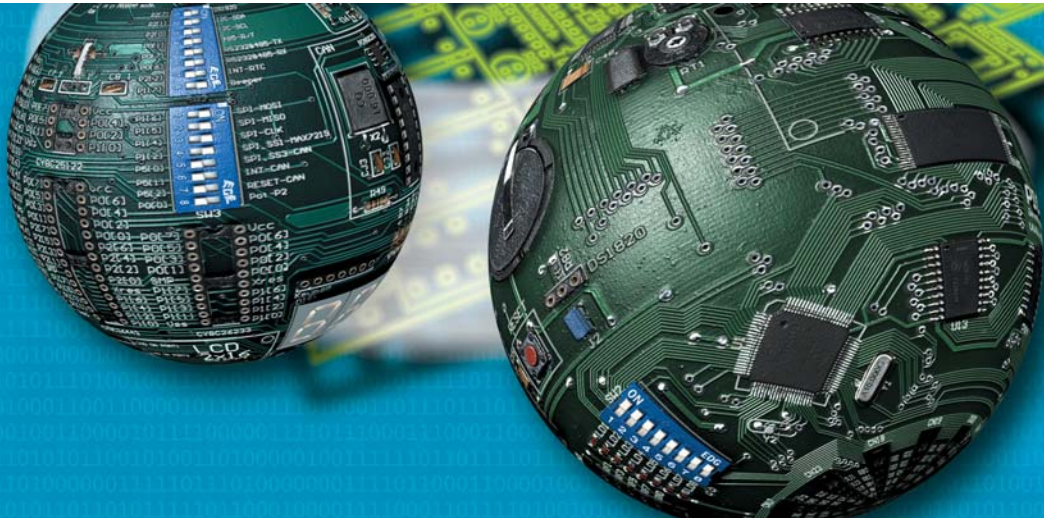


PASCAL COMPILER FOR ATMEL AVR MICROCONTROLLERS

mikroPASCAL

FOR AVR

Making it simple



Develop your applications quickly and easily with the world's most intuitive Pascal 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, mikroPascal makes a fast and reliable tool, which can satisfy needs of experienced engineers and beginners alike.

user's manual

Reader's note**DISCLAIMER:**

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

Copyright © mikroElektronika 2003 - 2006.

This manual covers mikroPascal version 4 and the related topics. New versions may contain changes without prior notice.

COMPILER BUG REPORTS:

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

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

CONTACT US:

mikroElektronika

Voice: + 381 (11) 30 66 377, + 381 (11) 30 66 378

Fax: + 381 (11) 30 66 379

Web: www.mikroe.com

E-mail: office@mikroe.com

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.

mikroPascal User's manual

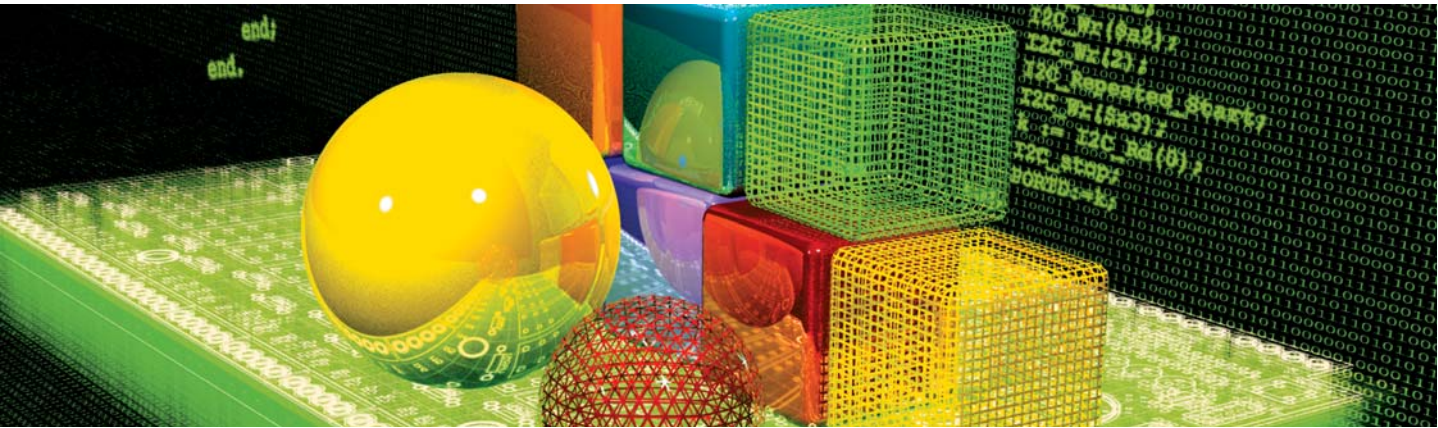


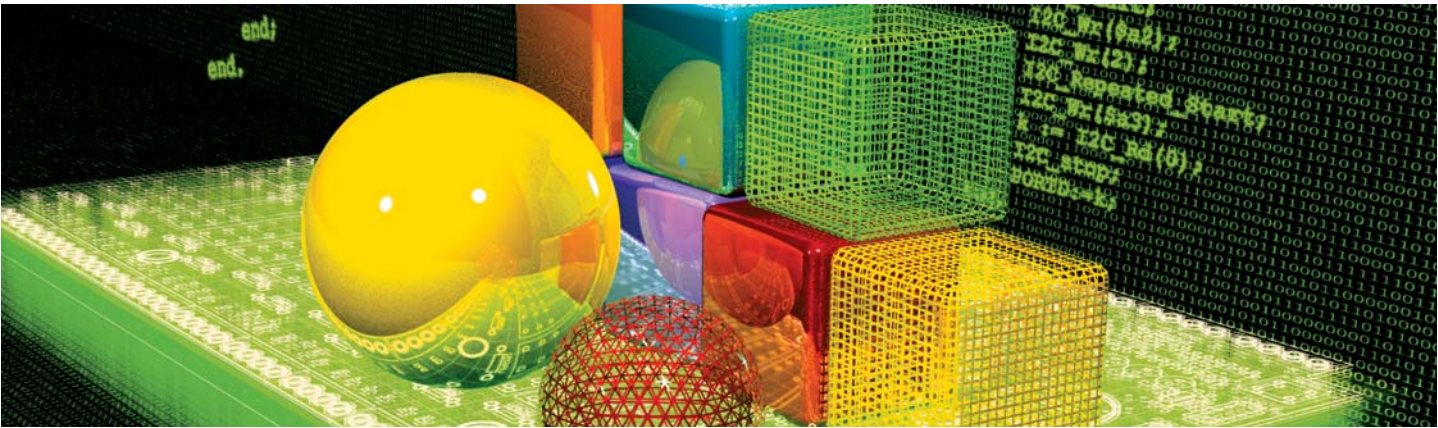
Table of Contents

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

CHAPTER 1: mikroPascal IDE	1
Quick Overview	1
Code Editor	3
Code Explorer	6
Debugger	7
Error Window	10
Statistics	11
Integrated Tools	14
Keyboard Shortcuts	16
 CHAPTER 2: Building Applications	 19
Source Files	20
Projects	21
New Project	21
Editing Project	21
Compilation	23
Output Files	23
Assembly View	23
Error Messages	24
 CHAPTER 3: mikroPascal Language Reference	 27
AVR Specifics	28
mikroPascal Specifics	29
Predefined Globals and Constants	29
Accessing Individual Bits	29
Interrupts	30
Linker Directives	31
Code Optimization	32
Lexical Elements	33
Whitespace	33
Comments	34
Tokens	35
Literals	36
Integer Literals	36
Floating Point Literals	36
Character Literals	37
String Literals	37
Keywords	38
Identifiers	39

Punctuators	40
Program Organization	42
Scope and Visibility	45
Units	46
Uses Clause	46
Main Unit	47
Other Units	48
Variables	49
Constants	50
Labels	51
Functions and Procedures	52
Functions	52
Procedures	53
Types	55
Simple Types	56
Arrays	57
Strings	58
Pointers	60
Records (Under Construction !!!)	61
Types Conversions	63
Implicit Conversion	63
Explicit Conversion	64
Operators	65
Precedence and Associativity	65
Arithmetic Operators	66
Relational Operators	67
Bitwise Operators	68
Boolean Operators	71
Expressions	72
Statements	73
asm Statement	73
Assignment Statements	74
Compound Statements	74
Conditional Statements	75
Iteration Statements	77
Jump Statements	79
Compiler Directives	82

CHAPTER 4: mikroPascal Libraries	84
Built-in Routines	86
Library Routines	92
ADC Library	93
CANSPI Library	95
Compact Flash Library	107
EEPROM Library	117
SPI Ethernet Library	119
Flash Memory Library	126
TWI(I2C) Library	128
Keypad Library	133
LCD Library (4-bit interface)	137
LCD Library (8-bit interface)	143
Graphic LCD Library	148
Multi Media Card Library	158
OneWire Library	167
PS/2 Library	178
PWM Library	182
Secure Digital Library	186
Software I2C Library	191
Software SPI Library	195
Software UART Library	198
Sound Library	201
SPI Library	205
USART Library	210
Util Library	218
Conversions Library	219
Math Library	223
Delays Library	230
String Library	231
LCD Custom Library	239
Port Expander Library	244
SPI Graphic LCD Library	252
Contact Us	263

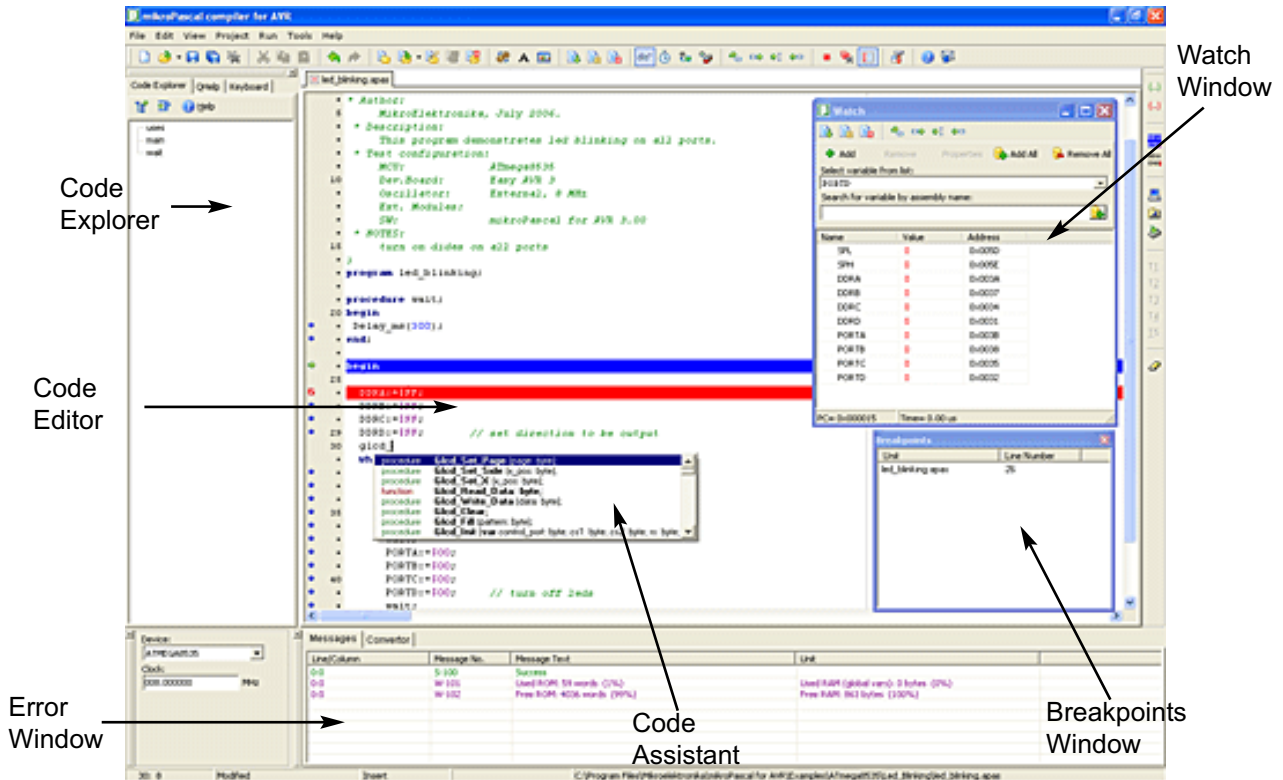


mikroPascal IDE

QUICK OVERVIEW

mikroPascal 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 AVR microcontrollers.



mikroPascal allows you to quickly develop and deploy complex applications:

- Write your Pascal source code using the highly advanced Code Editor
- Use the included mikroPascal libraries to dramatically speed up development: data acquisition, memory, displays, conversions, communications...
- 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.

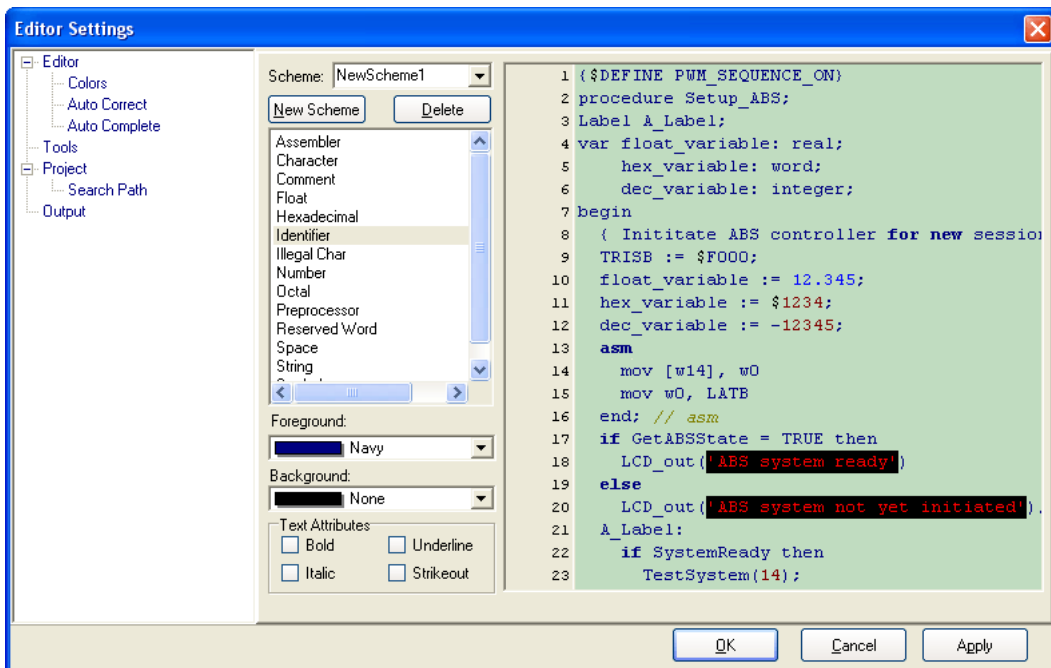
Advanced Editor features include:

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

You can customize these options from the 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 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(**channel : word;**)

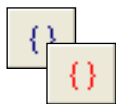
Code Template [CTR+J]

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 common typing mistakes. To access the list of recognized typos, select Tools > Options from the drop-down menu, or click the Tools Icon from the Settings Toolbar, and then select the 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.

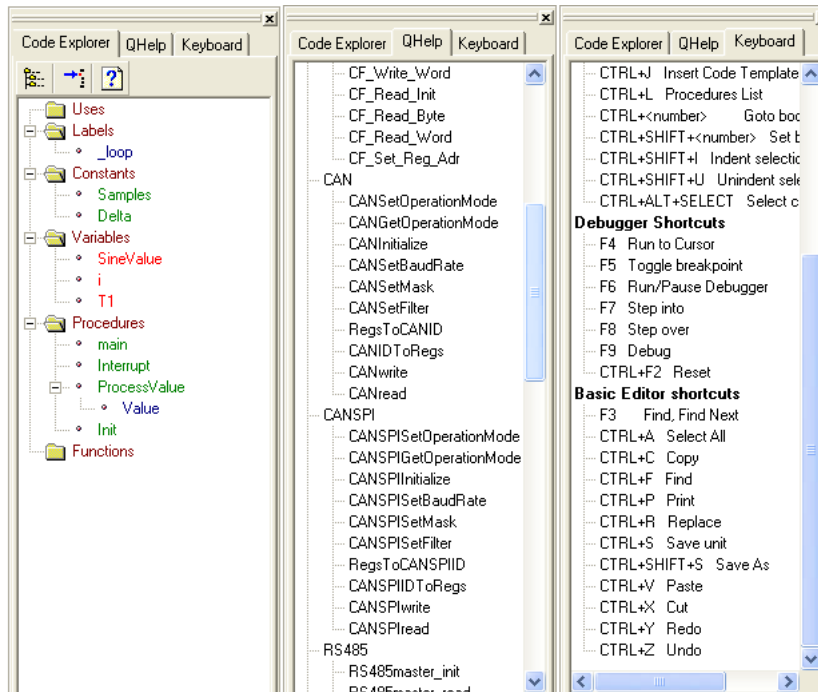
CODE EXPLORER

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 the 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 QHelp Tab opens the relevant Help topic. Keyboard Tab lists all available keyboard shortcuts in mikroPascal.



