appropriate AVR MCU.</p>
Requires	<p>AVR MCU with built-in ADC module. You should consult the Datasheet documentation for specific device (most devices from AVR families have it).</p> <p>Before using the function, be sure to configure the appropriate bits to designate the pins as input. Also, configure the desired pin as analog input, and set <code>Vref</code> (voltage reference value).</p>
Example	<code>tmp = Adc_Read(1) ' Read analog value from channel 1</code>

Library Example

This code snippet reads analog value from channel 3 and displays it on PORTC (two most significant bits) and PORTB.

```

program Adc_Test

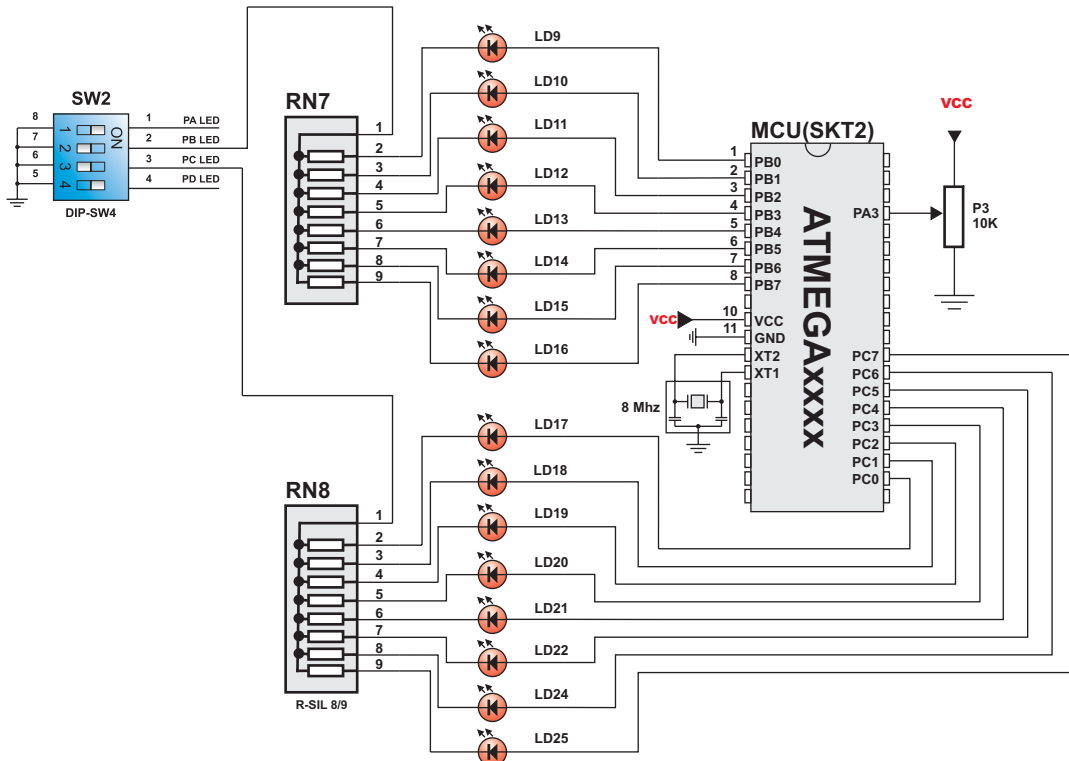
dim data as word

main:
    DDRB = $FF      ' PORTB output
    DDRC = $FF      ' PORTC output

    while true
        data = Adc_Read(3)    ' read channel 3
        PORTB = Lo(data)      ' display on led
        PORTC = Hi(data)      ' upper byte on portc leds
        Delay_ms(30)
    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_Total_Size  
Cf_Enable  
Cf_Disable  
Cf_Read_Init  
Cf_Read_Byte  
Cf_Write_Init  
Cf_Write_Byte  
  
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
```

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	<pre>' Wait until CF card is inserted: do nop loop until Cf_Detect = 1</pre>

Cf_Total_Size

Prototype	sub function Cf_Total_Size as logint
Returns	Card size in kilobytes.
Description	Returns size of Compact Flash card in kilobytes.
Example	size = Cf_Total_Size

Cf_Enable

Prototype	sub procedure Cf_Enable
Description	Enables the device. Routine needs to be called only if you have disabled the device by means of Cf_Disable. These two routines in conjunction allow you to free/occupy data line when working with multiple devices. Check the example at the end of the chapter.
Requires	Ports must be initialized. See Cf_Init.

Cf_Disable

Prototype	sub procedure Cf_Disable
Description	Routine disables the device and frees the data line for other devices. To enable the device again, call Cf_Enable. These two routines in conjunction allow you to free/occupy data line when working with multiple devices. Check the example at the end of the chapter.
Requires	Ports must be initialized. See Cf_Init.
Example	Cf_Disable

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_Fat_Init

Prototype	sub procedure Cf_Fat_Init(dim byref control_port as byte , dim wr, rd, a2, a1, a0, ry, ce, cd as byte , dim byref data_port as byte);
Returns	“1” if card is present, “0” if card is not present
Description	Initializes ports appropriately for FAT operations with CF card. Specify two different ports: ctrlport and dataport. wr, rd, a2, a1, a0, ry, ce and cd are pin nummbers on control port.
Requires	Nothing.
Example	Cf_Fat_Init(PORTD,6,5,2,1,0,7,3,4, 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
Description	Assigns file for FAT operations. If file isn't present, function creates new file with given filename. filename parameter is filename of file. 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)
Description	Opens file for reading. size is size of file in bytes.
Requires	CF must be initialized. 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. <i>bdata</i> is data read from file.
Requires	CF must be initialized. 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
Description	Opens file for writing. If there is file with given filename, procedure will overwrite the file.
Requires	CF must be initialized. 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
Description	Opens file for writing. This procedure continues writing from the last byte in file.
Requires	CF must be initialized. 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	CF must be initialized. 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 , data_len as word)
Description	Writes data to CF. <i>fdata</i> parameter is data written to CF. <i>data_len</i> number of bytes that is written to CF.
Requires	CF must be initialized. 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)
Description	Sets time attributes of file. You can set file year, month, day. hours, mins, seconds.
Requires	CF must be initialized. 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)

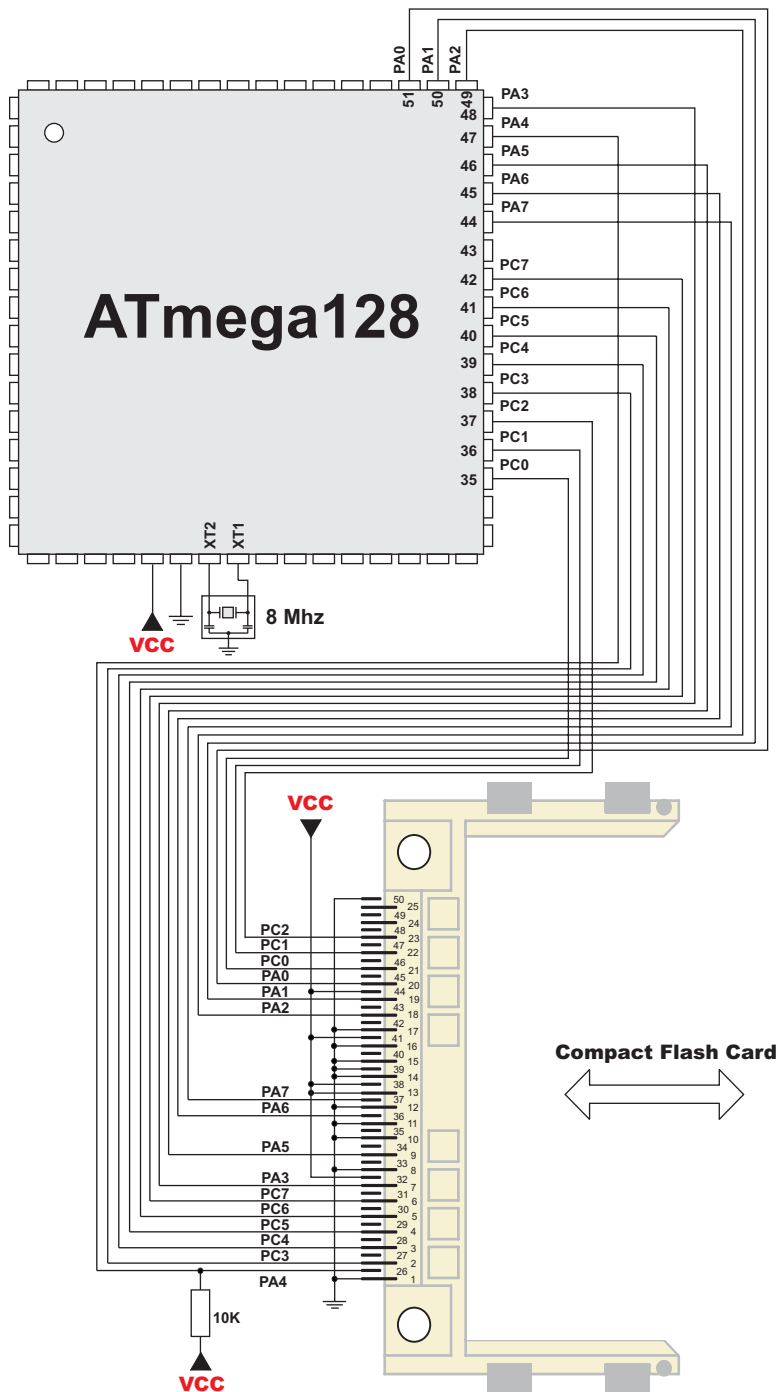
Cf_Fat_Get_File_Date

Prototype	sub procedure Cf_Fat_Get_File_Date(dim byref year as word , dim byref month, day, hours, mins as byte)
Returns	Nothing.
Description	Reads time attributes of file. You can read file year, month, day, hours, mins, seconds.
Requires	CF must be initialized. 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
Description	This function returns size of file in bytes.
Requires	CF must be initialized. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	Cf_Fat_Get_File_Size

HW Connection



EEPROM Library

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

Library Routines

Eeprom_Read
Eeprom_Write

Eeprom_Read

Prototype	sub function Eeprom_Read(dim address as byte) 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 256 locations.
Requires	Requires EEPROM module. Ensure minimum 20ms delay between successive use of routines Eeprom_Write and Eeprom_Read. Although AVR will write the correct value, Eeprom_Read might return an undefined result.
Example	take = Eeprom_Read(\$3F)

Eeprom_Write

Prototype	<code>sub procedure Eeprom_Write(dim address, data as byte);</code>
Description	Writes data to specified address. Parameter address is of byte type, which means it can address only 256 locations. Be aware that all interrupts will be disabled during execution of Eeprom_Write routine.
Requires	Requires EEPROM unit. Ensure minimum 20ms delay between successive use of routines Eeprom_Write and Eeprom_Read. Although AVR will write the correct value, Eeprom_Read might return an undefined result.
Example	<code>Eeprom_Write(\$32)</code>

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 as byte

main:
  DDRB = 0
  for i = 0 to 20
    Eeprom_Write(i, i + 6)
  next i

  for i = 0 to 20
    PORTB = Eeprom_Read(i)
    Delay_ms(300)
  next i
end.

```

Flash Memory Library

This library provides routines for accessing microcontroller Flash memory.

Library Routines

Flash_Read
Flash_Write

Flash_Read

Prototype	<code>sub function Flash_Read(dim address as word) as byte</code>
Returns	Returns data byte from Flash memory.
Description	Reads data from the specified address in Flash memory.
Example	<code>Flash_Read(\$D00)</code>

Flash_Write

Prototype	<code>sub procedure Flash_Write(dim address, data as word)</code>
Description	Writes chunk of data to Flash memory.
Example	<pre>' Write consecutive values in 64 consecutive locations for i = 0 to 63 toWrite[i] = i next i Flash_Write(\$0D00, toWrite)</pre>

Library Example

The example writes 64 consecutive values to 64 consecutive locations in flash memory. Then, it verifies the written data, with error indication on PORTB.