DEBUGGER



Start Debugger.

Source-level Debugger is an integral component of mikroPascal development environment. It is designed to simulate operations of Atmel Technology's AVR micros 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 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 Pascal (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 Pascal (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 Pascal (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.



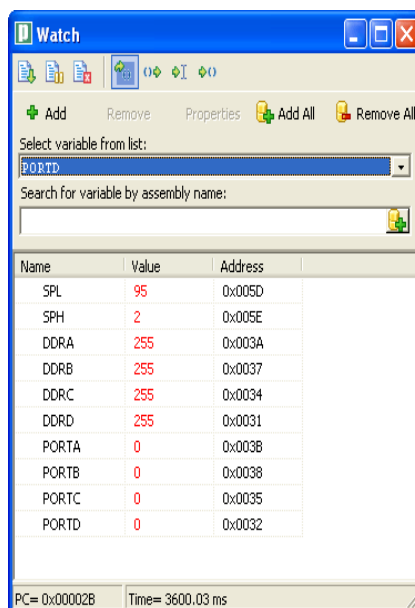
Toggle
Breakpoint.

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.



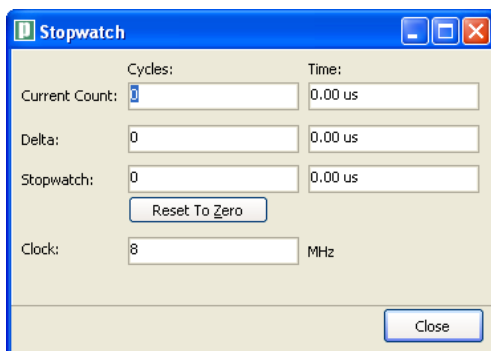
The Watch Window displays variables and registers of AVR, with their addresses and values. Values are updated as you go through the simulation. Use the drop-down menu to add and remove the items that you want to monitor. Recently changed items are colored red.

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 > Clockwatch.

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 Clockwatch 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.

STATISTICS

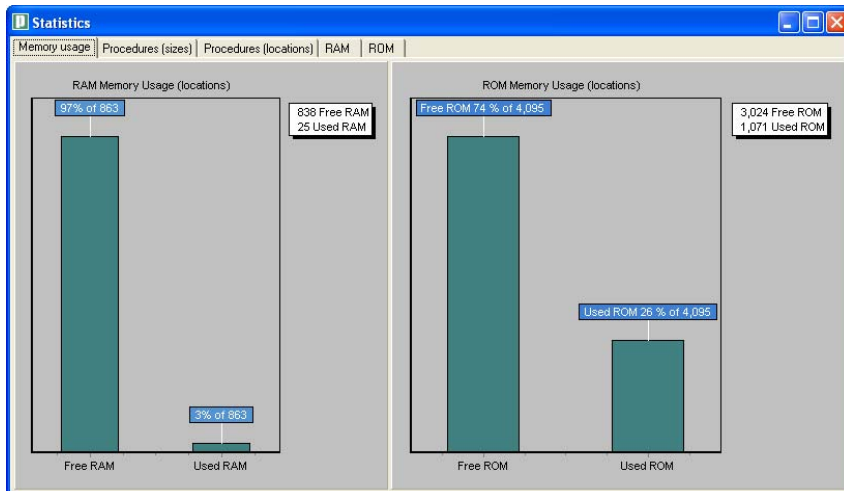


Statistics Icon.

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:

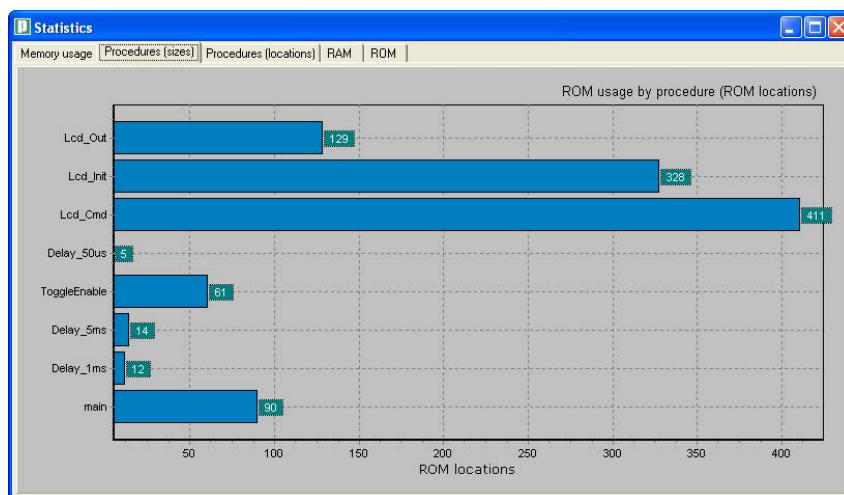
Memory Usage Window

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



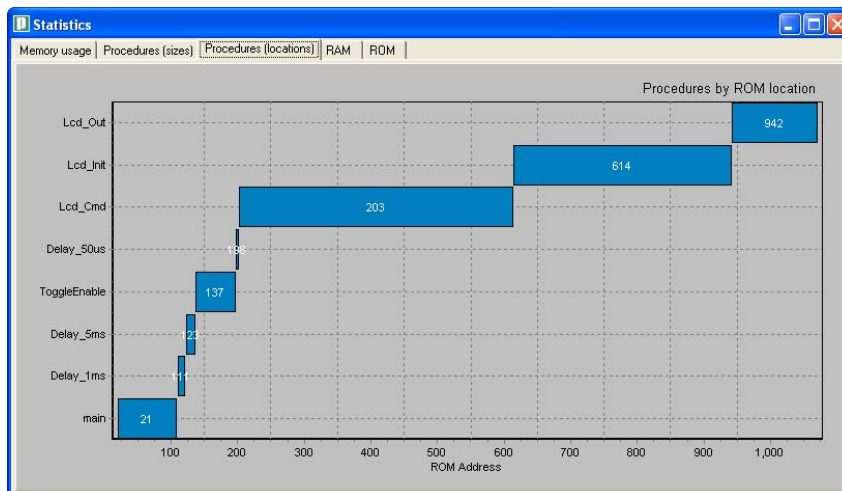
Procedures (Graph) Window

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

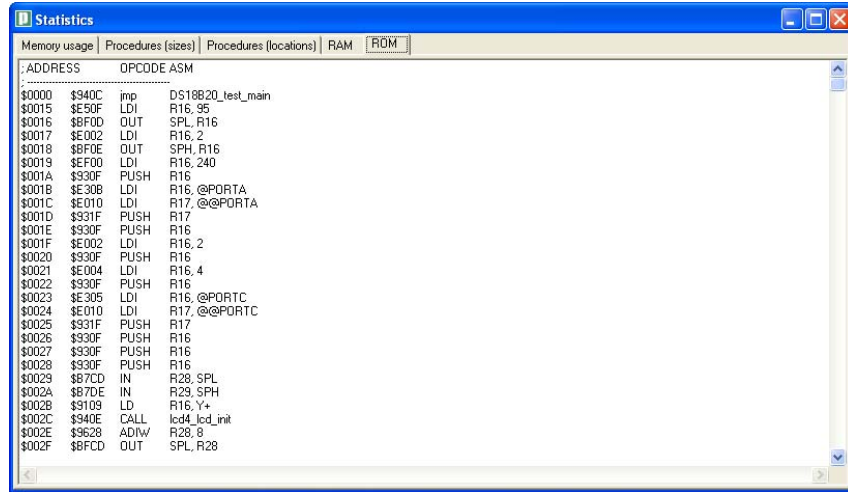
Summarizes all GPR and SFR registers and their addresses. It also displays symbolic names of variables and their addresses.

The 'RAM Window' displays two tables: 'General purpose registers (GPR)' and 'Special function registers (SFR)'.

General purpose registers (GPR)		Special function registers (SFR)	
Address	Register	Address	Register
<SP> + 0x00 i		0x005D	SPL
<SP> + 0x00 tmp		0x005E	SPH
<SP> + 0x00 tmp		0x0035	PORTC
<SP> + 0x00 tmpPtr		0x003B	PORTA
<SP> + 0x00 a			
<SP> + 0x00 ptr			
<SP> + 0x00 Row			
<SP> + 0x00 control_port			
<SP> + 0x00 out_char			
<SP> + 0x00 Column			
<SP> + 0x00 text			
<SP> + 0x00 EN			
<SP> + 0x00 RS			
<SP> + 0x00 data_port			
<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.

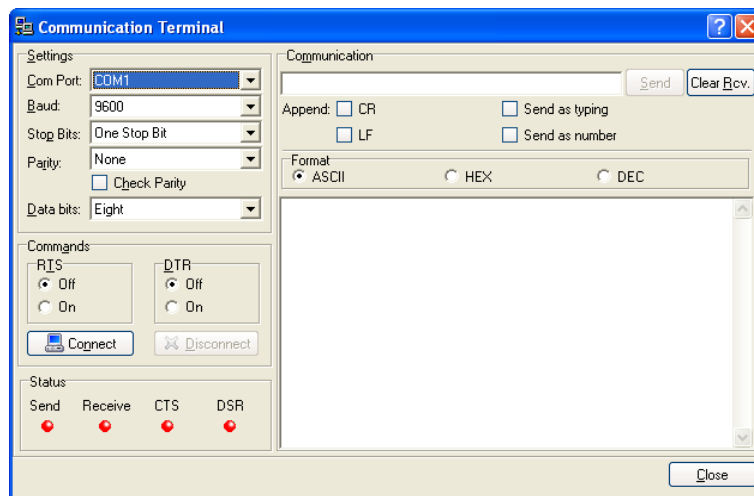


ADDRESS	OPCODE	ASM
\$0000	\$940C	jmp DS18B20_test_main
\$0015	\$E50F	LDI R16, 95
\$0016	\$8F0D	OUT SPL, R16
\$0017	\$E002	LDI R16, 2
\$0018	\$8F0E	OUT SPH, R16
\$0019	\$EF00	LDI R16, 240
\$001A	\$930F	PUSH R16
\$001B	\$E30B	LDI R16, @PORTA
\$001C	\$E010	LDI R17, @@PORTA
\$001D	\$931F	PUSH R17
\$001E	\$930F	PUSH R16
\$001F	\$E002	LDI R16, 2
\$0020	\$930F	PUSH R16
\$0021	\$E004	LDI R16, 4
\$0022	\$930F	PUSH R16
\$0023	\$E30B	LDI R16, @PORTC
\$0024	\$E010	LDI R17, @@PORTC
\$0025	\$931F	PUSH R17
\$0026	\$930F	PUSH R16
\$0027	\$930F	PUSH R16
\$0028	\$930F	PUSH R16
\$0029	\$87CD	IN R28, SPL
\$002A	\$87DE	IN R29, SPH
\$002B	\$9109	LD R16, Y+
\$002C	\$940E	CALL lcd4_lcd_init
\$002E	\$9628	ADIW R28, 8
\$002F	\$8FCD	OUT SPL, R28

INTEGRATED TOOLS

USART Terminal

mikroPascal 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.



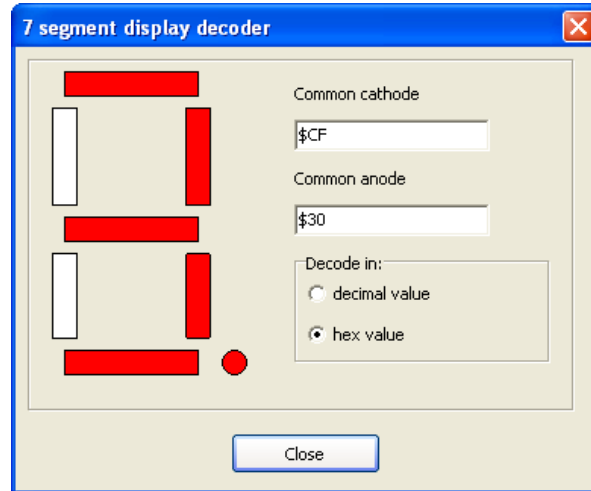
ASCII Chart

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

CHAR	DEC	HEX	BIN
NUL	0	0x00	0000 0000
SOH	1	0x01	0000 0001
STX	2	0x02	0000 0010
ETX	3	0x03	0000 0011
EOT	4	0x04	0000 0100
ENQ	5	0x05	0000 0101
ACK	6	0x06	0000 0110
BEL	7	0x07	0000 0111
BS	8	0x08	0000 1000
HT	9	0x09	0000 1001
LF	10	0x0A	0000 1010
VT	11	0x0B	0000 1011
FF	12	0x0C	0000 1100
CR	13	0x0D	0000 1101
S0	14	0x0E	0000 1110
SI	15	0x0F	0000 1111
DLE	16	0x10	0001 0000
DC1	17	0x11	0001 0001
DC2	18	0x12	0001 0010
DC3	19	0x13	0001 0011
DC4	20	0x14	0001 0100
NAK	21	0x15	0001 0101
SYN	22	0x16	0001 0110

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 mikroPascal 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

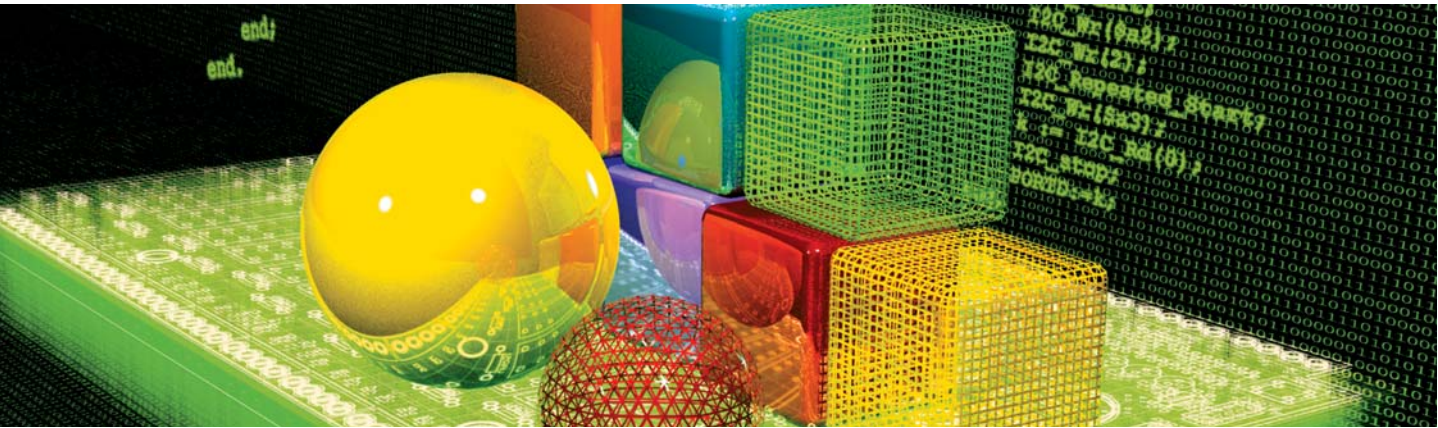
F3	Find, Find Next
CTRL+A	Select All
CTRL+C	Copy
CTRL+F	Find
CTRL+P	Print
CTRL+R	Replace
CTRL+S	Save unit
CTRL+SHIFT+S	Save As
CTRL+V	Paste
CTRL+X	Cut
CTRL+Y	Redo
CTRL+Z	Undo

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 RAM



Building Applications

Creating applications in mikroPascal 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.

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

SOURCE FILES

Source files containing Pascal code should have the extension `.apas`. List of source files relevant for the application is stored in project file with extension `.app`, 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 `uses` clause, mikroPascal will look for the file in following locations, in this particular order:

1. mikroPascal installation folder > “defs” folder
2. mikroPascal installation folder > “uses” folder
3. your custom search paths
4. the project folder (folder which contains the project file `.app`)

Managing Source Files



New File.

Creating a new source 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 `.apas`, is created automatically. mikroPascal does not require you to have source file named same as the project, it’s just a matter of convenience.

PROJECTS

mikroPascal organizes applications into *projects*, consisting of a single project file (extension `.app`) and one or more source files (extension `.apas`). 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 mikroPascal will create the appropriate project file.

Also, an empty source file named after the project will be created by default. mikroPascal 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.



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, mikroPascal will generate output files.

Output Files

Upon successful compilation, mikroPascal will generate output files in the project folder (folder which contains the project file `.app`). 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 mikroPascal, you can click the 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. The assembler 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.

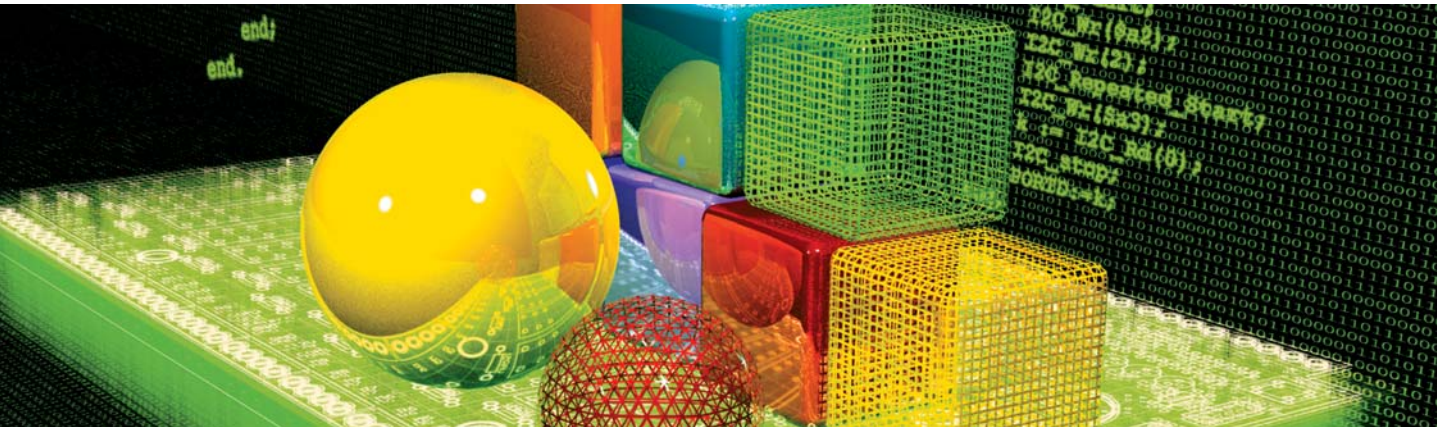
ERROR MESSAGES

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



mikroPascal Language Reference

Why Pascal 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 mikroPascal is very easy. This chapter will help you learn or recollect Pascal syntax, along with the specifics of programming AVR microcontrollers.

AVR SPECIFICS

In order to get the most from your mikroPascal 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 mikroPascal 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 mikroPascal.

mikroPascal 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.

Note:

AVcc must always be connected to VCC.

mikroPascal SPECIFICS

Predefined Globals and Constants

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

All AVR I/O Registers registers are implicitly declared as global variables of byte type, and are visible in the entire project. When creating a project, mikroPascal will include an appropriate .def file, containing declarations of available I/O Registers and constants (such as PORTB, DDRB, etc). Identifiers are all in upper-case, identical to nomenclature in Atmel datasheets.

For the complete set of predefined globals and constants, look for “Defs” in your mikroPascal installation folder, or probe the Code Assistant for specific letters (CTRL+SPACE in Editor).

Accessing Individual Bits

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

```
var myvar : longint;  
// range of applicable bits is myvar.0 .. myvar.31  
//...  
// If RB0 is set, set the 28th bit of myvar:  
if PORTB.0 = 1 then myvar.27 := 1;
```

There is no need for any special declarations; this kind of selective access is an intrinsic feature of mikroPascal 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:=0;).

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

Interrupt Example

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

```

procedure Timer_Interrupt0; org OVFOaddr;
begin
  asm
    CLI                      // disable interrupts
  end;
  PORTD:=not PORTD; // toggle PORTD
  asm
    SEI                      // enable interrupts
  end;
end;

```

Linker Directives

mikroPascal 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:

```
var x : byte; absolute $22;
// Variable x will occupy 1 byte at address $22

    y : 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:

```
var i : byte; absolute $33;
// Variable i will occupy 1 byte at address $33;

    jjjj : longint; absolute $30;
// Variable will occupy 4 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:

```
procedure proc(par : byte); org $200;
begin // Procedure will start at address $200
...
end;
```

Note: Directive `org` can be applied to any routine, but if `org` address is in IVT that procedure is considered as interrupt routine and has to satisfy interrupt conditions. See **Interrupt** for more info.

Code Optimization

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

Constant folding

All expressions that can be evaluated in the compile time (i.e. are constant) are being replaced by their result. $(3 + 5 \rightarrow 8)$;

Constant propagation

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

Copy propagation

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

Value numbering

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

"Dead code" elimination

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

Stack allocation

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

Local vars optimization

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

Better code generation and local optimization

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

LEXICAL ELEMENTS

These topics provide a formal definition of the mikroPascal 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 mikroPascal are derived from a series of operations performed on your programs by the compiler.

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

Whitespace

Whitespace is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, 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

```
var i : char;  
    j : word;
```

and

```
var  
i : char;  
  
    j : word;
```

are lexically equivalent and parse identically.

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'  
;
```

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.

There are two ways to create comments in mikroPascal. You can use multi-line comments:

```
{ All text between a left brace and a right brace  
  constitutes a comment. May span multiple lines. }
```

or single-line comments:

```
// Any text between a double-slash and the end of the  
// line constitutes a comment. May span one line only.
```


TOKENS

Token is the smallest element of a Pascal 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.

mikroPascal 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
;           // 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 0× indicates a hexadecimal numeral (for example, \$8F or 0×8F).

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

The allowed range of values is imposed by the largest data type in mikroPascal – `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.

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

Here are some examples:

```
0.           // = 0.0
2e-5         // = 2.0 * 10^-5
-.09E34      // = -0.09 * 10^34
```

Character Literals

Character literal is one character from the extended ASCII character set, enclosed with apostrophes. 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.

Note: Quotes ("") have no special meaning in mikroPascal.

String Literals

String literal is a sequence of up to 255 characters from the extended ASCII character set, enclosed with apostrophes. 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 apostrophes (*null string*) is stored as a single null character. You can assign a 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
```

Apostrophe itself cannot be a part of the string literal, i.e. there is no escape sequence. You could use the built-in function `Chr` to print an apostrophe:
`Chr(39)`. Also, see Strings and String Splicing.

KEYWORDS

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

Beside standard Pascal keywords, all relevant I/O registers are defined as global variables and represent reserved words that cannot be redefined (for example: PORTB, DDRB, TCCR0 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 mikroPascal:

absolute	function	read
and	goto	record
array	if	repeat
asm	implementation	shl
begin	interrupt	shr
boolean	in	step
break	is	string
case	label	then
char	mod	to
continue	not	type
const	or	unit
div	org	until
do	otherwise	uses
downto	print	var
else	procedure	while
end	program	with
for	real	xor

Also, mikroPascal 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 mikroPascal 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

Pascal 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 mikroPascal punctuators (also known as separators) include brackets, parentheses, comma, semicolon, colon, and dot.

Brackets

Brackets [] indicate single and multidimensional array subscripts:

```
var alphabet : array[ 1..30] of byte;
// ...
alphabet[ 3] := '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
function func2(n : word);   // Function declaration w/ parameters
```

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

Comma

The comma (,) separates the arguments in function calls:

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

Further, the comma separates identifiers in declarations:

```
var i, j, k : byte;
```

The comma also separates elements of array in initialization lists:

```
const MONTHS : array[ 1..12] of byte =  
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
```

Semicolon

The semicolon (;) is a statement terminator. Every statement in Pascal must be terminated by a semicolon. The only exception is the last (outer most) end statement in program which is terminated by dot.

For more information, see Statements.

Colon

Colon (:) is used in declarations to separate identifier list from type identifier. For example:

```
var  
    i, j : byte;  
    k    : word;
```

In the program, use the colon to indicate a labeled statement:

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

Dot

Dot (.) indicates access to a field of a record. For example:

```
person.surname := 'Smith';
```

For more information, refer to Records.

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

PROGRAM ORGANIZATION

mikroPascal 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 Units and to Scope and Visibility.

Organization of Main Unit

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 Units

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

Main unit should look like this:

```
program { program name }
uses { include other units }

//*****
/* Declarations (globals):
//*****

{ constants declarations }
const ...

{ variables declarations }
var ...

{ labels declarations }
label ...

{ procedures declarations }
procedure procedure_name
    { local declarations }
    begin
        ...
    end;

{ functions declarations }
function function_name
    { local declarations }
    begin
        ...
    end;

//*****
/* Program body:
//*****

begin
    { write your code here }
end.
```

Other units should look like this:

```

unit { unit name }
uses { include other units }

//*****
/* Interface (globals):
//*****
{ constants declarations }
const ...

{ variables declarations }
var ...

{ procedures prototypes }
procedure procedure_name(...);

{ functions prototypes }
function function_name(...);

//*****
/* Implementation:
//*****
implementation

{ constants declarations }
const ...

{ variables declarations }
var ...

{ labels declarations }
label ...

{ procedures declarations }
procedure procedure_name
{ local declarations }
begin
...
end;

{ functions declarations }
function function_name
{ local declarations }
begin
...
end;
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 unit, 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 unit, scope extends the interface section of a unit from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit.

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

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.

UNITS

In mikroPascal, each project consists of a single project file, and one or more unit files. Project file, with extension `.ppp` contains information about the project, while units, with extension `.ppas`, contain the actual source code.

Units allow you to:

- break large programs into encapsulated units 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 unit is stored in its own file and compiled separately; compiled units are linked to create an application. To build a project, the compiler needs either a source file or a compiled unit file for each unit.

Uses Clause

mikroPascal includes units by means of `uses` clause. It consists of the reserved word `uses`, followed by one or more comma-delimited unit names, followed by a semicolon. Extension of the file should not be included. There can be at most one `uses` clause in each source file, and it must appear immediately after the program (or unit) name.

Here's an example:

```
uses utils, strings, Unit2, MyUnit;
```

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

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

Every project in mikroPascal requires single main unit file. Main unit 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 unit. It contains the bare-bones of a program:

```
program MyProject;  
  
  { main procedure }  
begin  
    { Place program code here }  
end.
```

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

Place all global declarations (constants, variables, labels, routines) before the keyword `begin`.

Other Units

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

```
unit MyUnit;  
  
implementation  
  
end.
```

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

Interface Section

Part of the unit above the keyword `implementation` 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 unit. Prototypes must match the declarations exactly.

Implementation Section

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

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

By placing the prototype in the interface section of the unit (above the `implementation`) you can make the routine *public*, i.e. visible outside of unit. 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 `uses` statement, above the keyword `begin`.

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

```
var identifier_list : type;
```

The *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. For more information on variables' scope refer to the chapter Scope and Visibility.

Pascal allows shorthand syntax with only one keyword `var` followed by multiple variable declarations. For example:

```
var i, j, k : byte;  
    samples : array[100] of word;
```

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. Declare a constant like this:

```
const constant_name [ : 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” type that can accommodate value.

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

Pascal allows shorthand syntax with only one keyword `const` followed by multiple constant declarations. Here’s an example:

```
const
    MAX : longint = 10000;
    MIN = 1000;      // compiler will assume word type
    SWITCH = 'n';    // compiler will assume char type
    MSG = 'Hello';   // compiler will assume string type
    MONTHS : array[1..12] of byte =
        (31,28,31,30,31,30,31,31,30,31,30,31);
```


LABELS

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

```
label_identifier : statement
```

Before marking a statement, you must first declare the label. Labels are declared in declaration part of unit or routine, similar to variables and constants. Declare labels using the keyword `label`:

```
label label1, ..., labeln;
```

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

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

```
label loop;  
...  
loop: Beep;  
goto loop;
```

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. **Note:** mikroPascal does not support inline routines.

Functions

Function is declared like this:

```
function function_name(parameter_list) : return_type;  
    { local declarations }  
begin  
    { function body }  
end;
```

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 mikroPascal, parameters are always passed to function by value; to pass argument by the address, add the keyword `var` 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.

In standard Pascal, function name is automatically created local variable that can be used for returning a value of a function. Also, mikroPascal allows you to use the automatically created local variable `result` to assign the return value of a function if you find function name to be too ponderous.

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 ($n > 0$):

```
function power(x, n : byte) : longint;  
var i : byte;  
begin  
    i := 0; result := 1;  
    if n > 0 then  
        for i := 1 to n do result := result*x;  
end;
```

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

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

Procedures

Procedure is declared like this:

```
procedure procedure_name(parameter_list);  
    { local declarations }  
begin  
    { procedure body }  
end;
```

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 mikroPascal, parameters are always passed to procedure by value; to pass argument by the address, add the keyword `var` 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:

```
procedure time_prep(var sec, min, hr : byte);  
begin  
    sec := ((sec and $F0) shr 4)*10 + (sec and $0F);  
    min := ((min and $F0) shr 4)*10 + (min and $0F);  
    hr  := ((hr  and $F0) shr 4)*10 + (hr  and $0F);  
end;
```

TYPES

Pascal 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.

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

Type Categories

Types can be divided into:

- simple types
- arrays
- strings
- pointers

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 mikroPascal:

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
real	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:

```
array[ index_start .. index_end] of type
```

Each of the elements of an array is numbered from *index_start* through the *index_end*. Specifier *index_start* can be omitted along with dots, in which case it defaults to zero. 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:

```
var
    weekdays : array[ 1..7] of byte;
    samples  : array[ 50] of word;

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 : array[ 1..12] of byte
               = ( 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 );
```

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

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_name string[ length]
```

Specifier *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:

```
var
  msg1 : string[ 20] ;
  msg2 : 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:

```
var s : 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.

String Splicing

mikroPascal allows you to splice strings by means of plus character. This kind of concatenation is applicable to string variables/literals and character variables/literals. For control characters, use the non-quoted hash sign and a numeral (e.g. #13 for CR).

The result of splicing is of string type, naturally. See also `Strcat` function.

Here is an example:

```
var msg      : string[ 100] ;
    res_txt   : string[ 5] ;
    res, channel : word;

begin

    //...

    // Get result of ADC
    res := Adc_Read(channel);

    // Create string out of numeric result
    WordToStr(res, res_txt);

    // Prepare message for output
    msg := 'Result is' +      // Text "Result is"
           #13#10          +  // Append CR/LF sequence
           res_txt          +  // Result of ADC
           '.';              // Append a dot

    //...
```

Note: mikroPascal includes a String Library which automatizes string related tasks.

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 caret 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 caret after the variable name. For example, let's declare variable `p` which points to a `word`, and then assign the pointed memory location value 5:

```
var p : ^word;  
...  
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`).

RECORDS

A record (analogous to a structure in some languages) represents a heterogeneous set of elements. Each element is called *field*; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```
type recordTypeName = record
    fieldList1 : type1;
    ...
    fieldListn : typen;
end;
```

where *recordTypeName* is a valid identifier, each *type* denotes a type, and each *fieldList* is a valid identifier or a comma-delimited list of identifiers. The scope of a field identifier is limited to the record in which it occurs, so you don't need to worry about naming conflicts between field identifiers and other variables.

Note: In mikroPascal, you cannot use the `record` construction directly in variable declarations, i.e. without `type`.

For example, the following declaration creates a record type called `TDot`:

```
type
    TDot = record
        x, y : real;
    end;
```

Each `TDot` contains two fields: `x` and `y` coordinates; memory is allocated when you instantiate the record, like this:

```
var m, n: TDot;
```

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

A field can be of previously defined record type. For example:

```
// Structure defining a circle:
type
    TCircle = record
        radius : real;
        center : TDot;
    end;
```

Accessing Fields

You can access the fields of a record by means of dot (.) as a direct field selector.

If we declare variables `circle1` and `circle2` of previously defined type

`TCircle`:

```
var circle1, circle2 : TCircle;
```

we could access their individual fields 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 fields
```

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). mikroPascal 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:

<code>byte/char</code>	->	<code>word</code>
<code>short</code>	->	<code>integer</code>
<code>short</code>	->	<code>longint</code>
<code>integer</code>	->	<code>longint</code>
<code>integral</code>	->	<code>real</code>

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).

```
var i : byte; j : 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:

```
var a : byte; b : 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 mikroPascal:

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

Operators Precedence and Associativity

There are 4 precedence categories in mikroPascal. 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 shl shr	left-to-right
2	2	+ - or xor	left-to-right
1	2	= <> < > <= >=	left-to-right

Arithmetic Operators

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

All arithmetic operators associate from left to right.