```
program flash_avr

const FLASH_ERROR = $FF
const FLASH_OK = $AA

dim toRead, i as byte
dim toWrite as byte[64]

main:
  DDRB = 0                                ' PORTB is output

  for i = 0 to 63                          ' Initialize array
    toWrite[i] = i
  next i

  Flash_Write($0D00, toWrite)              ' Write array contents to address 0x0D00

  ' Verify write
  PORTB = 0                                ' Turn off PORTB
  toRead = FLASH_ERROR                     ' Initialize error state

  for i = 0 to 63
    toRead = Flash_Read($0D00+i)           ' Read 64 locations starting from 0x0D00
    if toRead <> toWrite[i] then           ' Stop at first error
      PORTB = FLASH_ERROR                  ' Indicate error
      break                                 ' Stop verify
    else PORTB = FLASH_OK                   ' Indicate no error
    end if
  next i
end.
```

TWI (I2C) Library

TWI full master MSSP module is available with a number of AVR0 MCU models. mikroBasic provides TWI library which supports the master TWI mode.

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

Library Routines

```
Twi_Init
Twi_Start
Twi_Repeated_Start
Twi_Is_Idle
Twi_Rd
Twi_Wr
Twi_Stop
```

I2C_Init

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

TwI_Start

Prototype	<code>sub function TwI_Start as byte</code>
Returns	If there is no error, function returns 0.
Description	Determines if TwI bus is free and issues START signal.
Requires	TwI must be configured before using this function. See TwI_Init.
Example	<code>if TwI_Start = 0 then ...</code>

TwI_Repeated_Start

Prototype	<code>sub procedure TwI_Repeated_Start</code>
Description	Issues repeated START signal.
Requires	TwI must be configured before using this function. See TwI_Init.
Example	<code>TwI_Repeated_Start</code>

TwI_Is_Idle

Prototype	<code>sub function TwI_Is_Idle as byte</code>
Returns	Returns 1 if TwI bus is free, otherwise returns 0.
Description	Tests if TwI bus is free.
Requires	TwI must be configured before using this function. See TwI_Init.
Example	<code>if TwI_Is_Idle then ...</code>

TwI_Rd

Prototype	sub function TwI_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 TwI_Start.
Example	<code>tmp = TwI_Rd(0) ' Read data and send not acknowledge signal</code>

TwI_Wr

Prototype	sub function TwI_Wr(dim data as byte) as byte
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter data) via TwI bus.
Requires	START signal needs to be issued in order to use this function. See TwI_Start.
Example	<code>TwI_Write(\$A3)</code>

TwI_Stop

Prototype	sub procedure TwI_Stop
Description	Issues STOP signal.
Requires	TwI must be configured before using this function. See TwI_Init.

Library Example

This code demonstrates use of TWI Library procedures and functions. AVR 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 TWI 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 AVR).

```

program Eeprom_test

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

main:
    Twi_Init(100000)           ' Initialize full master mode
    DDRD = 0                  ' PORTD is output
    PORTD = $FF               ' Initialize PORTD
    Twi_Start                 ' Issue Twi start signal
    Twi_Wr($A2)               ' Send byte via Twi(command to 24c02)
    EE_adr = 2
    Twi_Wr(EE_adr)            ' Send byte(address for EEPROM)
    EE_data = $AA
    Twi_Wr(EE_data)           ' Send data(data that will be written)
    Twi_Stop                  ' Issue Twi stop signal

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

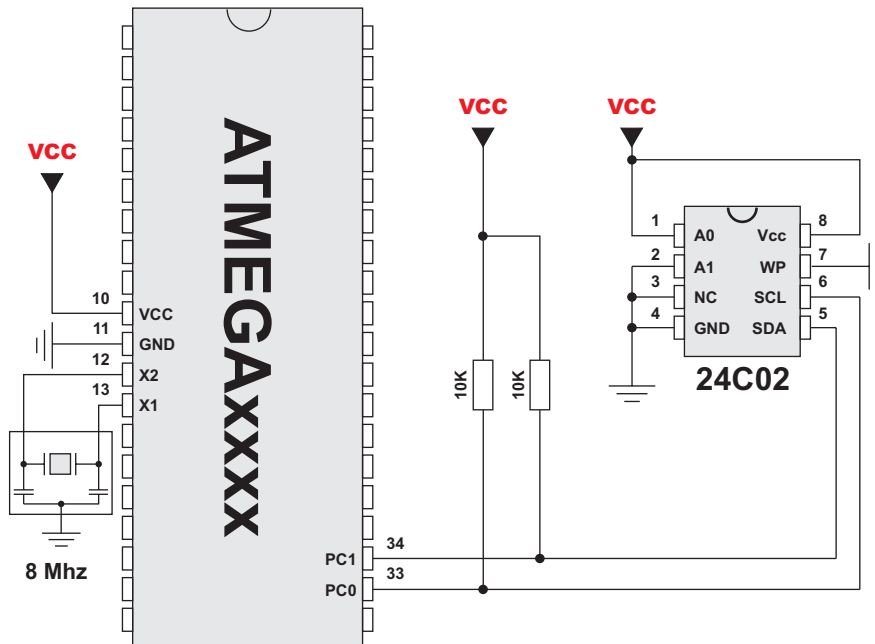
    Twi_Start                 ' Issue Twi start signal
    Twi_Wr($A2)               ' Send byte via Twi
    EE_adr = 2
    Twi_Wr(EE_adr)            ' Send byte(address for EEPROM)
    Twi_Repeated_Start       ' Issue Twi signal repeated start
    Twi_Wr($A3)               ' Send byte (request data from EEPROM)
    k = Twi_Rd(1)             ' Read the data
    Twi_Stop                  ' Issue Twi 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	<code>sub function Keypad_Released as word</code>
Returns	1..16, depending on the key.
Description	Call to <code>Keypad_Released</code> 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 <code>Keypad_Init</code> .
Example	<code>kp = Keypad_Released</code>

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(PORTC, 4, 2, PORTA, LCD_HI_NIBBLE) ' 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

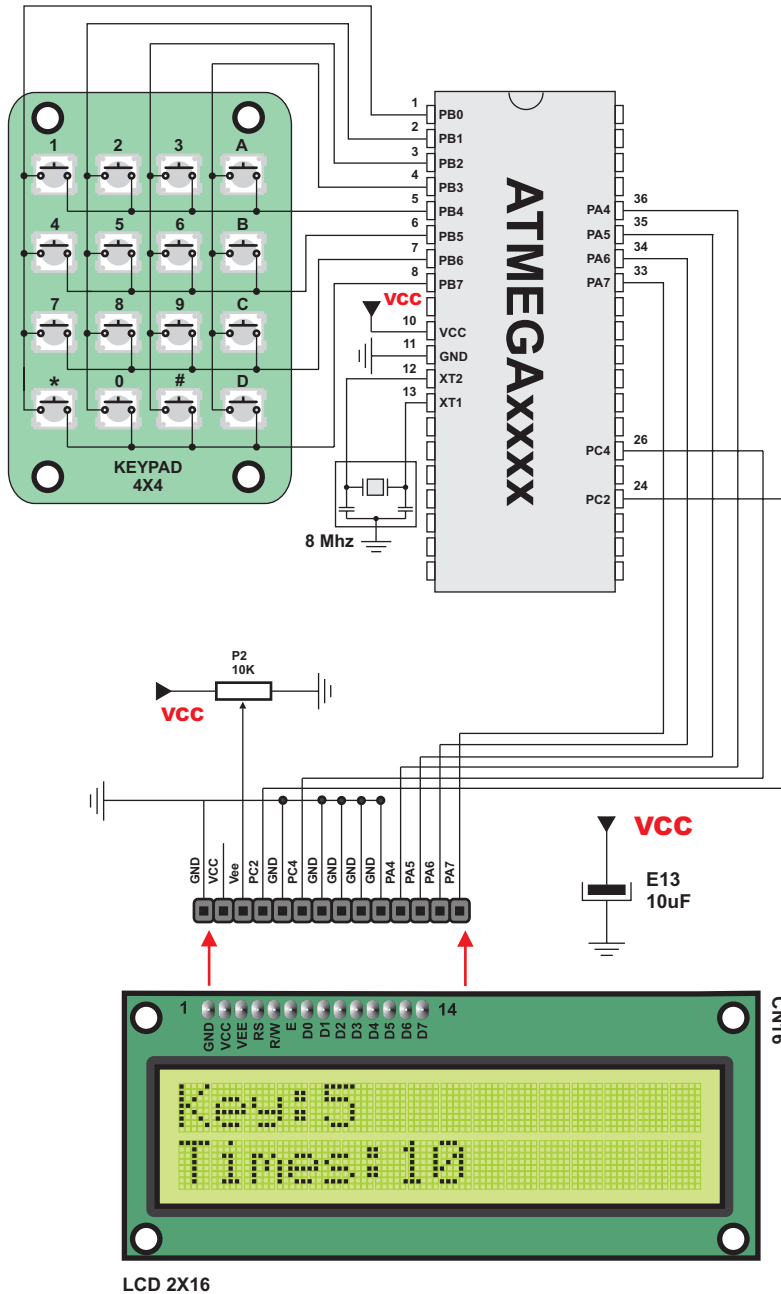
    '--- Wait for key to be pressed
    while kp = 0
      '--- un-comment one of the keypad reading functions
      kp = Keypad_Released
      'kp = Keypad_Read
    wend

    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 AVR and LCD are given at the end of the chapter.

Note: Be sure to designate port with LCD as output, before using any of the following library functions.

Library Routines

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

Lcd_Init

Prototype	sub procedure Lcd_Init(dim byref control_port as byte , dim EN, RS as byte , dim byref data_port as byte , dim nibble as byte)
Description	Initializes LCD at port with pin settings parameters : RS, EN, D7 .. D4. RW pin on LCD must be connected to GND.
Example	Lcd_Init(PORTC, 4, 2, PORTA, LCD_HI_NIBBLE)

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_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_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_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_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 below.
Requires	Port with LCD must be initialized. See 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

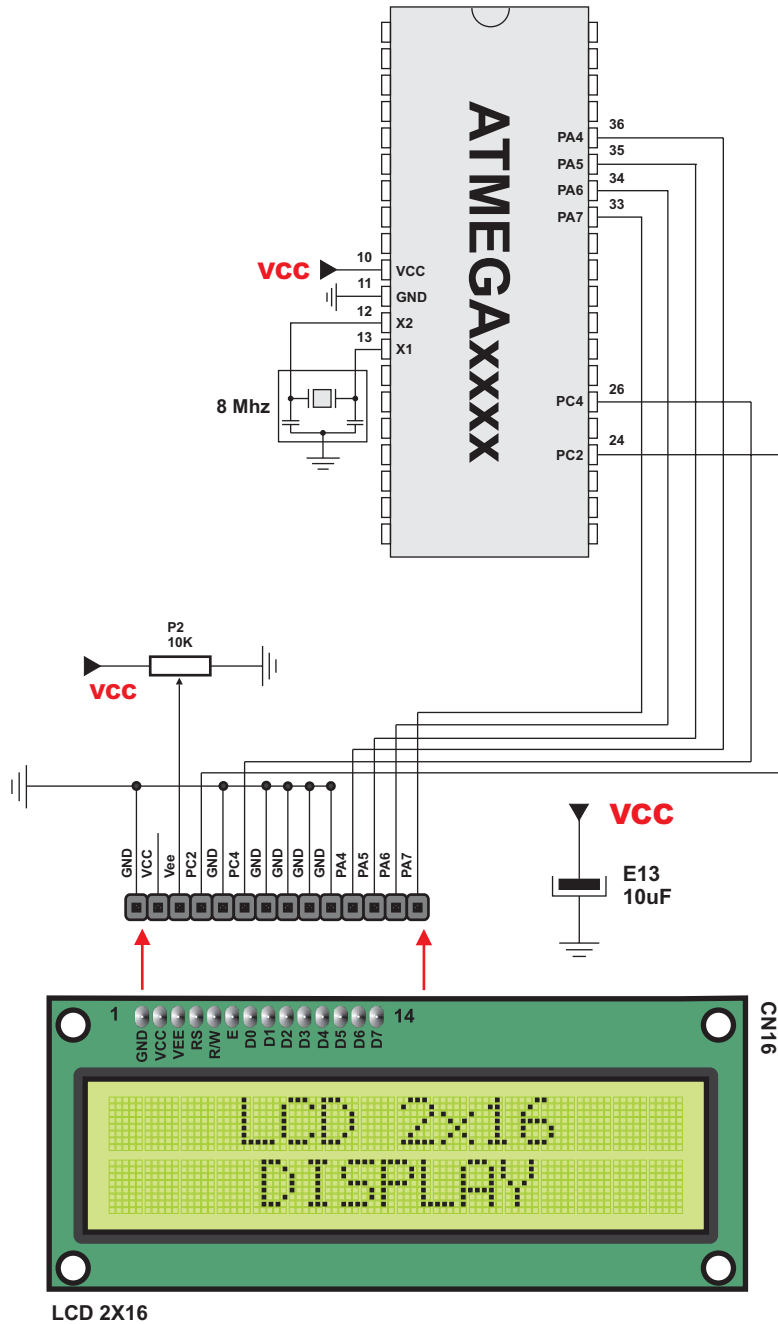
Library Example

```
program Lcd_default_test

dim text as char[20]

main:
  Lcd_Init(PORTC, 4, 2, PORTA, LCD_HI_NIBBLE)  ' Initialize LCD
  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 AVR and LCD are given at the end of the chapter.

Note: Be sure to designate Control and Data ports with LCD as output, before using any of the following functions.

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 control_port as byte , dim EN, RS as byte , dim byref data_port as byte)
Description	Initializes LCD at port with pin settings parameters : RS, EN, D7 .. D4. RW pin on LCD must be connected to GND.
Example	Lcd8_Init(PORTC, 4, 2, PORTA) ' <i>Initalize LCD8</i>

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_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_Init.
Example	Lcd8_Out_Cp("Here!") ' Print "Here!" at current cursor position

Lcd8_Chr

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

Lcd8_Chr_Cp

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

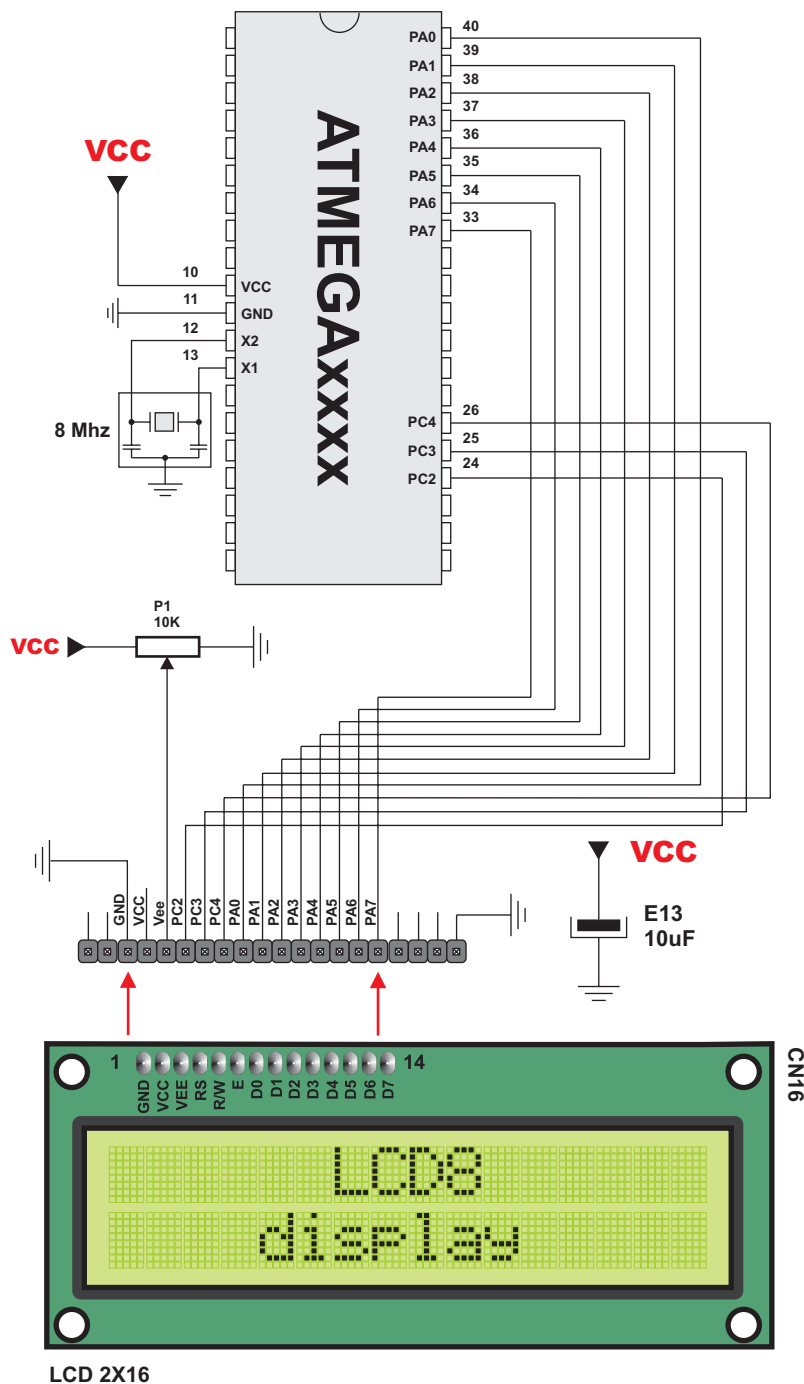
Lcd8_Cmd

Prototype	<code>sub procedure Lcd8_Cmd(dim command as byte)</code>
Description	Sends <code>command</code> 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 <code>Lcd8_Init</code> .
Example	<code>Lcd8_Cmd(LCD_CLEAR) ' Clear LCD display</code>

Library Example

```
program Lcd8_default_test
main:
  DDRC.4 = 1
  PORTC.4 = 0 ' for putting the reset line on GND
  Lcd8_Init(PORTC, 4, 2, PORTA)
  Lcd8_Out(2,1,"mikroElektronika")
end.
```

Hardware Connection



GLCD Library

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

Note: Be sure to designate port with GLCD as output, before using any of the following library procedures or functions.

Library Routines

```
Glcd_Init  
Glcd_Disable  
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_Partial_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 before using other routines of GLCD library.
Example	Glcd_Init(PORTB, 2, 0, 3, 5, 7, 1, PORTC)

Glcd_Disable

Prototype	sub procedure Glcd_Disable
Description	Routine disables the device and frees the data line for other devices. To enable the device again, call any of the library routines; no special command is required.
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Disable

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 <code>page</code> can be 0..7.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Set_Page(5)</code>

Glcd_Set_X

Prototype	sub procedure Glcd_Set_X(dim x as byte)
Description	Positions to <code>x</code> dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Set_X(25)</code>

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 <code>Glcd_Set_Side</code> , <code>Glcd_Set_X</code> , and <code>Glcd_Set_Page</code> to specify an exact position on GLCD. Then, you can use <code>Glcd_Write_Data</code> or <code>Glcd_Read_Data</code> on that location.
Requires	Reads data from from the current location of GLCD memory.
Example	<code>tmp = Glcd_Read_Data()</code>

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	<p>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.</p> <p>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".</p>
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	<i>' Use the custom 5x7 font "myfont" which starts with space (32):</i> Glcd_Set_Font(@myfont, 5, 7, 32)

Glcd_Write_Char

Prototype	sub procedure Glcd_Write_Char(dim character, x, page, color as byte)
Description	<p>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).</p> <p>Use routine Glcd_Set_Font to specify font, or the default 5x7 font (included with the library) will be used.</p>
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 const 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)

Glcd_Partial_Image

Prototype	sub procedure Glcd_Partial_Image(dim x1, y1, x2, y2, color as byte, dim const image as byte[1024])
Description	Displays partial bitmap image on the GLCD. Parameter <code>image</code> should be formatted as an array of 1024 bytes. Parameters <code>(x1, y1)</code> set the upper left corner, and <code>(x2, y2)</code> set the lower right corner of the clip. Parameter <code>color</code> defines the fill: 0 draws a “white” image (clear dots), 1 draws a “black” image (put dots), and 2 draws an inverted image (invert each dot). 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 <code>Glcd_Init</code> .
Example	<code>Glcd_Partial_Image(0, 0, 32, 64, 1, my_image)</code>

Library Example

The following drawing demo tests advanced routines of GLCD library.

```
program Glcd_Test
include "images" ' Pull in the file with my images

dim j, k as byte

main:
  Glcd_Init(PORTB, 2, 0, 3, 5, 7, 1, PORTD)

  ' Set font for displaying text
  Glcd_Set_Font(@FontSystem5x8, 5, 8, 32)

do

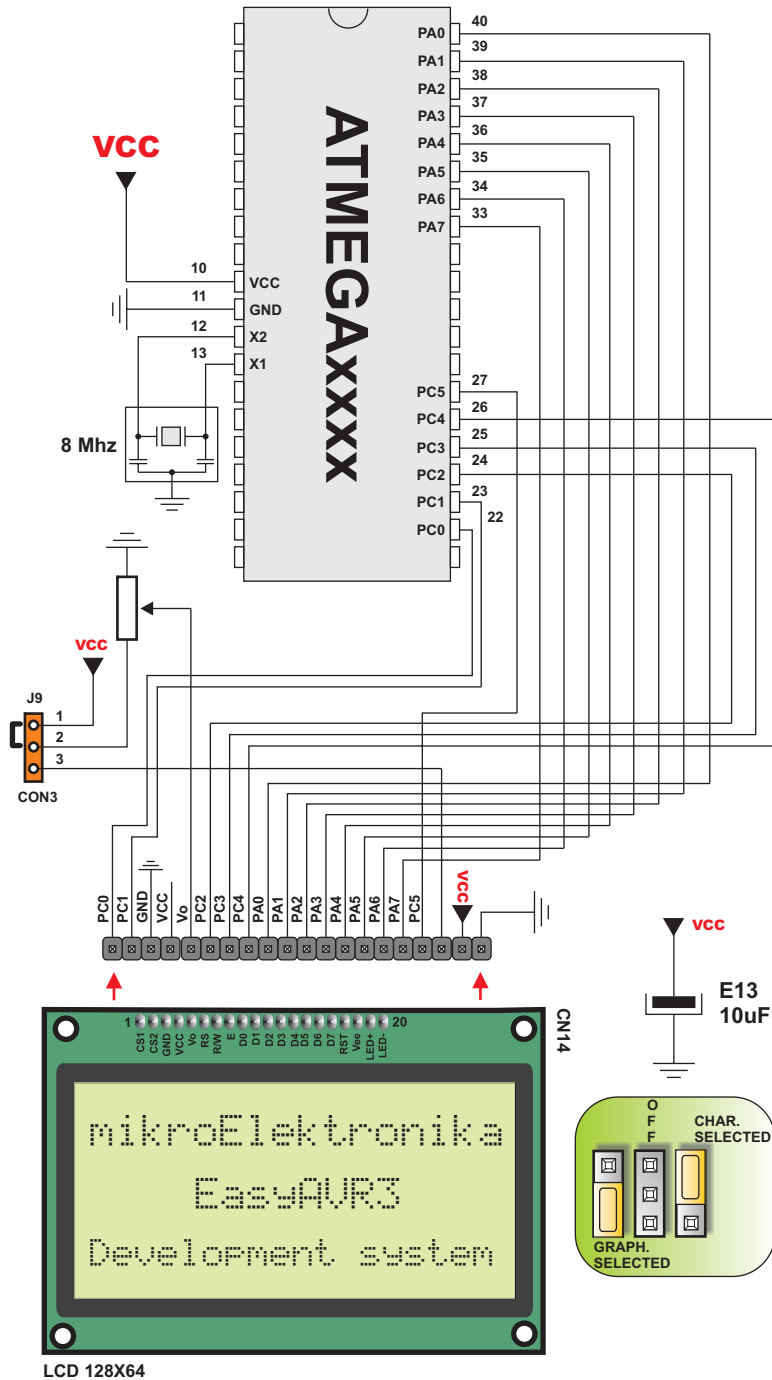
  ' Draw circles
  Glcd_Fill(0) ' Clear screen
  Glcd_Write_Text("Circles", 0, 0, 1)
  j = 4
  while j < 31
    Glcd_Circle(63, 31, j, 2)
    j = j + 4
  wend
  Delay_ms(4000)

  ' 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



Multi Media Card Library

The Multi Media Card (MMC) is a flash memory 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 PDAs.

mikroBasic provides a library for accessing data on Multi Media Card via SPI communication.

Library Routines