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

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

For example: `b := -a`

Division by Zero

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

Relational Operators

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

All relational operators associate from left to right.

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

Relational Operators in Expressions

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

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

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>shl</code>	bitwise shift left; moves the bits to the left, see below	3
<code>shr</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 zeros. If number is converted from more complex to less complex data type, data is simply truncated (upper bytes are lost).

For example:

```
var a : byte; b : 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 zeros if sign bit is 0 (number is positive). If number is converted from more complex data type to less complex, data is simply truncated (upper bytes are lost).

For example:

```
var a : byte; b : word;
...
  a := -12;
  b := $70FF;
  b := b and a;

  { a is sign extended, with the upper byte equal to $FF;
    b becomes $70F4 }
```

Bitwise Shift Operators

Binary operators `shl` and `shr` 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 (`shl`), left most bits are discarded, and “new” bytes on the right are assigned zeros. Thus, shifting unsigned operand to the 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 (`shr`), right most bits are discarded, and the “freed” bytes on the left are assigned zeros (in case of unsigned operand) or the value of the sign bit zeros (in case of signed operand). Shifting operand to the right by n positions is equivalent to dividing it by 2^n .

Boolean Operators

Although mikroPascal 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 mikroPascal.

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 semicolon (;). In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code.

The simplest statements include assignments, routine calls, and jump statements. These can be combined to form loops, branches, and other structured statements.

Statements can be roughly divided into:

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

asm Statement

mikroPascal 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;
```

Pascal 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 Pascal variable in embedded assembly only, be sure to at least initialize it (assign it initial value) in Pascal code; otherwise, linker will issue an error. This does not apply to predefined globals such as `PORTB`.

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.

Compound Statements

A compound statement, or *block*, is a list of statements enclosed by keywords `begin` and `end`:

```
begin  
    statements  
end;
```

Syntactically, a block is considered to be a single statement which allows using it when Pascal syntax requires a single statement. Blocks can be nested up to the limits of memory.

For example, `while` loop expects one statement in its body, so we can pass it a compound statement:

```
while i < n do  
    begin  
        temp := a[ i ] ;  
        a[ i ] := b[ i ] ;  
        b[ i ] := temp ;  
        n := n + 1 ;  
    end;
```


Conditional Statements

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

If Statement

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

```
if expression then statement1 [ else statement2]
```

When *expression* evaluates to true, *statement1* executes. If *expression* is false, *statement2* executes. The *expression* must convert to a boolean type (true or false); otherwise, the condition is ill-formed. The `else` keyword with an alternate statement (*statement2*) is optional. There should never be a semicolon before the keyword `else`.

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.

Case Statement

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

```
case selector of
    value_1 : statement_1
    ...
    value_n : statement_n
    [ else default_statement ]
end;
```

The *selector* is an expression which should evaluate as integral value. The *values* can be literals, constants, or expressions, and *statements* can be any statements. The `else` clause is optional. If using the `else` branch, note that there should never be a semicolon before the keyword `else`.

First, the *selector* expression (condition) is evaluated. The `case` statement then compares it against all the available *values*. If the match is found, the *statement* following the match evaluates, and `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_statement* in the `else` clause (if there is one) is executed.

Here's a simple example of `case` statement::

```
case operator of
    '*' : result := n1 * n2;
    '+' : result := n1 + n2;
    '-' : result := n1 - n2
else result := 0;
end;
```

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

```
case reg of
    1,2,3,4:  opmode := 1;
    5,6,7:   opmode := 2;
end;
```

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

Iteration Statements (Loops)

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

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 do statement
// or
for counter := initial_value downto final_value do statement
```

The *counter* is a variable which increments (or decrements if you use `downto`) with each iteration of the loop. Before the first iteration, *counter* is set to the *initial_value* and will increment (or decrement) until it reaches the *final_value*. With each iteration, *statement* will be executed.

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

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 do s := s + a[i] * b[i];
```

The `for` statement results in an endless loop if the *final_value* equals or exceeds the range of *counter*'s type. More legible way to create an endless loop in Pascal is to use the statement `while TRUE do`.

While Statement

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

```
while expression do statement
```

The *statement* is executed repeatedly as long as the *expression* evaluates true. The test takes place before the *statement* is 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:

```
s := 0; i := 0;
while i < n do
begin
    s := s + a[i] * b[i];
    i := i + 1;
end;
```

Repeat Statement

The `repeat` statement executes until the condition becomes false. Syntax of `repeat` statement is:

```
repeat statement until expression
```

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

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

```
s := 0; i := 0;
repeat
    begin
        s := s + a[i] * b[i];
        i := i + 1;
    end;
until i = n;
```

Jump Statements

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

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`, or `repeat` block).

For example:

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

// 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 the 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 `repeat` 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.

The label declaration, marked statement, and `goto` statement must belong to the same block. Hence it is not possible to jump into or out of a procedure or function.

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:

```
for (...) do  
  begin  
    for (...) do  
      begin  
        ...  
        if (disaster) then goto Error;  
        ...  
      end;  
    end;  
  .  
  .  
  .  
Error: // error handling code
```

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:

```
procedure Proc1();  
var error: byte;  
begin  
    ... // we're doing something here  
    if error = TRUE then exit;  
    ... // some code, which won't be executed if error is true  
end;
```

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

mikroPascal treats comments beginning with a “\$” immediately following the opening brace as a compiler directive; for example, { \$ELSE} . 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.

Supported compiler directives are:

```
$DEFINE  
$UNDEFINE  
$IFDEF  
$ELSE  
$ENDIF
```

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.

Note: Pascal does not support macros; directives \$DEFINE and \$UNDEFINE do not create/destroy macros; they only provide flags for directive \$IFDEF to check against.

Directives \$IFDEF..\$ELSE

Conditional compilation is carried out by \$IFDEF 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 is terminated by the \$ENDIF directive, and can have an optional \$ELSE clause:

```
{ $IFDEF flag}
    <block of code>
{ $ELSE}
    <alternate block of code>
{ $ENDIF}
```

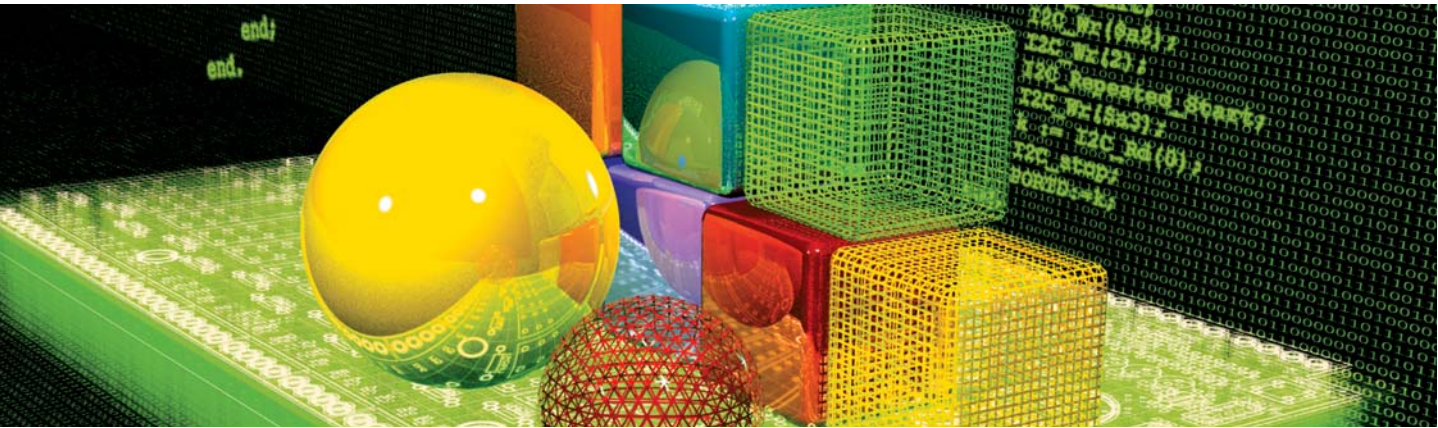
First, \$IFDEF checks if flag is defined by means of \$DEFINE. If so, only *<block of code>* will be compiled. Otherwise, *<alternate block of code>* will be compiled. The \$ENDIF ends the conditional sequence. The result of the preceding scenario is that only one section of code (possibly empty) is 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 an example:

```
// Uncomment the appropriate flag for your application:
//{$DEFINE resolution10}
//{$DEFINE resolution12}

{ $IFDEF resolution10}
    // <code specific to 10-bit resolution>
{ $ELSE}
    { $IFDEF resolution12}
        // <code specific to 12-bit resolution>
    { $ELSE}
        // <default code>
    { $ENDIF}
{ $ENDIF}
```

mikroPascal Libraries

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

BUILT-IN ROUTINES

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

Currently, mikroPascal includes the following built-in functions:

- Inc
- Dec
- Chr
- Ord
- SetBit
- ClearBit
- TestBit
- Lo
- Hi
- Higher
- Highest
- Swap
- Clock_Khz
- Clock_Mhz
- CLI
- SEI

Inc

Prototype	function Inc(var par : longint) : longint;
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	function Dec(var par : longint) : longint;
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	function Chr(<code>code</code> : byte) : char;
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);</code> // returns a linefeed character

Ord

Prototype	function Ord(character : char) : byte;
Returns	ASCII code of the character.
Description	Function returns ASCII code of the character. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Example	<code>c := Ord('A'); // returns 65</code>

SetBit

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

ClearBit

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

TestBit

Prototype	function TestBit(register, rbit : byte) : byte;
Returns	If bit is set, returns 1, otherwise returns 0.
Description	Function tests if the bit <code>rbit</code> of <code>register</code> is set. If set, function returns 1, otherwise returns 0. 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>flag := TestBit(PORTE, 2); // 1 if RE2 is set, otherwise 0</code>

Lo

Prototype	function Lo(number : byte..longint) : byte;
Returns	Returns the lowest 8 bits (byte) of <code>number</code> , bits 0..7.
Description	Function returns the lowest byte of <code>number</code> . Function does not interpret bit patterns of <code>number</code> – it merely returns 8 bits as found in register.
Example	<code>a := Lo(\$1AC30F4); // Equals \$F4</code>

Hi

Prototype	function Hi(number : word..longint) : byte;
Returns	Returns byte next to the lowest byte of <code>number</code> , bits 8..15.
Description	Function returns byte next to the lowest byte of <code>number</code> . Function does not interpret bit patterns of <code>number</code> – it merely returns 8 bits as found in register.
Example	<code>a := Hi(\$1AC30F4); // Equals \$30</code>

Higher

Prototype	function Higher(number : longint) : 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(\$1AC30F4); // Equals \$AC

Highest

Prototype	function Highest(number : longint) : 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(\$1AC30F4); // Equals \$01

Swap

Prototype	function Swap(var arg : byte) : 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 := \$F0; PORTA := Swap(PORTB); // PORTA = PORTB = \$0F

Clock_Khz

Prototype	function Clock_Khz : word;
Returns	Device clock in KHz.
Description	Returns device clock in KHz, rounded to the nearest integer.
Example	clk := Clock_Khz;

Clock_Mhz

Prototype	function Clock_Mhz : byte;
Returns	Device clock in MHz.
Description	Returns device clock in MHz, rounded to the nearest integer.
Example	clk := Clock_Mhz;

CLI

Prototype	asm CLI;
Returns	Nothing.
Description	Global interrupt disable.
Example	CLI;

SEI

Prototype	asm SEI;
Returns	Nothing.
Description	Global interrupt enable.
Example	SEI;

LIBRARY ROUTINES

mikroPascal provides a set of libraries which simplifies the initialization and use of AVR MCU and its units. 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
- CANSPI Library
- Compact Flash Library
- EEPROM Library
- SPI Ethernet Library
- Flash Memory Library
- Graphic LCD Library
- T6963C Graphic LCD Library
- TWI Library
- Keypad Library
- LCD Library
- LCD8 Library
- Multi Media Card Library
- OneWire Library
- PS/2 Library
- PWM Library
- Software I2C Library
- Software SPI Library
- Software UART Library
- Sound Library
- SPI Library
- USART Library
- Util Library
- Conversions Library
- Delays Library
- Math Library
- String Library
- Port Expander Library
- SPI Graphic LCD Library
- LCD Custom Library

ADC Library

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

Adc_Read

Prototype	function <code>Adc_Read(channel : byte) : word;</code>
Returns	10-bit unsigned value read from the specified ADC channel.
Description	<p>Initializes AVR's internal ADC unit 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	AVR MCU with built-in ADC unit. You should consult the Datasheet documentation for specific device).
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 PORTB (lower 8 bits) and PORTC(2 most significant bits).

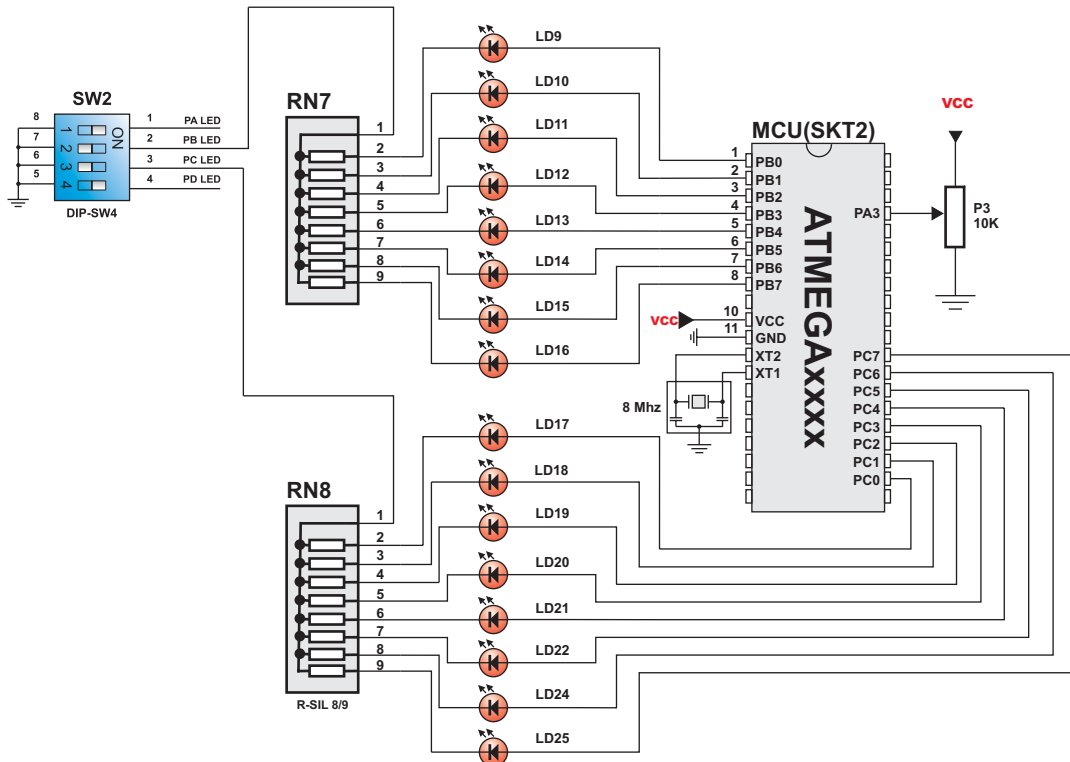
```

program Adc_Test;
  var data : word;

  begin
    DDRB := $FF;    // PORTB output
    DDRC := $FF;    // PORTC output

    while true do
      begin
        data := Adc_Read(3); // read channel 3
        PORTB := Lo(data);    // display on led
        PORTC := Hi(data);    // upper byte on portc leds
        Delay_ms(30);
      end;
    end.
  