```
Mmc_Init  
Mmc_Read_Sector  
Mmc_Write_Sector  
Mmc_Read_Cid  
Mmc_Read_Csd  
Mmc_Fat_Init  
  
Mmc_Fat_Assign  
Mmc_Fat_Delete  
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_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 <code>port</code> and <code>pin</code> designate the CS line used in the communication (parameter <code>pin</code> should be 0..7). The function returns 0 if MMC card is present and successfully initialized, otherwise returns 1. <code>Mmc_Init</code> needs to be called before using other functions of this library.
Example	<code>error = Mmc_Init(PORTC, 2) ' Init with CS line at RC2</code>

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 <code>sector</code> . Read data is stored in the array <code>data</code> . Function returns 0 if read was successful, or 1 if an error occurred.
Requires	Library needs to be initialized, see <code>Mmc_Init</code> .
Example	<code>error = Mmc_Read_Sector(sector, data)</code>

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 0 if MMC card is present and successfully initialized, otherwise returns 1.
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.
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	<p>This function designates (“assigns”) the file we’ll be working with. The function looks for the file specified by the <code>filename</code> 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.</p> <p>Whether the new file will be created or not is controlled by the parameter <code>create_file</code> - setting it to zero will prevent creation of new file, while giving it any non-zero value will do the opposite.</p> <p>The <code>filename</code> must be 8 + 3 characters in uppercase.</p>
Requires	Library needs to be initialized; see <code>Mmc_Fat_Init</code> .
Example	<pre>' 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 always 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)</pre>

Mmc_Fat_Delete

Prototype	sub function Mmc_Fat_Delete
Returns	Deletes file from MMC.
Description	<p>Ports must be initialized for FAT operations with MMC. See <code>Mmc_Fat_Init</code>.</p> <p>File must be assigned. See <code>Mmc_Fat_Assign</code>.</p>
Example	<code>Mmc_Fat_Delete</code>

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 <code>size</code> stores the size of the assigned file, in bytes.
Requires	The file must be assigned, see <code>Mmc_Fat_Assign</code> .
Example	<code>Mmc_Fat_Reset (size)</code>

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 <code>Mmc_Fat_Assign</code> .
Example	<code>Mmc_Fat_Rewrite()</code>

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 <code>Mmc_Fat_Assign</code> .
Example	<code>Mmc_Fat_Append()</code>

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 [256])
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) Mmc_Fat_Write("Hello world")

Mmc_Fat_Set_File_Date

Prototype	sub procedure Mmc_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_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	<pre>' 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)</pre>

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	<pre>' get file size dim yr as word dim mnth, dat, hrs, mins as byte ... file_name = "MYFILEXXTXT" Mmc_Fat_Assign(file_name) cf_size = Mmc_Fat_Get_File_Size()</pre>

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)

  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: **

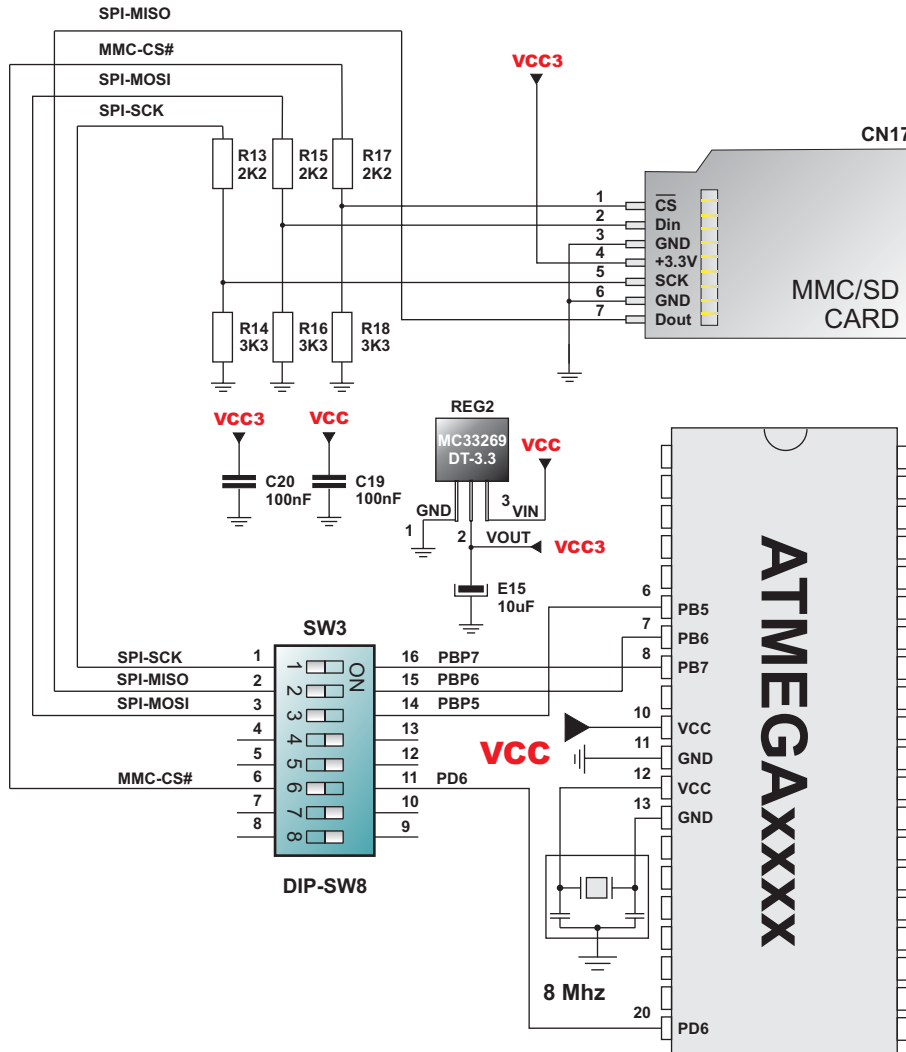
  tmp = Mmc_Read_Sector(55, data)    ' Read from sector 55

  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.
```

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 F_{osc} 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 <code>port</code> and <code>pin</code> specify the location of DS1820.
Requires	Works with Dallas DS1820 temperature sensor only.
Example	<code>Ow_Reset(PORTA, 5)</code> ' 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	<code>tmp = Ow_Read(PORTA, 5)</code>

Ow_Write

Prototype	sub procedure Ow_Write(dim byref port as byte , dim pin, par as byte)
Description	Writes one byte of data (argument <code>par</code>) via OneWire bus.
Example	<code>Ow_Write(PORTA, 5, \$CC)</code>

Library Example

The example reads the temperature from DS1820 sensor connected to RA5. Temperature value is continually displayed on LCD.

```

program onewire_test

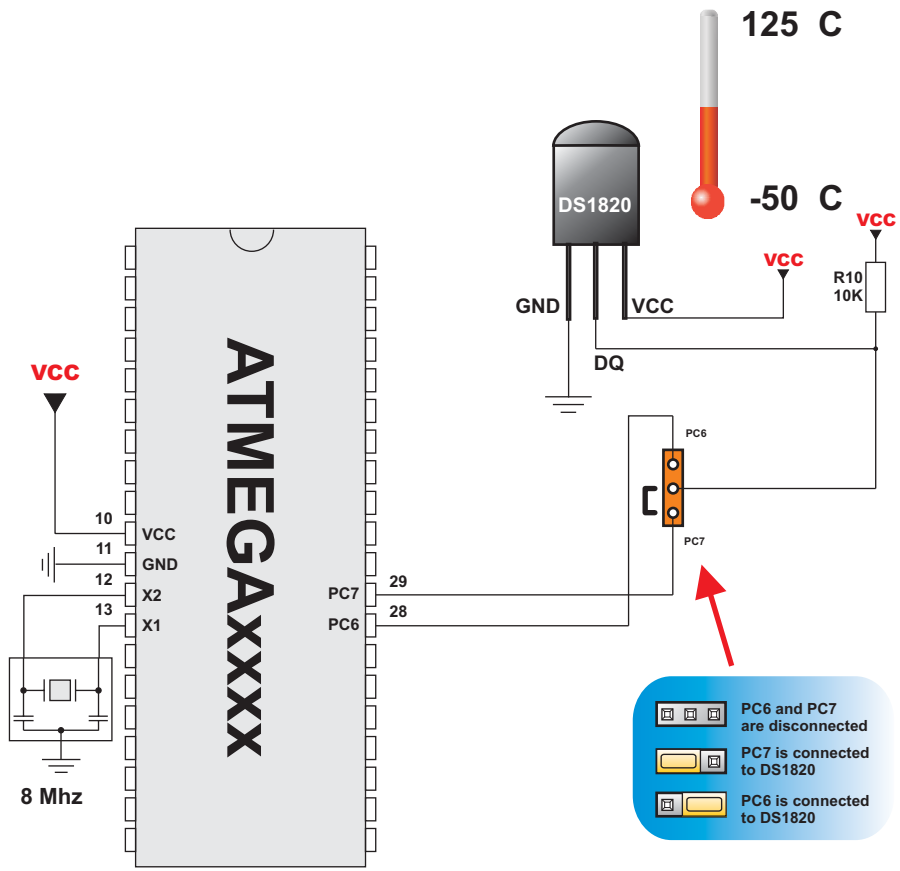
dim i, j1, j2, tmp_sign as byte
    text as char[6]

main:
    Usart1_init(9600)
    do
        ow_reset(PORTC,7)           ' onewire reset signal
        ow_write($CC)               ' issue command to DS1820
        ow_write($44)               ' issue command to DS1820
        delay_us(120)
        i = ow_reset(PORTC,7)
        ow_write($CC)               ' issue command to DS1820
        ow_write($BE)               ' issue command to DS1820
        delay_ms(200)
        j1 = ow_read                 ' get result
        j2 = ow_read                 ' get result
        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
        Usart1_write_char(tmp_sign)
        Usart1_write_char(text[0])
        Usart1_write_char(text[1])
        Usart1_write_char(46)       ' '.'
        ByteToStr(j2,text)          ' decimal
        Usart1_write_char(text[0])
        Usart1_write_char(223)      ' 'degree' character
        Usart1_write_char("C")
        Usart1_write_char(10)       ' next row
        Usart1_write_char(14)

        Delay_ms(100)
    loop until false              ' endless loop
end.
    