```

Hardware Connection



CANSPI Library

SPI unit is available with a number of AVRmicros. mikroPascal provides a library (driver) for working with the external CAN units (such as MCP2515 or MCP2510) via SPI.

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

Note: CANSPI functions are supported by any AVR MCU that has SPI interface. Example of HW connection is given at the end of the chapter.

Library Routines

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

Following routines are for the internal use by compiler only:

```
RegsToCANSPIID  
CANSPIIDToRegs
```

CANSPISetOperationMode

Prototype	procedure CANSPISetOperationMode(mode : byte, wait_flag : byte);
Description	<p>Sets CAN to requested mode, i.e. copies mode to CANSTAT. Parameter <code>mode</code> needs to be one of CAN_OP_MODE constants (see CAN constants, page 141).</p> <p>Parameter <code>wait_flag</code> needs to be either 0 or 0xFF: If set to 0xFF, this is a blocking call – the function won't "return" until the requested mode is set. If 0, this is a non-blocking call. It does not verify if CAN unit is switched to requested mode or not. Caller must use function CANSPIGetOperationMode to verify correct operation mode before performing mode specific operation.</p>
Requires	CANSPI functions are supported by any AVR MCU that has SPI interface.
Example	CANSPISetOperationMode(CAN_MODE_CONFIG, \$FF);

CANSPIGetOperationMode

Prototype	function CANSPIGetOperationMode : byte;
Returns	Current opmode.
Description	Function returns current operational mode of CAN unit.
Example	if (CANSPIGetOperationMode = CAN_MODE_CONFIG) then ...

CANSPIInitialize

Prototype	procedure CANSPIInitialize(SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);
Description	<p>Initializes CANSPI. All pending transmissions are aborted. Sets all mask registers to 0 to allow all messages.</p> <p>Filter registers are set according to flag value:</p> <pre> if ((CAN_CONFIG_FLAGS and CAN_CONFIG_VALID_XTD_MSG) = 0) then // Set all filters to XTD_MSG else if ((config and CONFIG_VALID_STD_MSG) = 0) then // Set all filters to STD_MSG else // Set half the filters to STD, and the rest to XTD_MSG </pre> <p>Parameters:</p> <p>SJW as defined in datasheet (1–4) BRP as defined in datasheet (1–64) PHSEG1 as defined in datasheet (1–8) PHSEG2 as defined in datasheet (1–8) PROPSEG as defined in datasheet (1–8) CAN_CONFIG_FLAGS is formed from predefined constants (see CAN constants).</p>
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	<pre> init := CAN_CONFIG_SAMPLE_THRICE and CAN_CONFIG_PHSEG2_PRG_ON and CAN_CONFIG_STD_MSG and CAN_CONFIG_DBL_BUFFER_ON and CAN_CONFIG_VALID_XTD_MSG and CAN_CONFIG_LINE_FILTER_OFF; ... CANSPIInitialize(1, 1, 3, 3, 1, init); // Initialize CANSPI </pre>

CANSPISetBaudRate

Prototype	procedure CANSPISetBaudRate(SJW, BRP, PHSEG1, PHSEG2, PROPSEG, CAN_CONFIG_FLAGS : byte);
Description	<p>Sets CANSPI baud rate. Due to complexity of CANSPI protocol, you cannot simply force a bps value. Instead, use this function when CANSPI is in Config mode. Refer to datasheet for details.</p> <p>Parameters:</p> <p>SJW as defined in datasheet (1–4) BRP as defined in datasheet (1–64) PHSEG1 as defined in datasheet (1–8) PHSEG2 as defined in datasheet (1–8) PROPSEG as defined in datasheet (1–8) CAN_CONFIG_FLAGS is formed from predefined constants (see CAN constants).</p>
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	<pre>init := CAN_CONFIG_SAMPLE_THRICE and CAN_CONFIG_PHSEG2_PRG_ON and CAN_CONFIG_STD_MSG and CAN_CONFIG_DBL_BUFFER_ON and CAN_CONFIG_VALID_XTD_MSG and CAN_CONFIG_LINE_FILTER_OFF ... CANSPISetBaudRate(1, 1, 3, 3, 1, init);</pre>

CANSPISetMask

Prototype	procedure CANSPISetMask(CAN_MASK : byte; value : longint; CAN_CONFIG_FLAGS : byte);
Description	<p>Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers.</p> <p>Parameters: CAN_MASK is one of predefined constant values (see CAN constants); value is the mask register value; CAN_CONFIG_FLAGS selects type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG.</p>
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	<pre>// Set all mask bits to 1, i.e. all filtered bits are relevant: CANSPISetMask(CAN_MASK_B1, -1, CAN_CONFIG_XTD_MSG); { Note that -1 is just a cheaper way to write \$FFFFFFFF. Complement will do the trick and fill it up with ones }</pre>

CANSPISetFilter

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

CANSPIRead

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

CANSPIWrite

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

CAN Constants

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

CAN_OP_MODE

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

```
const CAN_MODE_BITS           = $E0;  // Use it to access mode bits
const CAN_MODE_NORMAL        = 0;
const CAN_MODE_SLEEP         = $20;
const CAN_MODE_LOOP          = $40;
const CAN_MODE_LISTEN        = $60;
const CAN_MODE_CONFIG        = $80;
```

CAN_CONFIG_FLAGS

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

```
const CAN_CONFIG_DEFAULT      = $FF;  // 11111111

const CAN_CONFIG_PHSEG2_PRG_BIT = $01;
const CAN_CONFIG_PHSEG2_PRG_ON  = $FF;  // XXXXXXX1
const CAN_CONFIG_PHSEG2_PRG_OFF = $FE;  // XXXXXXX0

const CAN_CONFIG_LINE_FILTER_BIT = $02;
const CAN_CONFIG_LINE_FILTER_ON  = $FF;  // XXXXXX1X
const CAN_CONFIG_LINE_FILTER_OFF = $FD;  // XXXXXX0X

const CAN_CONFIG_SAMPLE_BIT     = $04;
const CAN_CONFIG_SAMPLE_ONCE    = $FF;  // XXXXX1XX
const CAN_CONFIG_SAMPLE_THRICE   = $FB;  // XXXXX0XX

const CAN_CONFIG_MSG_TYPE_BIT   = $08;
const CAN_CONFIG_STD_MSG        = $FF;  // XXXX1XXX
const CAN_CONFIG_XTD_MSG        = $F7;  // XXXX0XXX

// continues..
```

```
// ..continued
```

```
const CAN_CONFIG_DBL_BUFFER_BIT      = $10;
const CAN_CONFIG_DBL_BUFFER_ON       = $FF;    // XXX1XXXX
const CAN_CONFIG_DBL_BUFFER_OFF      = $EF;    // XXX0XXXX

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

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

```
init := CAN_CONFIG_SAMPLE_THRICE and CAN_CONFIG_PHSEG2_PRG_ON and
        CAN_CONFIG_STD_MSG       and CAN_CONFIG_DBL_BUFFER_ON and
        CAN_CONFIG_VALID_XTD_MSG and CAN_CONFIG_LINE_FILTER_OFF;
//...
CANInitialize(1, 1, 3, 3, 1, init);    // initialize CAN
```

CAN_TX_MSG_FLAGS

CAN_TX_MSG_FLAGS are flags related to transmission of a CAN message:

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

const CAN_TX_FRAME_BIT         = $08;
const CAN_TX_STD_FRAME         = $FF;    // XXXXX1XX
const CAN_TX_XTD_FRAME         = $F7;    // XXXXX0XX

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

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

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

CAN_RX_MSG_FLAGS

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

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

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

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

CAN_MASK

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

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

CAN_FILTER

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

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

Library Example

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

```

program canSPI_first;

uses CANspi;

var aa, aa1, len, aa2 : byte;
    data : array[8] of byte;
    id    : longint;
    flag  : byte;

begin
    DDRB:=DDRB and $A3; // seting to input direction registers
    DDRB:=DDRB or  $BF; // seting to output direction registers

    Spil_Init;

    DDRC      := $FF;
    PORTC     := 0;

    DDRD      := $FF;
    PORTD     := 0;

    aa := 0;
    aa1 := 0;
    aa2 := 0;
    aa := CAN_CONFIG_SAMPLE_THRICE and // form value to be used
          CAN_CONFIG_PHSEG2_PRG_ON and // with CANSPIInitialize
          CAN_CONFIG_STD_MSG and
          CAN_CONFIG_DBL_BUFFER_ON and
          CAN_CONFIG_VALID_XTD_MSG;

    PORTB.2 := 1;

    aa1 := CAN_TX_PRIORITY_0 and // form value to be used
           CAN_TX_XTD_FRAME and // with CANSPISendMessage
           CAN_TX_NO_RTR_FRAME;
    PORTB.0 := 1;
    CANSPIInitialize( 1,1,3,3,1,aa);

    // initialize external CAN module

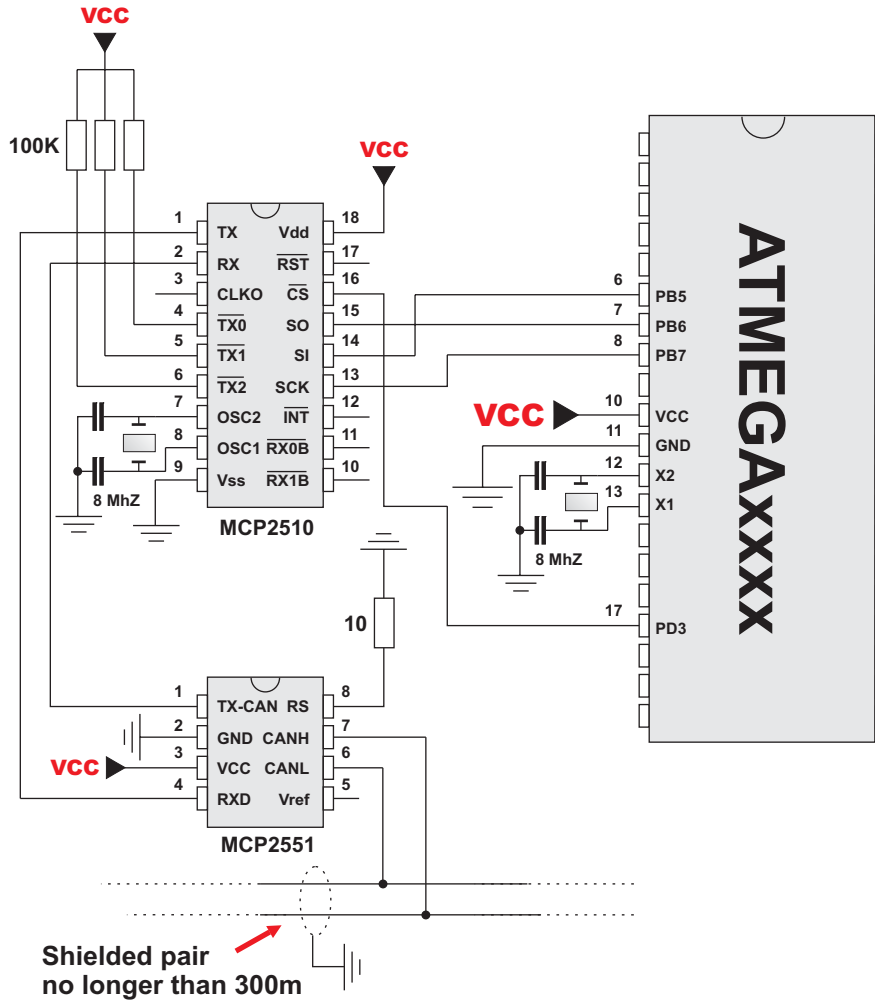
    // continues ..

```

```
// continued ..

portd := 3;
CANSPISetOperationMode(CAN_MODE_CONFIG,0xFF);
// set CONFIGURATION mode
portd := 4;
ID := -1;
CANSPISetMask(CAN_MASK_B1,id,CAN_CONFIG_XTD_MSG);
// set all mask1 bits to ones
CANSPISetMask(CAN_MASK_B2,id,CAN_CONFIG_XTD_MSG);
// set all mask2 bits to ones
CANSPISetFilter(CAN_FILTER_B2_F4,3,CAN_CONFIG_XTD_MSG);
// set id of filter B1_F1 to 12111
CANSPISetOperationMode(CAN_MODE_NORMAL,0xFF);
// set NORMAL mode
portd := 5;
data[ 0] := 23;
id := 12111;
CANSPIWrite(id,data,1,aal);
while true do
  begin
    flag := CANSPIRead(id , data , len, aa2); // receive data, if any
    if ((id = 3) and (flag = true)) then
      begin
        PORTC := data[ 0] ;
        data[ 0] := data[ 0] + 1;           // output data at portB
        id:=12111;
        delay_ms(1000);
        CANSPIWrite(id,data,1,aal);
        if (len = 2) then                  // send incremented data back
          begin
            PORTD := data[ 1] ;           // if message contains two data bytes
          end;
        end;
      end;
    end;                                // output second byte at portd
  end;
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
```

Cf_Init

Prototype	procedure Cf_Init(var ctrlport, dataport : 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	function Cf_Detect : 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: repeat nop; until Cf_Detect = 1;</pre>

Cf_Total_Size

Prototype	function Cf_Total_Size : logint;
Returns	Card size in kilobytes.
Description	Returns size of Compact Flash card in kilobytes.
Example	size := Cf_Total_Size();

Cf_Enable

Prototype	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.
Example	Cf_Enable();

Cf_Disable

Prototype	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	procedure Cf_Read_Init(address : longint; sectcnt : byte);
Description	Initializes CF card for reading. Parameter 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	function Cf_Read_Byte : 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	procedure Cf_Write_Init(address : longint; sectcnt : byte);
Description	Initializes CF card for writing. Parameter <code>address</code> specifies sector address where data will be stored, and <code>sectcnt</code> 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	procedure Cf_Write_Byte(data : byte);
Description	Writes one byte (<code>data</code>) 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	procedure Cf_Fat_Init(var control_port: byte; wr, rd, a2, a1, a0, ry, ce, cd : byte; var data_port : 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	function Cf_Fat_Assign(var filename : array [12] of char, create_file : byte): 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	procedure Cf_Fat_Reset(var size : 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	procedure Cf_Fat_Read(var bdata : 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	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	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	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	procedure Cf_Fat_Write(var fdata : array[512] of byte; data_len : 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); <i>// write data to the assigned file</i>

Cf_Fat_Set_File_Date

Prototype	procedure Cf_Fat_Set_File_Date(year : word; month, day, hours, mins, seconds : 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	procedure Cf_Fat_Get_File_Date(var year : word; var month, day, hours, mins: 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	function Cf_Fat_Get_File_Size: 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;

Library Examples

The example waits until the CF card is inserted, and when plugged, it writes and reads bytes.

```
program Cf_example;
var i : word;

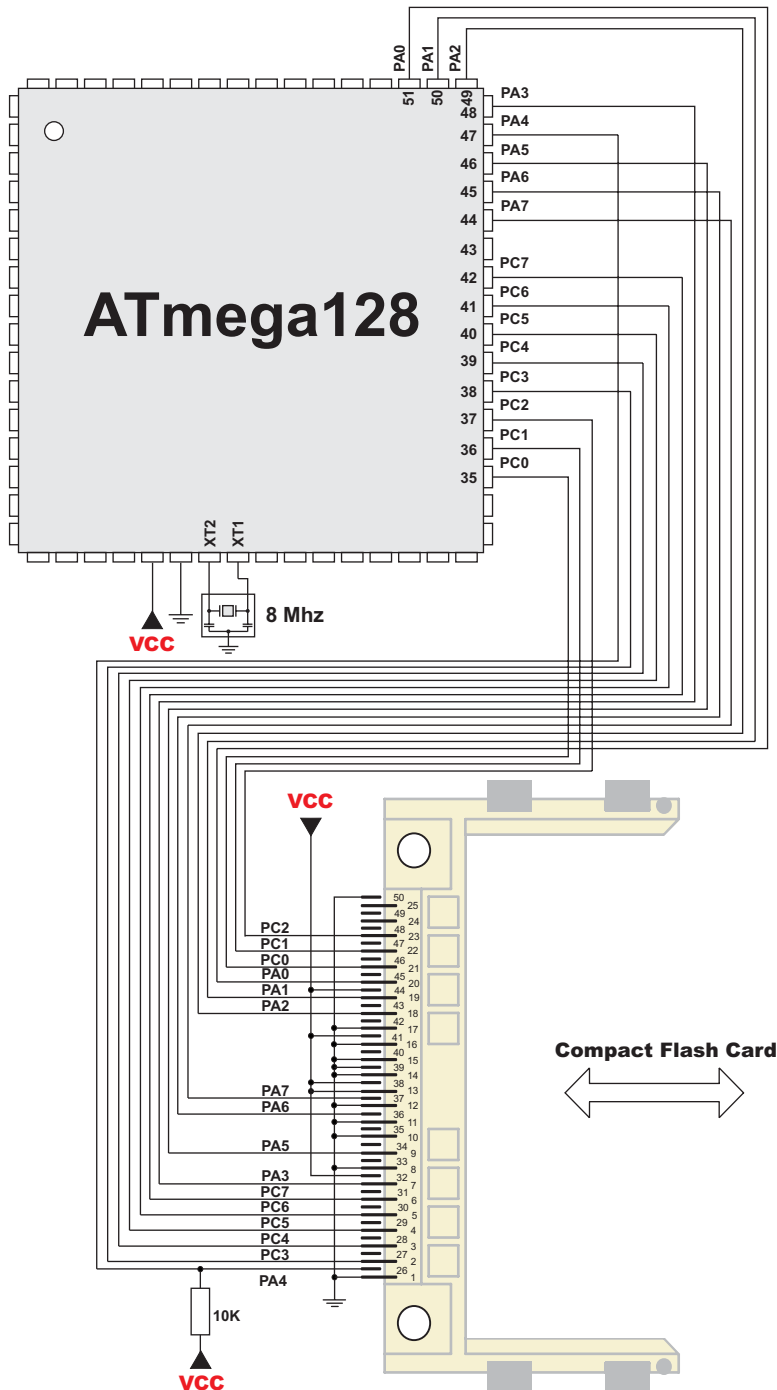
begin
  DDRD := $FF;                                // PORTD is output
  Cf_Init(PORTA,6,5,2,1,0,7,3,4, PORTC);    // Initialize ports

  repeat nop;
  until Cf_Detect();                          // Wait until CF card is inserted

  Delay_ms(500);
  Cf_Write_Init(590, 1);                      // Initialize write at sector address 590
                                           //      of 1 sector (512 bytes)
  for i:= 0 to 511 do                        // Write 512 bytes to sector (590)
    Cf_Write_Byte(i + 11);

  PORTD := $FF;
  Delay_ms(1000);
  Cf_Read_Init(590, 1);                      // Initialize write at sector address 590
                                           //      of 1 sector (512 bytes)
  for i := 0 to 511 do                      // Read 512 bytes from sector (590)
    begin
      PORTD := Cf_Read_Byte;                // Read byte and display on PORTC
      Delay_ms(1000);
    end;
end.
```

HW Connection



EEPROM Library

EEPROM data memory is available with a number of AVR micros. mikroPascal includes library for comfortable work with EEPROM.

Library Routines

Eeprom_Read
Eeprom_Write

Eeprom_Read

Prototype	function Eeprom_Read(address : byte) : 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 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	tmp := Eeprom_Read(\$3F);

Eeprom_Write

Prototype	procedure Eeprom_Write(address, data : byte);
Description	<p>Writes data to specified address. Parameter address is of byte type, which means it can address only 256 locations.</p> <p>Be aware that all interrupts will be disabled during execution of Eeprom_Write routine (I bit of SREG register will be cleared). Routine will set this bit on exit.</p>
Requires	<p>Requires EEPROM unit.</p> <p>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.</p>
Example	Eeprom_Write(\$32);

Library Example

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

```

program Eeprom_Test;
var i, j : byte;

begin
    DDRB := 0;
    for i := 0 to 20 do
        Eeprom_Write(i, i + 6);

    for i := 0 to 20 do
        begin
            PORTB := Eeprom_Read(i);
            Delay_ms(300);
        end;
    end.

```

SPI Ethernet Library

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

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

This library is designed to simplify handling of the underlying hardware (ENC28J60). It works with any AVR with integrated SPI and more than 4 Kb ROM memory. This library is tested with ATMEGA16@8Mhz.

Library Routines

```
ENC28J60_Init  
ENC28J60_doPacket  
ENC28J60_putByte  
ENC28J60_getByte  
ENC28J60_UserTCP  
ENC28J60_UserUDP
```

ENC28J60_Init

Prototype	procedure ENC28J60_Init(var resetPort : byte ; resetBit : byte ; var CSportPtr : byte ; CSbit : byte ; var mac : array [6] of byte ; var ip : array [4] of byte ; fullDuplex : byte);
Returns	Nothing.
Description	Initialize SPI & ENC controller. This function is splited into 2 parts to help linker when coming short of memory. resetPort - pointer to reset pin port resetBit - reset bit number on resetPort CSport - pointer to CS pin port CSbit - CS bit number on CSport mac - pointer to array of 6 char with MAC address ip - pointer to array of 4 char with IP address fullDuplex - either ENC28J60_HALFDUPLEX for half duplex or ENC28J60_FULLDUPLEX for full duplex
Requires	Nothing.
Example	ENC28J60_Init(PORTB, 0, PORTB, 1, myMacAddr, myIpAddr, ENC28J60_FULLDUPLEX);

ENC28J60_doPacket

Prototype	procedure ENC28J60_doPacket;
Returns	Nothing.
Description	Process one incoming packet if available. This function is public.
Requires	ENC28J60_init must have been called before this function must be called as often as possible by use.
Example	ENC28J60_doPacket;

ENC28J60_putByte

Prototype	procedure ENC28J60_putByte(v : byte);
Returns	Nothing.
Description	v - value to store Store one byte to current EWRPT ENC location this function is public.
Requires	ENC28J60_init must have been called before calling this function.
Example	ENC28J60_putByte(0xa0);

ENC28J60_getByte

Prototype	function ENC28J60_getByte : byte ;
Returns	Value of byte @ addr.
Description	Get next byte from current ERDPT ENC location this function is public.
Requires	ENC28J60_init must have been called before calling this function.
Example	b := ENC28J60_getByte;

ENC28J60_UserTCP

Prototype	function ENC28J60_UserTCP(var remoteHost : array [4] of byte ; remotePort, localPort, reqLength : word) : word ;
Returns	Returns the length in bytes of the HTTP reply, or 0 if nothing to transmit.
Description	This function is called by the library. The user accesses to the HTTP request by successive calls to ENC28J60_getByte the user puts data in the transmit buffer by successive calls to ENC28J60_putByte the function must return the length in bytes of the HTTP reply, or 0 if nothing to transmit. If you don't need to reply to HTTP requests, just define this function with a return(0) as single statement.
Requires	ENC28J60_init must have been called before calling this function.
Example	

ENC28J60_UserUDP

Prototype	function ENC28J60_UserUDP(var remoteHost : array [4] of byte ; remotePort, destPort, reqLength : word) : word ;
Returns	Returns the length in bytes of the UDP reply, or 0 if nothing to transmit.
Description	This function is called by the library. The user accesses to the UDP request by successive calls to ENC28J60_getByte. The user puts data in the transmit buffer by successive calls to ENC28J60_putByte. The function must return the length in bytes of the UDP reply, or 0 if nothing to transmit. If you don't need to reply to UDP requests, just define this function with a return(0) as single statement.
Requires	ENC28J60_init must have been called before calling this function.
Example	

Library Example

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

```
program enc_ethernet;

uses enc_eth;

var myMacAddr    : array[ 6] of byte;
    myIpAddr     : array[ 4] of byte;

begin
    DDRB  := DDRB and $A0; // seting to input direction registers
    DDRB  := DDRB or  $BF; // seting to output direction registers
    PORTB := 0;

    DDRC  := 0x00;
    PORTC := 0;
    DDRD  := 0xFF;
    PORTD := 0;

    httpCounter := 0;

    myMacAddr[ 0] := 0x00;
    myMacAddr[ 1] := 0x14;
    myMacAddr[ 2] := 0xA5;           // set MAC address
    myMacAddr[ 3] := 0x76;
    myMacAddr[ 4] := 0x19;
    myMacAddr[ 5] := 0x3F;

    myIpAddr[ 0] := 192;
    myIpAddr[ 1] := 168;           // set IP address
    myIpAddr[ 2] := 20;
    myIpAddr[ 3] := 60;

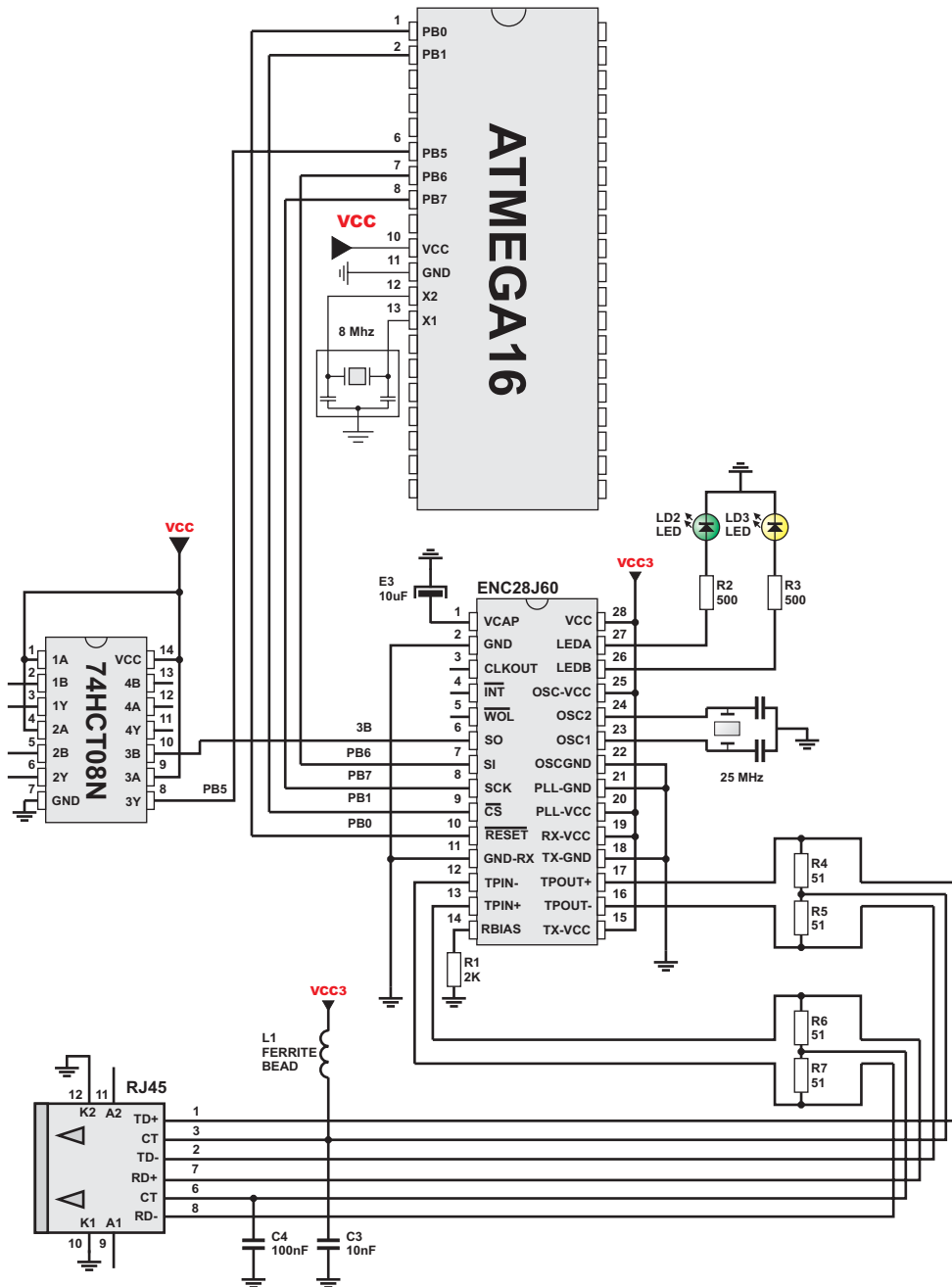
    // continues...
```

```
// continued...

ENC28J60_Init(PORTB, 0, PORTB, 1, myMacAddr, myIpAddr, ENC28J60_FULLDUPLEX) ;

while true do                                // do forever
  begin
    ENC28J60_doPacket();                      // process incoming Ethernet packets
    {*
      * add your stuff here if needed
      * ENC28J60_doPacket() must be called as often as possible
      * otherwise packets could be lost
    *}
  end;
end.
```

HW Connection



Flash Memory Library

This library provides routines for accessing microcontroller Flash memory.

Library Routines

Flash_Read
Flash_Write (Not implemented)

Flash_Read

Prototype	function Flash_Read(address : word) : byte;
Returns	Returns data byte from Flash memory.
Description	Reads data from the specified address in Flash memory.
Example	tmp := Flash_Read(\$0D00);

Flash_Write (Not Impemented)

Prototype	procedure Flash_Write(address, data : word);
Description	Writes chunk of data to Flash memory. Keep in mind that this function erases target memory before writing Data to it. This means that if write was unsuccessful, previous data will be lost.
Example	<pre>// Write consecutive values in 64 consecutive locations for i := 0 to 63 do toWrite[i] := i; // Write contents of the array to the address \$0D00: Flash_Write(\$0D00, toWrite);</pre>

Library Example

The following code demonstrates use of Flash Memory library routines.

```
program flash_avr;

const FLASH_ERROR : byte = $FF;
      FLASH_OK      : byte = $AA;

var toRead, i : byte;
    toWrite : array[64] of byte;

begin
  DDRB := 0;                                // PORTB is output

  PORTB := 0;                                // Turn off PORTB
  toRead := FLASH_ERROR;                     // Initialize error state

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

TWI (I2C) Library

TWI(I2C) full master TWI unit is available with a number of AVR MCU models. mikroPascal provides TWI library which supports the master TWI mode.

Note: Examples for AVR micros with unit on other ports can be found in your mikroPascal installation folder, subfolder “Examples”.

Library Routines

```
Twi_Init
Twi_Busy
Twi_Start
Twi_Stop
Twi_Read
Twi_Write
Twi_Status
Twi_Close
```

Twi_Init

Prototype	procedure Twi_Init(const clock : 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 TWI.
Example	Twi_Init(100000);

Twi_Busy

Prototype	function Twi_Busy : byte ;
Returns	Returns 1 if TWI start sequence is finished, 0 if TWI start sequence is not finished.
Description	Issues repeated START signal.
Requires	TWI must be configured before using this function. See Twi_Init.
Example	if (Twi_Busy = 1) then ...

Twi_Start

Prototype	function Twi_Start : byte;
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	<pre>if Twi_Start = 0 then ...</pre>

Twi_Read

Prototype	function Twi_Read(ack : byte) : 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	<pre>tmp := Twi_Read(0); // Read data and send not acknowledge signal</pre>

Twi_Write

Prototype	function Twi_Write(data : byte) : 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	<pre>TWI_Write(\$A3);</pre>

Twi_Stop

Prototype	<code>procedure Twi_Stop;</code>
Description	Issues STOP signal.
Requires	TWI must be configured before using this function. See <code>Twi_Init</code> .