```

Hardware Connection



- PC6 and PC7 are disconnected
- PC7 is connected to DS1820
- PC6 is connected to DS1820

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 AVR-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_Key_Read

Ps2_Init

Prototype	sub procedure Ps2_Init(dim byref port as byte , dim clock, data as byte)
Description	Initializes port for work with PS/2 keyboard, with custom pin settings. Parameters data and clock specify pins of port for Data line and Clock line, respectively. Data and clock need to be in range 0..7 and cannot point to the same pin. You need to call Ps2_Init 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, 3, 2)

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	<p>The procedure retrieves information about key pressed.</p> <p>Parameter <code>value</code> holds the value of the key pressed. For characters, numerals, punctuation marks, and space, <code>value</code> will store the appropriate ASCII value. Procedure “recognizes” the function of Shift and Caps Lock, and behaves appropriately.</p> <p>Parameter <code>special</code> is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of these, <code>special</code> will be set to 1, otherwise 0.</p> <p>Parameter <code>pressed</code> is set to 1 if the key is pressed, and 0 if released.</p>
Requires	PS/2 keyboard needs to be initialized; see <code>Ps2_Init</code> .
Example	<pre>' 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</pre>

Library Example

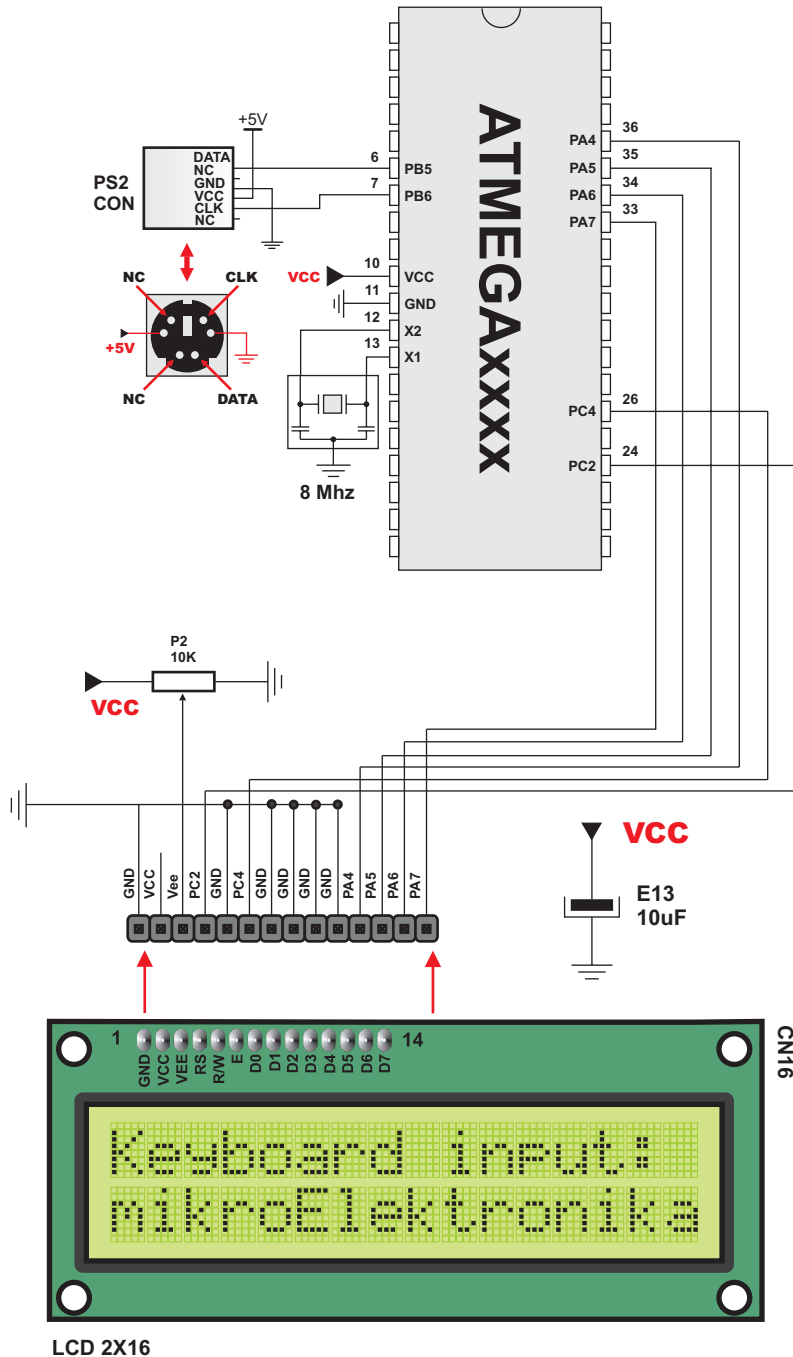
This simple example reads values of keys pressed on PS/2 keyboard and sends them via USART.

```
program ps2_test

dim keydata,
    special,
    down      as byte

main:
    keydata = 0
    special = 0
    down    = 0
    Ps2_Init(PORTC,0,1)
    Delay_ms(100)
    Usart1_Init(9600)
    Delay_ms(100)
    Usart1_Write_Text("You can type now:")
    while true
        if Ps2_Key_Read(keydata, special, down)=1 then
            if (down<>0) and (keydata = 16) then
                Usart1_Write_Char(keydata)
            else
                if (down<>0) and (keydata = 13) then
                    Usart1_Write_Char(13)
                    Usart1_Write_Char(10)
                else
                    if (down<>0) and (special=0) and (keydata<>0) then
                        Usart1_Write_Char(keydata)
                    end if
                end if
            end if
            Delay_ms(50)
        end if
    end if
wend
end.
```

Hardware Connection



PWM Library

CCP module is available with a number of AVRmicros. mikroBasic provides library which simplifies using PWM HW Module.

Note: These routines support module on PB3, and won't work with modules on other ports. You can find examples for AVRmicros with module on other ports in mikroBasic installation folder, subfolder "Examples". Also, mikroBasic does not support enhanced PWM modules.

Library Routines

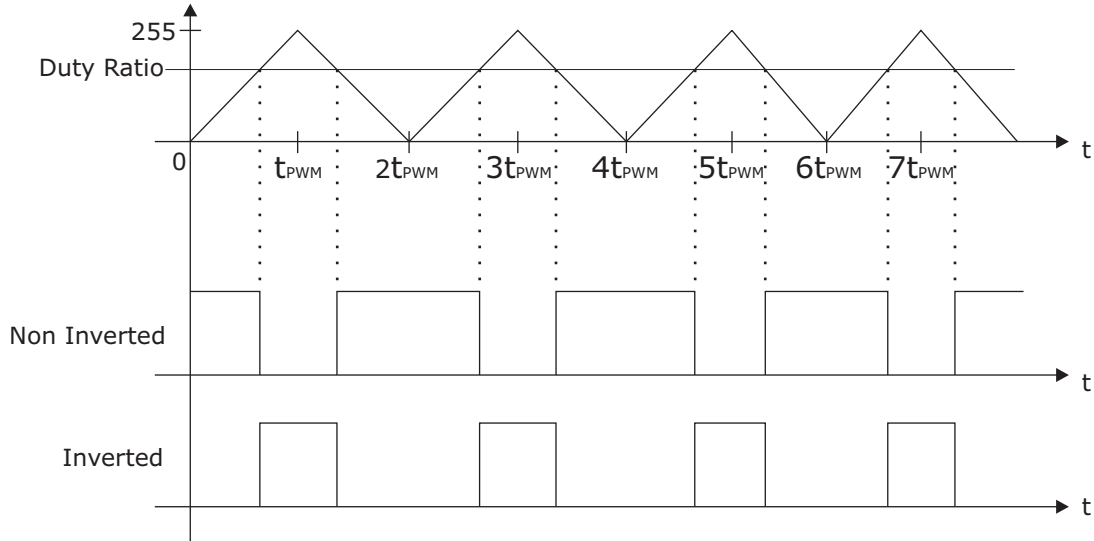
Pwm_Init
Pwm_Change_Duty
Pwm_Start
Pwm_Stop

Pwm_Init

Prototype	sub procedure Pwm_Init(dim wave_mode, prescaler, inverted , duty as byte)
Description	Initializes the PWM module. Parameter wave_mode is a desired PWM mode. There are two modes: Phase and Fast. Parameter prescaler chooses prescale value (N=1,8,64,256 or 1024). inverted parameter is for choosing between inverted and non inverted PWM signal. duty parameter sets duty ratio from 0 to 255. PWM signal graphs and formulas are shown on next page.
Requires	You need a CMO module on PORTB to use this library. Check mikroBasic installation folder, subfolder "Examples", for alternate solutions.
Example	Initialize PWM module: Pwm_Init (PWM_PHASE_CORRECT_MODE, PWM_PRESCALER_1024, PWM_NON_INVERTED, duty)

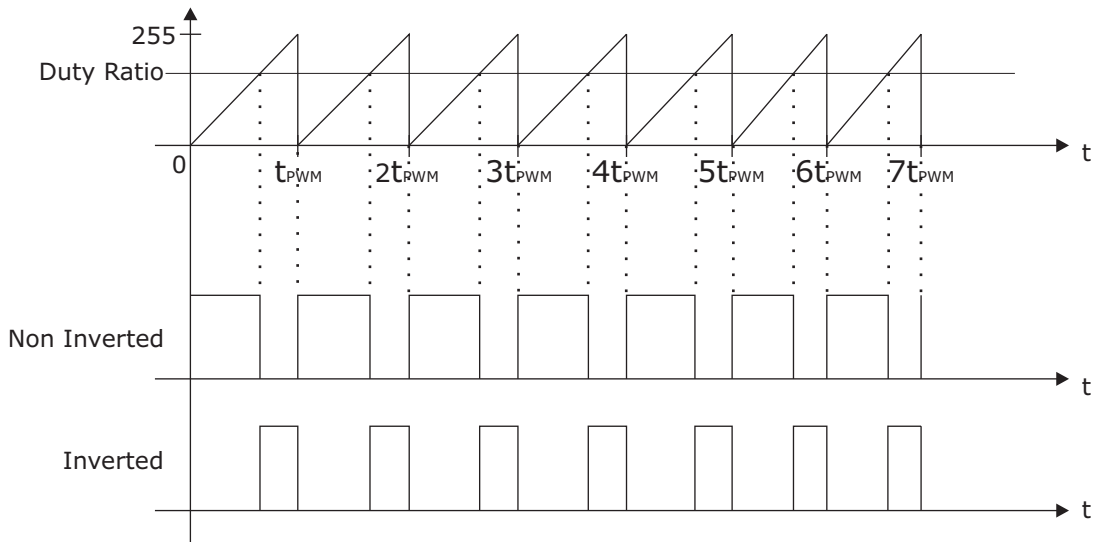
PHASE MODE

$$f_{pwm} = \frac{f_{clk\ i/o}}{N \cdot 510}$$



FAST MODE

$$f_{pwm} = \frac{f_{clk\ i/o}}{N \cdot 256}$$



Pwm_Change_Duty

Prototype	sub procedure Pwm_Change_Duty(dim duty_ratio as byte)
Description	Changes PWM duty ratio. Parameter <code>duty_ratio</code> 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 $(\text{Percent} * 255) / 100$.
Requires	You need a CMO module on PORTB to use this library. Check mikroBasic installation folder, subfolder "Examples", for alternate solutions.
Example	<code>Pwm_Change_Duty(192) // Set duty ratio to 75%</code>

Pwm_Start

Prototype	sub procedure Pwm_Start
Description	Starts PWM. It is not necessary to call Pwm_Start after Pwm_Init.
Requires	You need a CMO module on PORTB to use this library. Check mikroBasic installation folder, subfolder "Examples", for alternate solutions.
Example	<code>Pwm_Start()</code>

Pwm_Stop

Prototype	procedure Pwm_Stop
Description	Stops PWM.
Requires	You need a CMO module on PORTB to use this library. Check mikroBasic installation folder, subfolder "Examples", for alternate solutions.
Example	<code>Pwm_Stop()</code>

Library Example

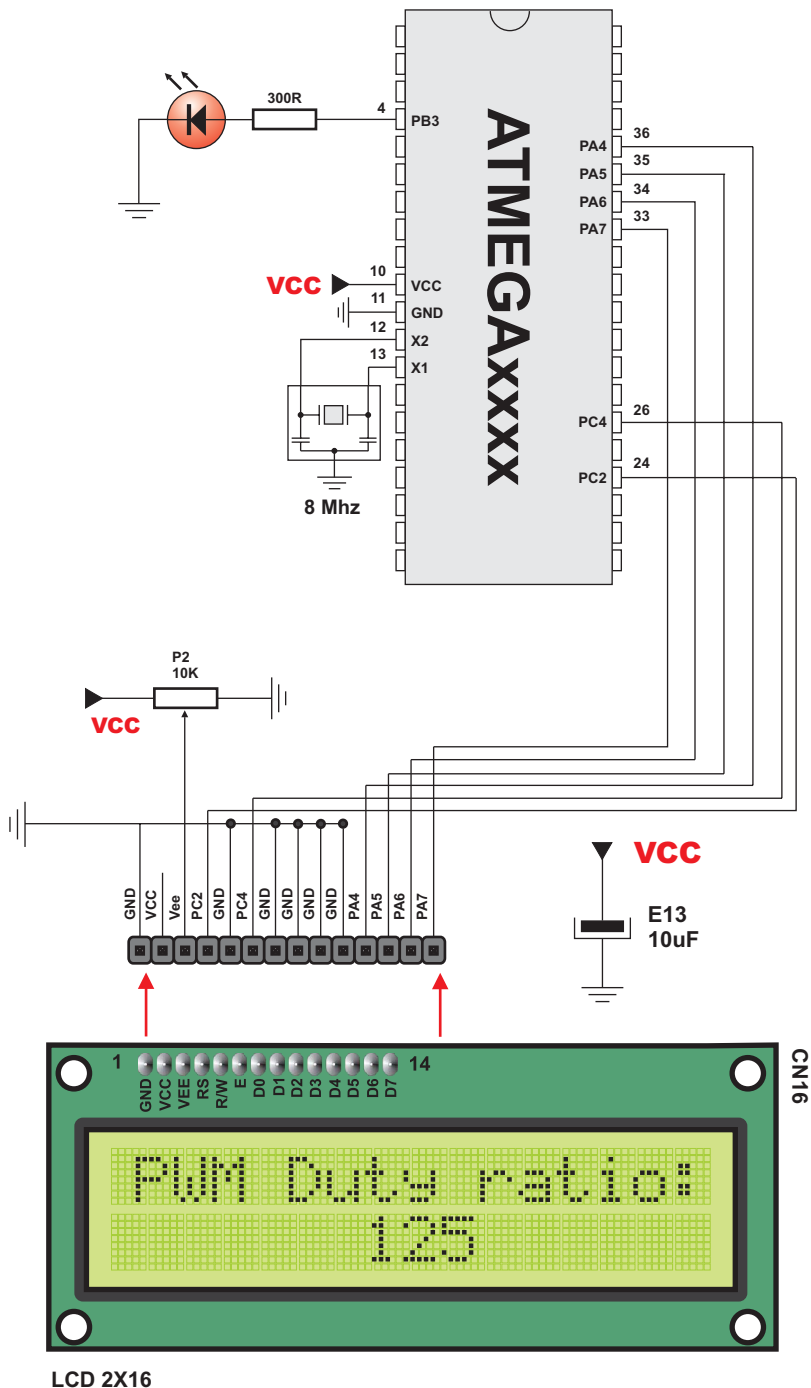
The example changes PWM duty ratio on pin PB3 continually. If LED is connected to PB3, you can observe the gradual change of emitted light.