Twi_Status

Prototype	<code>function Twi_Status: byte;</code>
Returns	Returns value of status register.
Description	Returns status of TWI.
Requires	TWI must be configured before using this function. See <code>Twi_Init</code> .
Example	<code>status := Twi_Status;</code>

Twi_Close

Prototype	<code>procedure Twi_Close;</code>
Returns	Nothing.
Description	Closes TWI connection.
Requires	TWI must be configured before using this function. See <code>Twi_Init</code> .
Example	<code>Twi_Close;</code>

Library Example

This code demonstrates use of TWI Library procedures and functions. 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 TWI from EEPROM and send its value to PORTD, to check if the cycle was successful. Check the figure below.

```

program Eeprom_Test;

var   EE_adr, EE_data, k : byte;
      jj : word;

begin
  Twi_Init(100000);      // Initialize full master mode
  DDRD := 0;             // PORTD is output
  PORTD := $FF;          // Initialize PORTD
  Twi_Start;             // Issue TWI start signal
  Twi_Write($A2);        // Send byte via TWI(command to 24c02)
  EE_adr := 2;
  Twi_Write(EE_adr);      // Send byte(address for EEPROM)
  EE_data := $AA;
  Twi_Write(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 do nop;

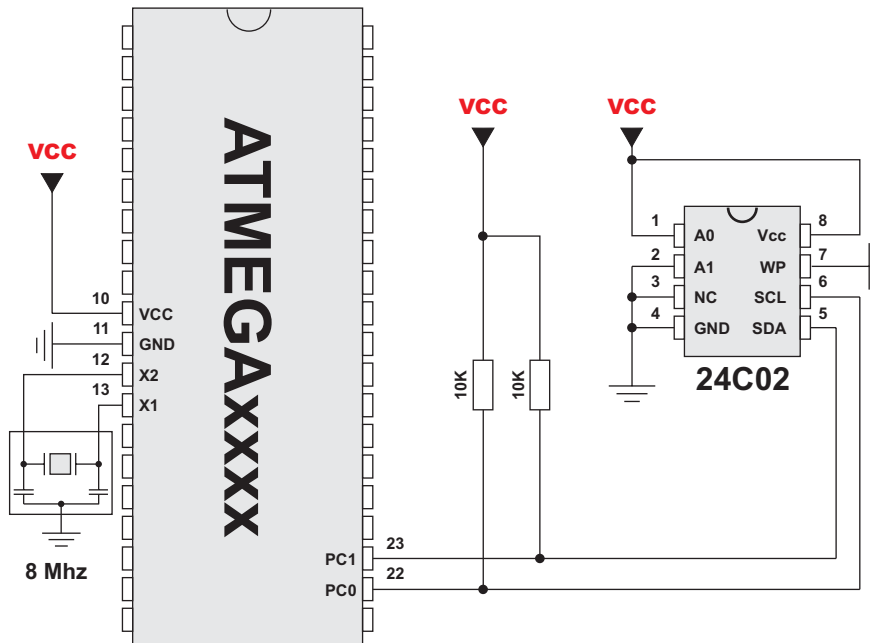
  Twi_Start;             // Issue TWI start signal
  Twi_Write($A2);        // Send byte via TWI
  EE_adr := 2;
  Twi_Write(EE_adr);     // Send byte(address for EEPROM)
  Twi_Busy; // Issue TWI signal repeated start
  Twi_Write($A3);        // Send byte (request data from EEPROM)
  k := Twi_Read;         // Read the data
  Twi_Stop;              // Issue TWI stop signal
  PORTD := k;            // Show data on PORTD

  // Endless loop
  while true do nop;

end.

```

HW Connection



Keypad Library

mikroPascal 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	procedure Keypad_Init(var port : 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	function Keypad_Read : 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	function Keypad_Released : word;
Returns	1..16, depending on the key.
Description	Call to Keypad_Released is a blocking call: function waits until any key is pressed and released. When released, function returns 1 to 16, depending on the key.
Requires	Port needs to be appropriately initialized; see Keypad_Init.
Example	kp := Keypad_Released();

Library Example

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

```

program keypad_test;

var kp, cnt : byte;
    txt : string[ 5];

begin
    cnt := 0;
    Keypad_Init(PORTB);
    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:');

    repeat
    begin
        kp := 0;

        //--- Wait for key to be pressed
        while kp = 0 do
        begin
            //--- un-comment one of the keypad reading functions
            kp := Keypad_Released();
            //kp := Keypad_Read();
        end;

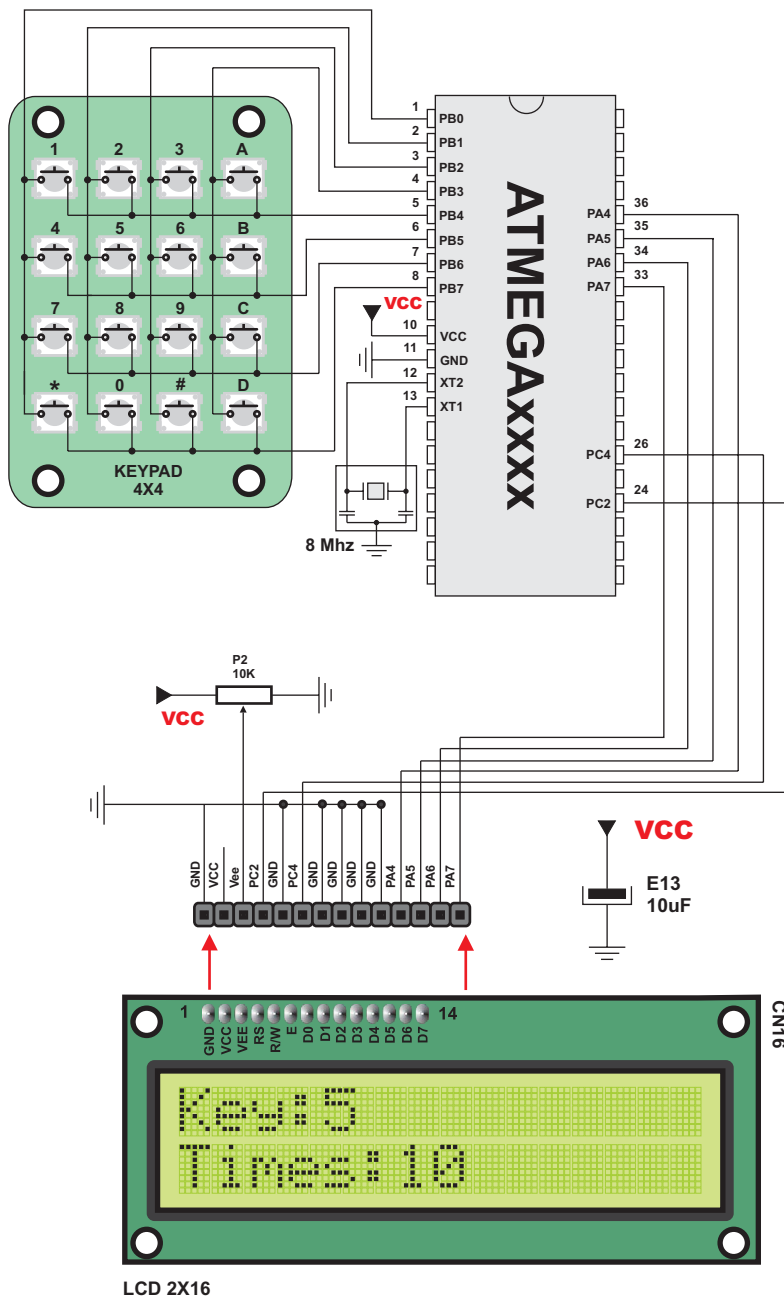
        Inc(cnt);

        //--- prepare value for output
        if kp > 10 then
            kp := kp + 54
        else
            kp := kp + 47;

        //--- print it on LCD
        Lcd_Chrc(1, 10, kp);
        WordToStr(cnt, txt);
        Lcd_Out(2, 10, txt);
    end;
    until FALSE;
end.

```

HW Connection



LCD Library (4-bit interface)

mikroPascal 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	procedure Lcd_Init(var control_port : byte; EN, RS : byte; var data_port : byte; nibble : 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	procedure Lcd_Out(row, col : byte; var text : string[21]);
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	procedure Lcd_Out_Cp(var text : string[21]);
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. Lcd_Init.
Example	Lcd_Out_Cp('Here!'); // Print "Here!" at current cursor position

Lcd_Chr

Prototype	procedure Lcd_Chr(row, col, character : byte);
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	Port with LCD must be initialized. See <code>Lcd_Init</code> .
Example	<code>Lcd_Chr(2, 3, 'i');</code> // Print "i" at line 2, char 3

Lcd_Chr_Cp

Prototype	procedure Lcd_Chr_Cp(character : byte);
Description	Prints <code>character</code> on LCD at current cursor position. Both variables and literals can be passed as <code>character</code> .
Requires	Port with LCD must be initialized. See <code>Lcd_Init</code> .
Example	<code>Lcd_Chr_Cp('e');</code> // Print "e" at current cursor position

Lcd_Cmd

Prototype	procedure Lcd_Cmd(command : byte);
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 shown on the page 140.
Requires	Port with LCD must be initialized. See <code>Lcd_Init</code> .
Example	<code>Lcd_Cmd(LCD_Clear);</code> // Clear LCD display

LCD Commands

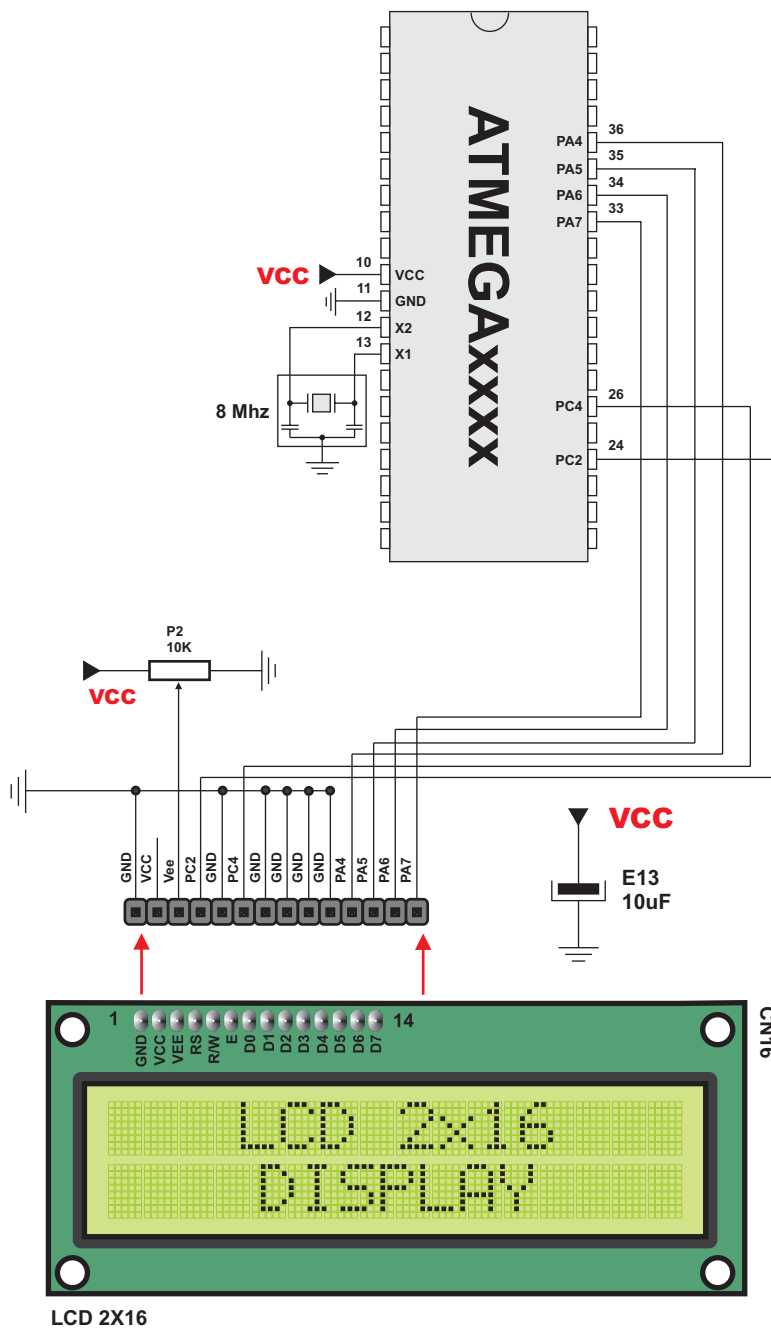
LCD Command	Purpose
LCD_FIRST_ROW	Move cursor to 1st row
LCD_SECOND_ROW	Move cursor to 2nd row
LCD_THIRD_ROW	Move cursor to 3rd row
LCD_FOURTH_ROW	Move cursor to 4th row
LCD_CLEAR	Clear display
LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to original position. Display data 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;
var text: array[20] of char;

begin
  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)

mikroPascal 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	procedure Lcd8_Init(var control_port : byte; EN, RS : byte; var data_port : 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);

Lcd8_Out

Prototype	procedure Lcd8_Out(row, col : byte; var text : string[21]);
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	procedure Lcd8_Out_Cp(var text : string[21]);
Description	Prints text on LCD at current cursor position. Both string variables and literals can be passed as text.
Requires	Ports with LCD must be initialized. See Lcd8_Config or Lcd8_Init.
Example	Lcd8_Out_Cp('Here!'); // Print "Here!" at current cursor position

Lcd8_Chr

Prototype	procedure Lcd8_Chr(row, col, character : byte);
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');</code> <i>// Print "i" at line 2, char 3</i>

Lcd8_Chr_Cp

Prototype	procedure Lcd8_Chr_Cp(character : byte);
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');</code> <i>// Print "e" at current cursor position</i>

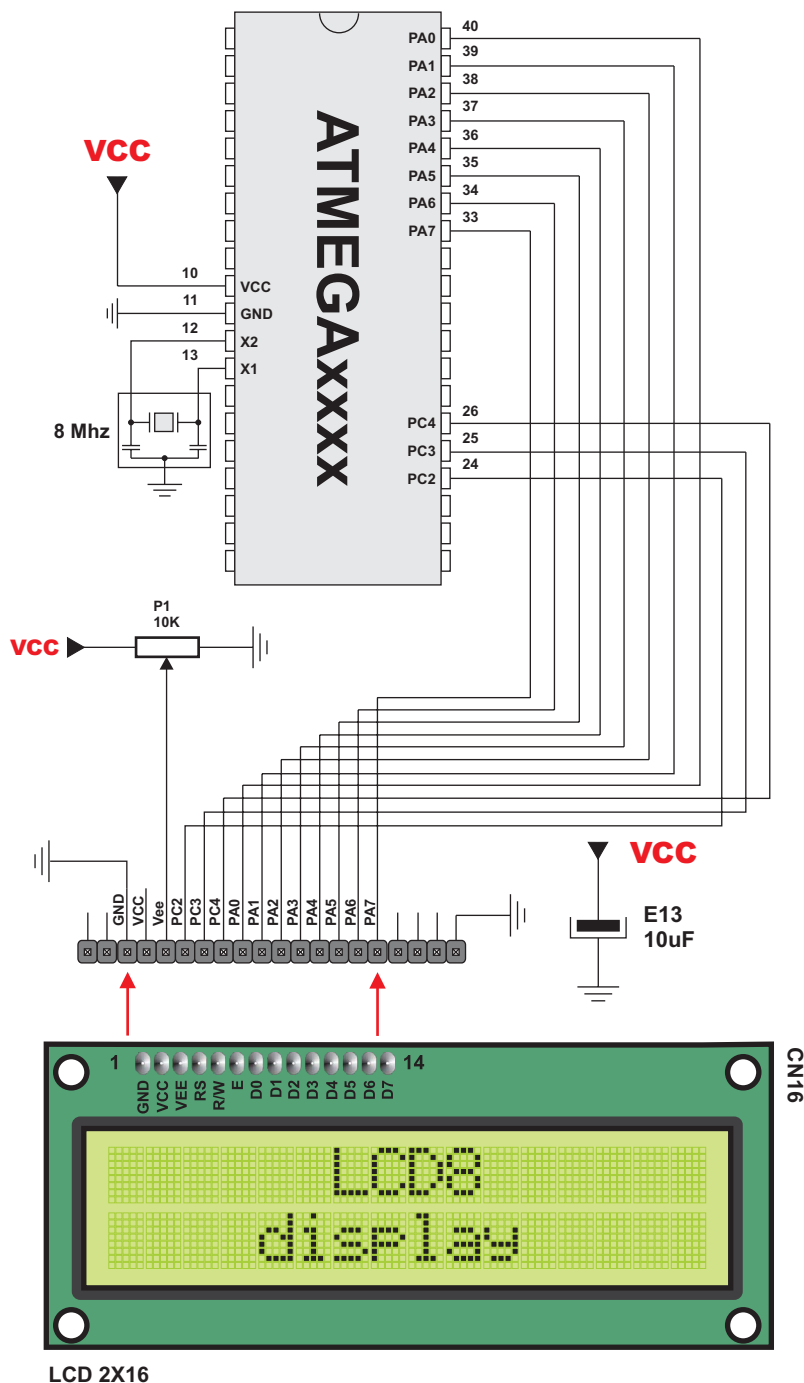
Lcd8_Cmd

Prototype	procedure Lcd8_Cmd(command : byte);
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);</code> <i>// Clear LCD display</i>

Library Example

```
program Lcd8_default_test;  
begin  
    DDRC := DDRC or $10;  
    PORTC := PINC and $EF;      // for putting the reset line on GND  
    Lcd8_Init(PORTC, 4, 2, PORTA);  
    Lcd8_Out(2,1,'mikroElektronika');  
end.
```


Hardware Connection



GLCD Library

mikroPascal 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.