```
program Pwm_Test

dim text as string[7]
    duty as byte

main:
    Lcd_Init(PORTC, 4, 2, PORTA, LCD_HI_NIBBLE)
    Lcd_Cmd(LCD_CURSOR_OFF)
    Lcd_Out(1,1,"PWM Duty ratio:") ' PWM on OC0 pin = PORTB.3 on mega16
    duty = 20
    Pwm_Init(PWM_PHASE_CORRECT_MODE, PWM_PRESCALER_1024, PWM_NON_INVERTED, duty)
    while true
        ByteToStr(duty, text)
        Lcd_Out(2,7, text)
        Delay_ms(100)
        duty = duty + 1
        Pwm_Set_Duty(duty)
    wend
end.
```

Hardware Connection



Secure Digital Library

Secure Digital (SD) is a flash memory memory card standard, based on the older Multi Media Card (MMC) format. SD cards are currently available in sizes up to and including 2 GB, and are used in cell phones, mp3 players, digital cameras, and PDAs.

mikroBasic provides a library for accessing data on SD Card via SPI communication.

Library Routines

```
Sd_Init
Sd_Read_Sector
Sd_Write_Sector
Sd_Read_Cid
Sd_Read_Csd

Sd_Fat_Init
Sd_Fat_Assign
Sd_Fat_Delete
Sd_Fat_Reset
Sd_Fat_Read
Sd_Fat_Rewrite
Sd_Fat_Append
Sd_Fat_Write
Sd_Fat_Set_File_Date
Sd_Fat_Get_File_Date
Sd_Fat_Get_File_Size
```

Sd_Init

Prototype	sub function Sd_Init(dim byref port as byte, dim pin as byte) as byte
Returns	Returns 0 if SD 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 SD card is present and successfully initialized, otherwise returns 1. Sd_Init needs to be called before using other functions of this library.
Example	<code>error = Sd_Init(PORTC, 2) ' Init with CS line at RC2</code>

Sd_Read_Sector

Prototype	<code>sub function Sd_Read_Sector(dim sector as longint, dim byref data as byte[512]) as byte</code>
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads one sector (512 bytes) from SD card at sector address <code>sector</code> . Read data is stored in the array <code>data</code> . Function returns 0 if read was successful, or 1 if an error occurred.
Requires	Library needs to be initialized, see <code>Sd_Init</code> .
Example	<code>error = Sd_Read_Sector(sector, data)</code>

Sd_Write_Sector

Prototype	<code>sub function Sd_Write_Sector(dim sector as longint, dim byref data as byte[512]) as byte</code>
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 SD card at sector address <code>sector</code> . 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 <code>Sd_Init</code> .
Example	<code>error = Sd_Write_Sector(sector, data)</code>

Sd_Read_Cid

Prototype	sub function Sd_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 Sd_Init.
Example	error = Sd_Read_Cid(data)

Sd_Read_Csd

Prototype	sub function Sd_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 Sd_Init.
Example	error = Sd_Read_Csd(data)

Sd_Fat_Init

Prototype	sub function Sd_Fat_Init(dim byref sdport as byte , dim sdpin as byte) as byte
Returns	Returns 0 if SD card is present and successfully initialized, otherwise returns 1.
Description	Initializes ports appropriately for FAT operations with SD card. Specify SD port, and SD pin.
Example	success = Sd_Fat_Init(PORTC, 2)

Sd_Fat_Assign

Prototype	sub function Sd_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	<p>This function designates (“assigns”) the file we’ll be working with. The function looks for the file specified by the <code>filename</code> 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.</p> <p>Whether the new file will be created or not is controlled by the parameter <code>create_file</code> - setting it to zero will prevent creation of new file, while giving it any non-zero value will do the opposite.</p> <p>The <code>filename</code> must be 8 + 3 characters in uppercase.</p>
Requires	Library needs to be initialized; see <code>Sd_Fat_Init</code> .
Example	<pre>' Assign the file "EXAMPLE1.TXT" in the root directory of SD. ' If the file is not found, routine will create one. ' In this case, function return value will always be non-zero Sd_Fat_Assign("EXAMPLE1TXT", 1) ' Assign the file "EXAMPLE2.TXT" in the root directory of SD. ' If the file is not found, routine will NOT create new one. file_found = Sd_Fat_Assign("EXAMPLE2TXT", 0)</pre>

Sd_Fat_Delete

Prototype	sub function Sd_Fat_Delete
Returns	Deletes file from SD.
Description	<p>Ports must be initialized for FAT operations with SD. See <code>Sd_Fat_Init</code>.</p> <p>File must be assigned. See <code>Sd_Fat_Assign</code>.</p>
Example	<code>Sd_Fat_Delete</code>

Sd_Fat_Reset

Prototype	<code>sub procedure Sd_Fat_Reset(dim byref size as longint)</code>
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 <code>size</code> stores the size of the assigned file, in bytes.
Requires	The file must be assigned, see <code>Sd_Fat_Assign</code> .
Example	<code>Sd_Fat_Reset(size)</code>

Sd_Fat_Rewrite

Prototype	<code>sub procedure Sd_Fat_Rewrite</code>
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 <code>Sd_Fat_Assign</code> .
Example	<code>Sd_Fat_Rewrite()</code>

Sd_Fat_Append

Prototype	<code>sub procedure Sd_Fat_Append</code>
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 <code>Sd_Fat_Assign</code> .
Example	<code>Sd_Fat_Append()</code>

Sd_Fat_Read

Prototype	<code>sub procedure Sd_Fat_Read(dim byref data as byte)</code>
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 Sd_Fat_Read.
Requires	The file must be assigned, see Sd_Fat_Assign. Also, file pointer must be initialized; see Sd_Fat_Reset.
Example	<code>Sd_Fat_Read(mydata)</code>

Sd_Fat_Write

Prototype	<code>sub procedure Sd_Fat_Write(dim byref fdata as char[256])</code>
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 Sd_Fat_Assign. Also, file pointer must be initialized; see Sd_Fat_Append or Sd_Fat_Rewrite.
Example	<code>Sd_Fat_Write(txt)</code> <code>Sd_Fat_Write("Hello world")</code>

Sd_Fat_Set_File_Date

Prototype	<code>sub procedure Sd_Set_File_Date(dim year as word, dim month, day, hours, min, sec as byte)</code>
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 Sd_Fat_Assign and Sd_Fat_Reset.
Example	<code>' April 1st 2005, 18:07:00</code> <code>Sd_Set_File_Date(2005, 4, 1, 18, 7, 0)</code>

Sd_Fat_Get_File_Date

Prototype	sub procedure Sd_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 Sd_Fat_Assign.
Example	<pre>' get Date/time of file dim yr as word dim mnth, dat, hrs, mins as byte ... file_Name = "MYFILEABTXT" Sd_Fat_Assign(file_Name) Sd_Fat_Get_File_Date(yr, mnth, dat, hrs, mins)</pre>

Sd_Fat_Get_File_Size

Prototype	sub function Sd_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 Sd_Fat_Assign.
Example	<pre>' get file size dim yr as word dim mnth, dat, hrs, mins as byte ... file_name = "MYFILEXXTXT" Sd_Fat_Assign(file_name) cf_size = Sd_Fat_Get_File_Size()</pre>

Library Example

The following code tests SD 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 sd_test
dim tmp as byte
dim i as word
dim data as byte[512]

main:
  Usart_Init(9600)

  tmp = Sd_Init(PORTD, 6)

  ' Fill the buffer with the "M" character
  for i = 0 to 512
    data[i] = "M"
  next i

  tmp = Sd_Write_Sector(55, data)      ' Write it to SD card, sector 55

  ' Fill the buffer with the "E" character
  for i = 0 to 512
    data[i] = "E"
  next i

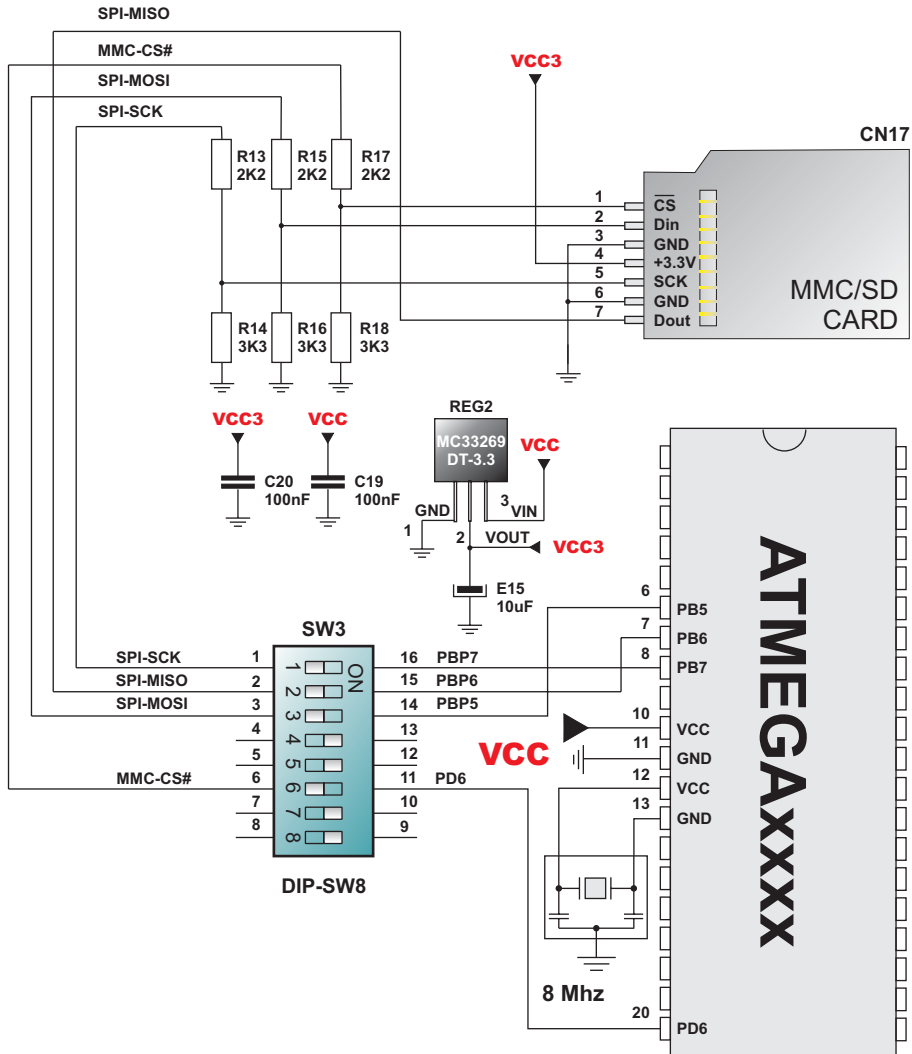
  tmp = Sd_Write_Sector(56, data)      ' Write it to SD card, sector 56
  tmp = Sd_Read_Sector(55, data)       ' Read from sector 55

  ' Send 512 bytes from buffer to USART
  if tmp = 0 then
    for i = 0 to 512
      Usart_Write(data[i])
    next i
  end if

  tmp = Sd_Read_Sector(56, data)       ' Read from sector 56

  ' Send 512 bytes from buffer to USART
  if tmp = 0 then
    for i = 0 to 512
      Usart_Write(data[i])
    next i
  end if
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 I ² C. 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 I ² C 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 I ² C 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. AVR 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 TWI Library.

```

program soft_i2c_test

dim ee_adr, ee_data as byte

main:
  ' Initialize full master mode
  Soft_I2c_Init(PORTD, 3, 4)
  DDRC = $FF           ' PORTC is output
  PORTC = $FF         ' Initialize PORTC
  Soft_I2c_Start()    ' Issue I2C signal: start
  Soft_I2C_Write($A2) ' Send byte via I2C (command to 24c02)
  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

  Delay_ms(100)       ' Pause while EEPROM writes data

  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
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, 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 <code>buffer</code> and receives data.
Requires	Soft SPI must be initialized and communication established before using this function. See <code>Soft_Spi_Config</code> .
Example	<code>tmp = Soft_Spi_Read(buffer)</code>

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 <code>Soft_Spi_Config</code> .
Example	<code>Soft_Spi_Write(1)</code>

Library Example

The example demonstrates using Software SPI library. Assumed HW configuration is: MCP4921 (chip select pin) connected to PD5, and SDO, SDI, SCK pins are connected to corresponding pins of MCP4921. Hardware connection is given on SPI section.

```
program Soft_SPI_test

dim value as word

sub procedure Dac_Output(dim out_value as word)
dim loc_ as byte
    PORTD.5=0      ' clear CS
    loc_ = out_value >> 8
    loc_ = loc_ or $30
    Soft_Spi_Write(loc_)
    Delay_us(50)
    loc_ = out_value and $FF
    Soft_Spi_Write(loc_)
    Delay_us(50)
    PORTD.5 = 1    ' set CS
end sub

main:
    DDRD.5 = 1     ' set direction of CS line to be output.
    Soft_Spi_Init(PORTB, 6, 5, 7, SOFT_SPI_PRESCALER_256,
    SOFT_SPI_LO_2_HI_IDLE_LO, SOFT_SPI_MODE_8)
    while TRUE
        value = 1
        while value < $FFF
            Dac_Output(value)    ' changing the voltage from zero to AREF.
            Delay_ms(5)
            inc(value)
        wend
    wend
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 <code>Soft_Uart_Init(dim byref port as byte, dim rx, tx, baud_rate, inverted as byte)</code>
Description	<p>Initializes software UART. Parameter <code>port</code> specifies port of MCU on which RX and TX pins are located; parameters <code>rx</code> and <code>tx</code> need to be in range 0–7 and cannot point at the same pin; <code>baud_rate</code> is the desired baud rate. Maximum baud rate depends on AVR's clock and working conditions.</p> <p>Parameter <code>inverted</code>, if set to non-zero value, indicates inverted logic on output.</p> <p><code>Soft_Uart_Init</code> needs to be called before using other functions from Soft UART Library.</p>
Example	<p>This will initialize software UART and establish the communication at 9600 bps:</p> <pre>Soft_Uart_Init(PORTB, 1, 2, 9600, 0)</pre>

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 <code>error</code> will be zero if the transfer was successful. This is a non-blocking function call, so you should test the <code>error</code> manually (check the example below).
Requires	Soft UART must be initialized and communication established before using this function. See <code>Soft_Uart_Init</code> .
Example	<pre>' Here's a loop which holds until data is received: error = 1 do data = Soft_Uart_Read(error) loop until error = 0</pre>

Soft_Uart_Write

Prototype	sub procedure Soft_Uart_Write(dim data as byte)
Description	Function transmits a byte (<code>data</code>) via UART.
Requires	<p>Soft UART must be initialized and communication established before using this function. See <code>Soft_Uart_Init</code>.</p> <p>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.</p>
Example	<code>Soft_Uart_Write(\$0A)</code>

Library Example

The example demonstrates simple data exchange via software UART. When AVR MCU receives data, it immediately sends the same data back. If AVR is connected to the PC (see the figure below), you can test the example from mikroBasic terminal for RS232 communication, menu choice **Tools > Terminal**. Hardware connection is given in USART library section.

```
program SOFT_UART_TEST

dim i, err_ as byte

main:
  Soft_Uart_Init(PORTD, 0, 1, 19200, SOFT_UART_NORMAL)
  Soft_Uart_Write_Text("mikroElektronika")
  while true
    i = Soft_Uart_Read(err_)           ' read the received data
    Soft_Uart_Write(i)                ' send data via USART
  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_Play_Khz

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 PB2

Sound_Play

Prototype	sub procedure Sound_Play(dim freq_in_hz, period_ms 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 before using this function.
Example	If you want to play sound of 1234Hz and play it for 250 ms do like this: Sound_Play(1234, 250)

Sound_Play_Khz

Prototype	sub procedure Sound_Play_Khz(dim freq_in_khz, period_ms as word)
Description	Plays the sound at the specified port and pin (see Sound_Init). Parameter freq_in_khz is a sound frequency given in Khz, and generated sound lasts for a specified period in miliseconds (period_ms).
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 before using this function.
Example	If you want to play sound of 1KHz, and play it for 150 ms do like this: Sound_Play_Khz(1, 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. Sound frequencies in this example are generated by pressing buttons connected on pins PB4, PB5, PB6 or PB7.

```
program Sound_Test

sub procedure Tone1
    Sound_Play(6000, 400)      ' frequency is 6kHz and duration is 400ms
end sub

sub procedure Tone2
    Sound_Play(7000, 400)      ' frequency is 7kHz and duration is 400ms
end sub

sub procedure Tone3
    Sound_Play(8000, 400)      ' frequency is 8kHz and duration is 400ms
end sub
```

// continues on next page...

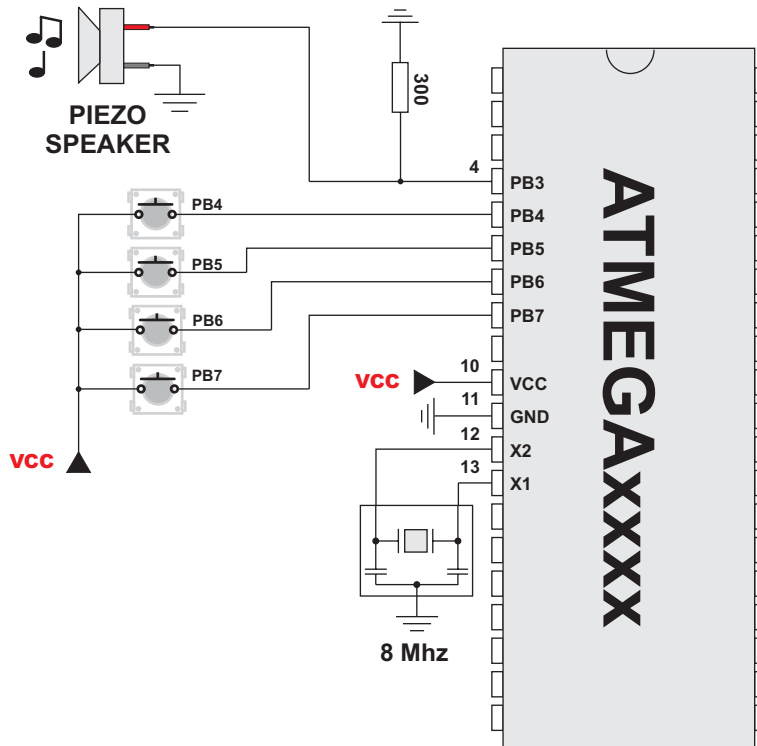
```
//continues here...
```

```
sub procedure Melody           ' Plays the melody "Yellow house"
  Tone1  Tone2  Tone3  Tone3
  Tone1  Tone2  Tone3  Tone3
  Tone1  Tone2  Tone3  Tone1
  Tone2  Tone3  Tone3  Tone1
  Tone2  Tone3  Tone3  Tone3
  Tone2  Tone1
end sub

main:
  Sound_Init(PORTC,3)          ' put piezo on portc.3
  Sound_Play_Khz(3, 200)

  while true
    if Button(PORTB,7,1,1) then  ' RB7 plays Tone1
      Tone1
    end if
    while PINB.7 = 1
      nop                        ' Wait for button to be released
    wend
    if Button(PORTB,6,1,1) then  ' RB6 plays Tone2
      Tone2
    end if
    while PINB.6 = 1
      nop                        ' Wait for button to be released
    wend
    if Button(PORTB,4,1,1) then  ' RB4 plays Melody
      Melody
    end if
    while PINB.4 = 1
      nop                        ' Wait for button to be released
    wend
  wend
end.
```

Hardware Connection



SPI Library

SPI module is available with a number of AVR MCU models. mikroBasic provides a library for initializing Slave mode and comfortable work with Master mode. AVR can easily communicate with other devices via SPI: A/D converters, D/A converters, LTC1290, etc. You need AVR MCU with hardware integrated SPI

Note: This library supports module on PORTB or PORTC, and will not work with modules on other ports. Examples for AVRmicros 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	<p>Configures and initializes SPI with default settings. Spi_Init_Advanced or Spi_Init needs to be called before using other functions from SPI Library.</p> <p>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.</p> <p>For custom configuration, use Spi_Init_Advanced.</p>
Requires	You need AVR 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	<p>Configures and initializes SPI. Spi_Init_Advanced or Spi_Init needs to be called before using other functions of SPI Library.</p> <p>Parameter mast_slav determines the work mode for SPI; can have the values:</p> <pre> 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 </pre> <p>The data_sample determines when data is sampled; can have the values:</p> <pre> DATA_SAMPLE_MIDDLE ' Input data sampled in middle of interval DATA_SAMPLE_END ' Input data sampled at the end of interval </pre> <p>Parameter clock_idle determines idle state for clock; can have the following values:</p> <pre> CLK_IDLE_HIGH ' Clock idle HIGH CLK_IDLE_LOW ' Clock idle LOW </pre> <p>Parameter transmit_edge can have the following values:</p> <pre> LOW_2_HIGH ' Data transmit on low to high edge HIGH_2_LOW ' Data transmit on high to low edge </pre>
Requires	You need AVR MCU with hardware integrated SPI.
Example	<p>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:</p> <pre> Spi_Init_Advanced(MASTER_OSC_DIV4, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH) </pre>

Spi_Read

Prototype	sub function Spi_Read(dim buffer as byte) as byte
Returns	Returns the received data.
Description	Provides clock by sending <code>buffer</code> and receives data at the end of period.
Requires	SPI must be initialized and communication established before using this function. See <code>Spi_Init_Advanced</code> or <code>Spi_Init</code> .
Example	<code>take = Spi_Read(buffer)</code>

Spi_Write

Prototype	sub procedure Spi_Write(dim data as byte) as byte
Description	Writes <code>byte data</code> to SSPBUF, and immediately starts the transmission.
Requires	SPI must be initialized and communication established before using this function. See <code>Spi_Init_Advanced</code> or <code>Spi_Init</code> .
Example	<code>Spi_Write(1)</code>

Library Example

The code demonstrates how to use SPI library procedures and functions. Assumed HW configuration is: MCP4921 (chip select pin) connected to PD5, and SDO, SDI, SCK pins are connected to corresponding pins of MCP4921.

```
program SPI_DAC_Test

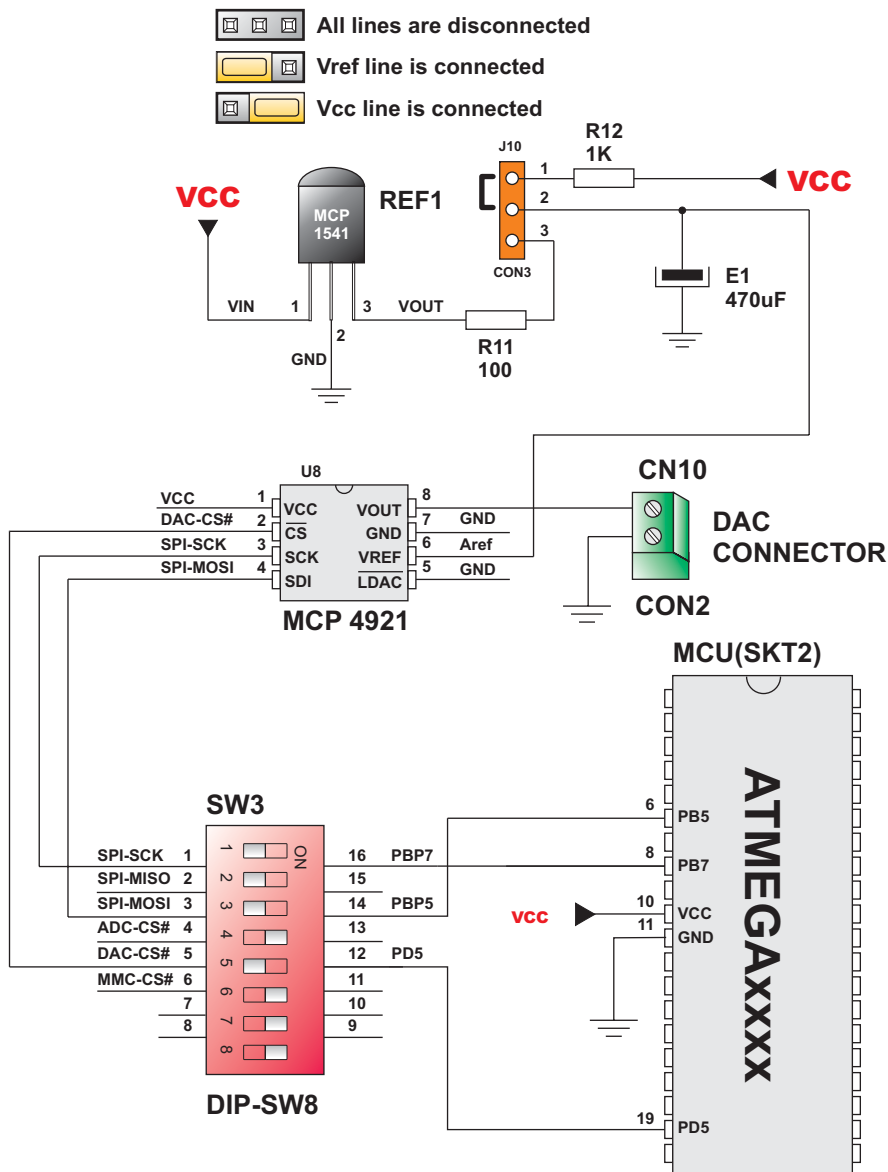
dim value as word

sub procedure Dac_Output(dim out_value as word)
dim loc_ as byte
PORTD.5=0      ' clear CS
loc_ = out_value >> 8
loc_ = loc_ or $30
Spi1_Write(loc_)
Delay_us(50)
loc_ = out_value and $FF
Spi1_Write(loc_)
Delay_us(50)
PORTD.5 = 1    ' set CS
end sub

main:
DDR.D.5 = 1    ' set direction of CS line to be output.
Spi1_Init

while TRUE
value = 1
while value < $FFF
Dac_Output(value)
Delay_ms(10)
value=value+1
wend
wend
end.
```

HW Connection



USART Library

USART hardware module is available with a number of AVRmicros. 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 AVR MCU with hardware integrated USART.

Note: USART library functions support module on PORTB, PORTC, or PORTG, and will not work with modules on other ports. Examples for AVRmicros with module on other ports can be found in “Examples” in mikroBasic installation folder.

Usart_Init

Library Routines

Usart_Data_Ready
 Usart_Read
 Usart_Write

Note: Certain AVRmicros 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 AVR 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	<pre>' If data is ready, read it: if Usart_Data_Ready = 1 then receive = Usart_Read end if</pre>

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	<pre>' If data is ready, read it: if Usart_Data_Ready = 1 then receive = Usart_Read end if</pre>

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 a simple data exchange via USART. When AVR MCU receives data, it immediately sends it back. If AVR is connected to the PC (see the figure below), you can test the example from the mikroBasic terminal for RS-232 communication, menu choice Tools > Terminal.

```

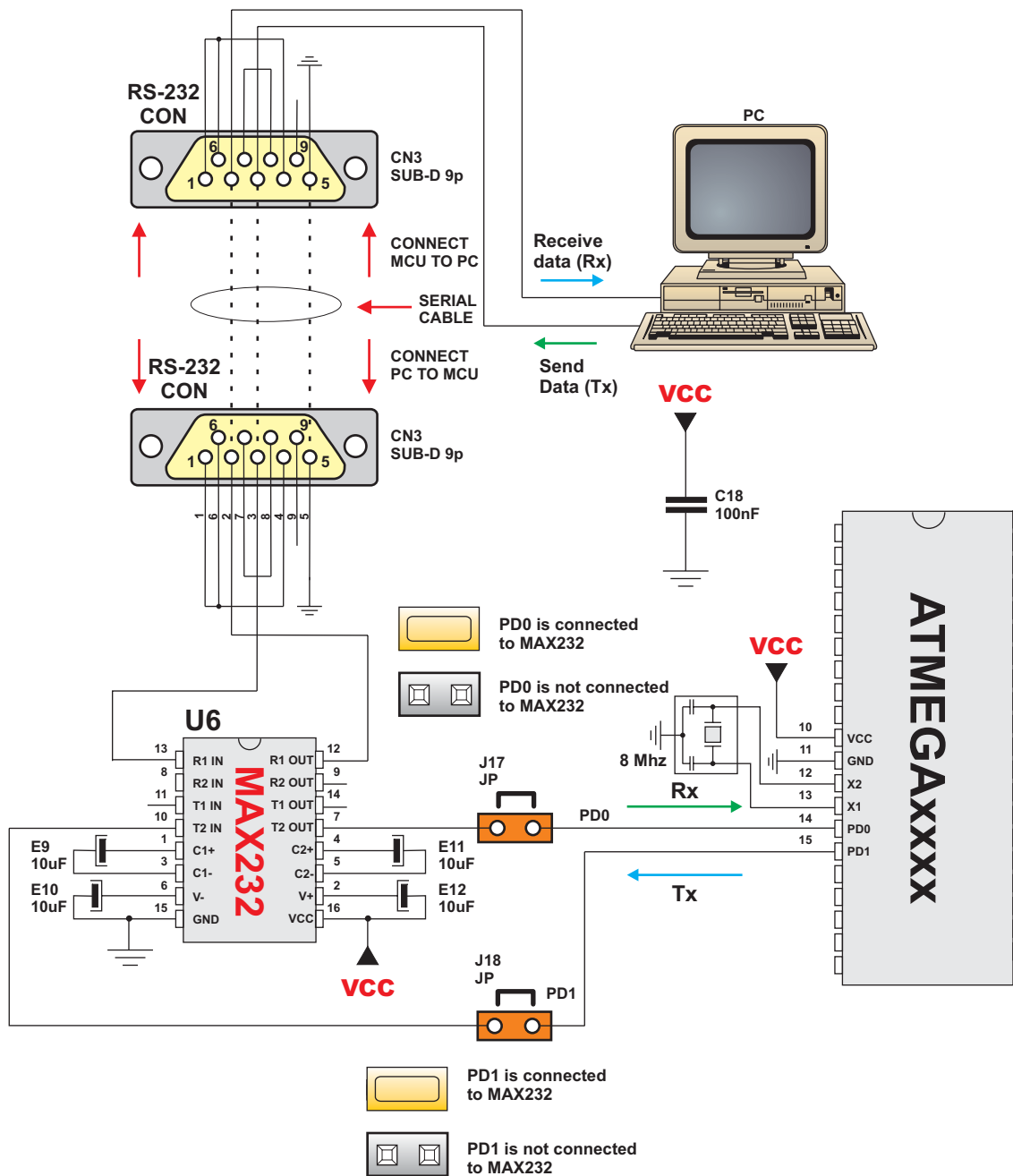
program USART_Test

dim data as byte

main:
  Usart1_Init(9600)
  while true
    while USart1_Data_Ready = 0
      nop
    wend
    data = USart1_Read
    USart1_Write_Char(data)
  wend
end.

```

Hardware 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	<p>Function eliminates the influence of contact flickering upon pressing a button (debouncing).</p> <p>Parameter <code>port</code> specifies the location of the button; parameter <code>pin</code> is the pin number on designated <code>port</code> and goes from 0..7; parameter <code>time</code> is a debounce period in milliseconds; parameter <code>active_state</code> can be either 0 or 1, and it determines if the button is active upon logical zero or logical one.</p>
Example	<p>Example reads PB0, to which the button is connected; on transition from 1 to 0 (release of button), PORTD is inverted:</p> <pre> 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 </pre>

Conversions Library

mikroBasic Conversions Library provides routines for converting numerals to strings, and routines for BCD/decimal conversions.

Library Routines

You can get text representation of numerical value by passing it to one of the following routines:

```
ByteToStr
ShortToStr
WordToStr
IntToStr
LongIntToStr
```

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	<pre>dim t as word dim txt as string[3] '... t = 24 ByteToStr(t, txt) ' txt is " 24" (one blank here)</pre>

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	<pre> dim t as short dim txt as string[4]; '... t = -24 ShortToStr(t, txt) ' txt is " -24" (one blank here) </pre>

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	<pre> dim t as word dim txt as string[5] '... t = 437 WordToStr(t, txt) ' txt is " 437" (two blanks here) </pre>

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	<pre> dim j as integer dim txt as string[6] '... j = -4220 IntToStr(j, txt) ' txt is " -4220" (one blank here) </pre>

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	<pre> dim jj as longint dim txt as string[11] '... jj = -3700000 LongintToStr(jj, txt) ' txt is " -3700000" (three blanks here) </pre>

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) ' <i>b equals 52</i>

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) ' <i>b equals \$52</i>

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

Delay 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(dim const time_in_us as longint)
Returns	Nothing.
Description	Creates a software delay in duration of time_in_us microseconds (a constant). Range of applicable constants depends on the oscillator frequency. Maximum 4,500,000 cycles. This is an “inline” routine; code is generated in the place of the call.
Example	Delay_us(10) // Ten microseconds pause

Delay_ms

Prototype	sub procedure Delay_ms(dim const time_in_ms as word)
Returns	Nothing.
Description	Creates a software delay in duration of time_in_ms milliseconds (a constant). Range of applicable constants depends on the oscillator frequency. Maximum 4,500,000 cycles. This is an “inline” routine; code is generated in the place of the call.
Example	Delay_ms(1000) // One second pause

Delay_Cyc

Prototype	sub procedure Delay_Cyc(dim number_in_cycles as longint);
Returns	Nothing.
Description	Creates a delay based on MCU clock. Maximum 4,500,000 cycles.
Example	Delay_Cyc(10)

String Library

The String Library provides a number of routines for string handling.

Library Routines

Length
Compare
Concat

Length

Prototype	<code>sub function Length(dim byref text as string[50]) as byte</code>
Description	Function returns number of characters in text. End character 0 does not count against string's length.

Compare

Prototype	<code>sub function Compare(dim byref a, b as string[50]) as byte</code>
Description	Function compares strings a and b, character-by-character. If all characters match, function returns TRUE, otherwise it returns FALSE. Algorithm for character matching is case sensitive.

Concat

Prototype	sub function Concat (dim byref a, b as string [30], dim byref sum as string [60])
Description	Function appends string b to string a (concatenates 2 strings) and returns the result. Newly created string has length equal to (Length(a) + Length(b)).

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@mikroelektronika.co.yu

Web: www.mikroelektronika.co.yu