Basic routines:

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_Init

Prototype	procedure Glcd_Init(var ctrlport : byte; cs1, cs2, rs, rw, rst, en : byte; var dataport : byte);
Description	<p>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.</p> <p>This procedure needs to be called before using other routines of GLCD library.</p>
Example	Glcd_Init(PORTB, 2, 0, 3, 5, 7, 1, PORTC);

Glcd_Disable

Prototype	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	procedure Glcd_Set_Side(x : byte);
Description	<p>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.</p>
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Set_Side(0);

Glcd_Set_Page

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

Glcd_Set_X

Prototype	procedure Glcd_Set_X(x : byte);
Description	Positions to x dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Set_X(25);

Glcd_Read_Data

Prototype	function Glcd_Read_Data : byte;
Returns	One word from the GLCD memory.
Description	Reads data from from the current location of GLCD memory. Use the functions Glcd_Set_Side, Glcd_Set_X, and Glcd_Set_Page to specify an exact position on GLCD. Then, you can use Glcd_Write_Data or Glcd_Read_Data on that location.
Requires	Reads data from from the current location of GLCD memory.
Example	tmp := Glcd_Read_Data();

Glcd_Write_Data

Prototype	procedure Glcd_Write_Data(data : 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	procedure Glcd_Fill(pattern : 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); <i>// Clear screen</i>

Glcd_Dot

Prototype	procedure Glcd_Dot(x, y, color : 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); <i>// Invert the dot in the upper left corner</i>

Glcd_Line

Prototype	procedure Glcd_Line(x1, y1, x2, y2, color : 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	procedure Glcd_V_Line(y1, y2, x, color : 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	procedure Glcd_H_Line(x1, x2, y, color : 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	procedure Glcd_Rectangle(x1, y1, x2, y2, color : 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	procedure Glcd_Box(x1, y1, x2, y2, color : 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	procedure Glcd_Circle(x, y, radius, color : 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, 1);

Glcd_Set_Font

Prototype	procedure Glcd_Set_Font(font_address : longint; font_width, font_height : byte; font_offset : 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	<pre>// Use the custom 5x7 font "myfont" which starts with space (32): Glcd_Set_Font(@myfont, 5, 7, 32);</pre>

Glcd_Write_Char

Prototype	procedure Glcd_Write_Char(character, x, page, color : 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	<pre>Glcd_Write_Char('C', 0, 0, 1);</pre>

Glcd_Write_Text

Prototype	procedure Glcd_Write_Text(text : string [20] ; x, page, color : byte);
Description	<p>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).</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_Text('Hello world!', 0, 0, 1);

Glcd_Image

Prototype	procedure Glcd_Image(image : array [0..1023] of byte);
Description	Displays bitmap image on the GLCD. Parameter image should be formatted as an array of 1024 bytes. Use the mikroPascal’s integrated Bitmap-to-LCD editor (menu option Tools > Graphic LCD Editor) to convert image to a constant array suitable for display on GLCD.
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Image(my_image);

Library Example

The following drawing demo tests advanced routines of GLCD library.

```

program Glcd_Test;
var j, k : byte;

begin

    Glcd_Init(PORTC, 0, 1, 2, 3, 5, 4, PORTA);

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

    repeat
        begin
            // Draw my image
            Glcd_Image(mikro_logo_bmp);
            Delay_ms(4000);

            // Draw circles
            Glcd_Fill(0); // Clear screen
            Glcd_Write_Text('Circles', 0, 0, 1);

            j := 4;
            while j < 31 do
                begin
                    Glcd_Circle(63, 31, j, 2);
                    j := j + 4;
                end;
            Delay_ms(4000);

            // Draw boxes
            Glcd_Fill(0); // Clear screen
            Glcd_Write_Text('Rectangles', 0, 0, 1);
            j := 0;
            while j < 31 do
                begin
                    Glcd_Box(j, 0, j + 20, j + 25, 2);
                    j := j + 4;
                end;
            Delay_ms(4000);

        end
    until FALSE;

end.

```

[illegible]

T6963C Graphic LCD Library

mikroPascal provides a library for drawing and writing on Toshiba T6963C Graphic LCD (changeable size).

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

Library Routines

```
T6963C_init  
T6963C_writeData  
T6963C_writeCommand  
T6963C_setPtr  
T6963C_waitReady  
T6963C_fill  
T6963C_dot  
T6963C_write_char  
T6963C_write_text  
T6963C_line  
T6963C_rectangle  
T6963C_box  
T6963C_circle  
T6963C_image  
T6963C_sprite  
T6963C_set_cursor
```

T6963C_init

Prototype	procedure T6963C_init(width, height, fntW : word ; var data : word ; var cntrl : word ; wr, rd, cd, rst : word);
Description	<p>Initializes the Graphic Lcd controller. This function must be called before all T6963C Library Routines.</p> <p>width - Number of horizontal (x) pixels in the display.</p> <p>height - Number of vertical (y) pixels in the display.</p> <p>fntW - Font width, number of pixels in a text character, must be set accordingly to the hardware.</p> <p>data - Address of the port on which the Data Bus is connected.</p> <p>cntrl - Address of the port on which the Control Bus is connected.</p> <p>wr - !WR line bit number in the *cntrl port.</p> <p>rd - !RD line bit number in the *cntrl port.</p> <p>cd - !CD line bit number in the *cntrl port.</p> <p>rst - !RST line bit number in the *cntrl port.</p> <p>Display RAM :</p> <p>The library doesn't know the amount of available RAM.</p> <p>The library cuts the RAM into panels : a complete panel is one graphics panel followed by a text panel, The programer has to know his hardware to know how much panel he has.</p>
Requires	Nothing.
Example	<pre> T6963C_init(240, 128, 8, PORTB, PORTC, 3, 2, 1, 5) ; { * * init display for 240 pixel width and 128 pixel height * 8 bits character width * data bus on PORTF * control bus on PORTD * bit 5 is !WR * bit 7 is !RD * bit 6 is !CD * bit 4 is RST *} </pre>

T6963C_writeData

Prototype	procedure T6963C_writeData(data : byte);
Description	Routine that writes data to T6963C controller.
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_writeData(AddrL);

T6963C_writeCommand

Prototype	procedure T6963C_writeCommand(data : byte);
Description	Routine that writes command to T6963C controller.
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_writeCommand(T6963C_CURSOR_POINTER_SET);

T6963C_setPtr

Prototype	procedure T6963C_setPtr(p : word ; c : byte);
Description	This routine sets the memory pointer p for command c.
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_writeCommand(T6963C_CURSOR_POINTER_SET);

T6963C_waitReady

Prototype	procedure T6963C_waitReady;
Description	This routine pools the status byte, and loops until ready.
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_waitReady;

T6963C_fill

Prototype	procedure T6963C_fill(v : byte ; start , len : word);
Description	This routine fills length with bytes to controller memory from start address.
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_fill(0x33,0x00FF,0x000F);

T6963C_dot

Prototype	procedure T6963C_dot(x , y : integer ; color : byte);
Description	This sets current text work panel. It writes string str row x line y. mode = T6963C_ROM_MODE_[OR EXOR AND].
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_dot(x0, y0, pcolor);

T6963C_write_char

Prototype	procedure T6963C_write_char(c , x , y , mode : byte);
Description	This routine sets current text work panel. It writes char c row x line y. mode = T6963C_ROM_MODE_[OR EXOR AND]
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_write_char('A',22,23,AND);

T6963C_write_text

Prototype	procedure T6963C_write_text(var str : array [10] of byte ; x , y , mode : byte);
Description	This sets current text work panel. It writes string str row x line y. mode = T6963C_ROM_MODE_[OR EXOR AND]
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_write_text("GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR);

T6963C_line

Prototype	procedure T6963C_line(x0, y0, x1, y1 : integer ; pcolor : byte);
Description	This routine current graphic work panel. It's draw a line from (x0, y0) to (x1, y1). pcolor = T6963C_[WHITE[BLACK]
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_line(0, 0, 239, 127, T6963C_WHITE);

T6963C_rectangle

Prototype	procedure T6963C_rectangle(x0, y0, x1, y1 : integer ; pcolor : byte);
Description	It sets current graphic work panel. It draws the border of the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE);

T6963C_box

Prototype	procedure T6963C_box(x0, y0, x1, y1 : integer ; pcolor : byte);
Description	This routine sets current graphic work panel. It draws a solid box in the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_box(0, 119, 239, 127, T6963C_WHITE);

T6963C_circle

Prototype	procedure T6963C_circle(x, y : integer ; r : longint ; pcolor : word);
Description	This routine sets current graphic work panel. It draws a circle, center is (x, y), diameter is r. pcolor = T6963C_[WHITE[BLACK]
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_circle(120, 64, 110, T6963C_WHITE);

T6963C_image

Prototype	procedure T6963C_image(const pic : ^ byte);
Description	This routine sets current graphic work panel : It fills graphic area with picture pointer by MCU. MCU must fit the display geometry. For example : for a 240x128 display, MCU must be an array of (240/8)*128 = 3840 bytes .
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_image(my_image);

T6963C_sprite

Prototype	procedure T6963C_sprite(px, py : byte ; const pic : ^ byte ; sx, sy : byte);
Description	This routine sets current graphic work panel. It fills graphic rectangle area (px, py)-(px + sx, py + sy) witch picture pointed by MCU. Sx and sy must be the size of the picture. MCU must be an array of sx*sy bytes.
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_sprite(76, 4, einstein, 88, 119); <i>// draw a sprite</i>

T6963C_set_cursor

Prototype	void T6963C_set_cursor(unsigned char x, unsigned char y);
Description	This routine sets cursor row x line y.
Requires	Ports must be initialized. See T6963C_init.
Example	T6963C_set_cursor(cposx, cposy);

Library Example

The following drawing demo tests advanced routines of T6963C GLCD library.

```

program T6963_test;

uses    T6963Clib, bitmap, bitmap2;

var     panel : byte;      // current panel
          i : word;         // general purpose register
          curs : byte;      // cursor visibility
          cposx,
          cposy : word;     // cursor x-y position

procedure wait;
begin
    delay_ms(2000);
end;

begin

    { *
      * init display for 240 pixel width and 128 pixel height
      * 8 bits character width
      * data bus on PORTF
      * control bus on PORTD
      * bit 5 is !WR
      * bit 7 is !RD
      * bit 6 is !CD
      * bit 4 is RST
      * }

    T6963C_init(240, 128, 8, PORTB, PORTC, 3, 2, 1, 5) ;
    T6963C_graphics(1) ;
    T6963C_text(1) ;

    //continues...

```

```
//continued...

wait;

{*
 * draw circles
*}

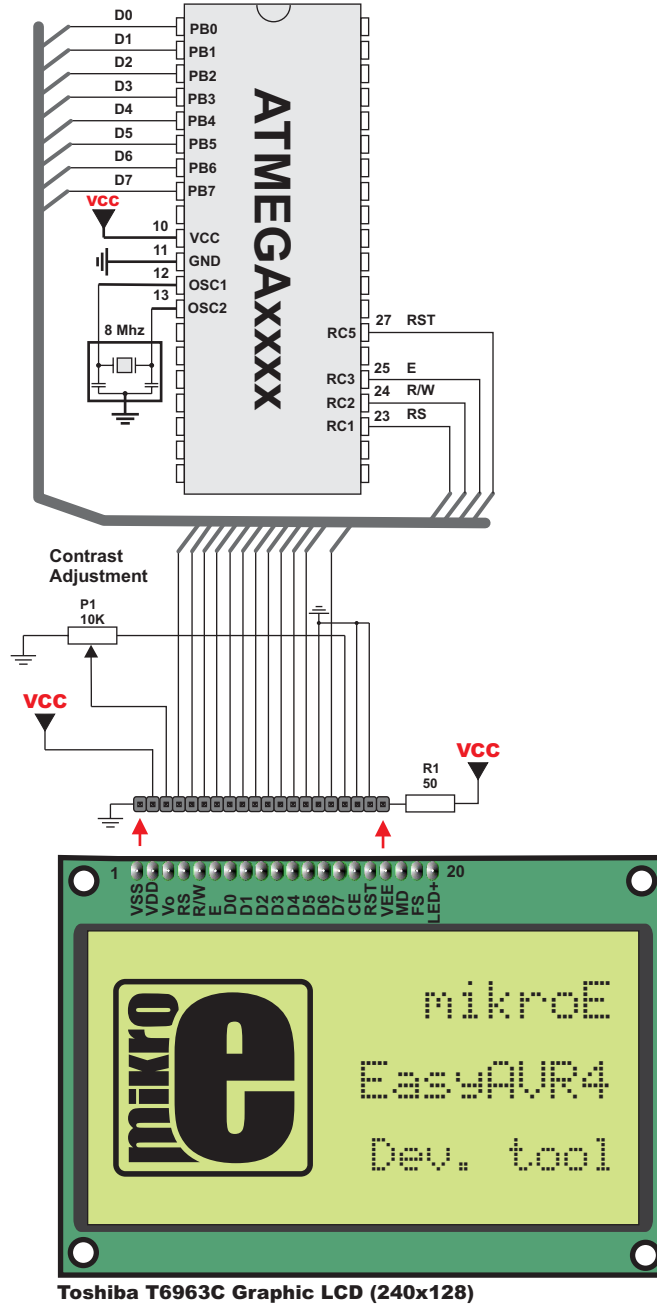
T6963C_circle(120, 64, 30, T6963C_WHITE) ;
T6963C_circle(120, 64, 50, T6963C_WHITE) ;
T6963C_circle(120, 64, 70, T6963C_WHITE) ;
T6963C_circle(120, 64, 90, T6963C_WHITE) ;
T6963C_circle(120, 64, 110, T6963C_WHITE) ;
T6963C_circle(120, 64, 130, T6963C_WHITE) ;
delay_ms(1000);

T6963C_sprite(76, 4, einstein, 88, 119) ;
// draw a sprite
delay_ms(1000);

T6963C_setGrPanel(1) ;           // select other graphic panel
delay_ms(1000);

T6963C_image(mikroPascal_logo_glcd_bmp)
// fill the graphic screen with a picture
end.
```

Hardware Connection



Multi Media Card Library

The Multi Media Card (MMC) is a flash memory card standard. MMC cards are currently available in sizes up to and including 1 GB, and are used in cell phones, mp3 players, digital cameras, and PDA's.

mikroPascal provides a library for accessing data on Multi Media Card via SPI communication. This library also supports SD(Secure Digital) memory cards.

Secure Digital Card

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

Notes:

- Library functions create and read files from the root directory only;
- Library functions populate both FAT1 and FAT2 tables when writing to files, but the file data is being read from the FAT1 table only; i.e. there is no recovery if FAT1 table is corrupted.

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_Fat_Get_Swap_File
```

Mmc_Init

Prototype	function Mmc_Init(var port : byte; pin : byte): byte;
Returns	Returns 0 if MMC card is present and successfully initialized, otherwise returns 1.
Description	Initializes hardware SPI communication; parameters port and pin designate the CS line used in the communication (parameter pin should be 0..7). The function returns 0 if MMC card is present and successfully initialized, otherwise returns 1. Mmc_Init needs to be called before using other functions of this library.
Example	<code>error := Mmc_Init(PORTD, 6); // Init with CS line at RC2</code>

Mmc_Read_Sector

Prototype	function Mmc_Read_Sector(sector : longint; var data : array [512] of byte) : byte;
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads one sector (512 bytes) from MMC card at sector address sector. Read data is stored in the array data. Function returns 0 if read was successful, or 1 if an error occurred.
Requires	Library needs to be initialized, see Mmc_Init.
Example	<code>error := Mmc_Read_Sector(sector, data);</code>

Mmc_Write_Sector

Prototype	function Mmc_Write_Sector(sector : longint; var data : array [512] of byte) : 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 <i>sector</i> . 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	<code>error := Mmc_Write_Sector(sector, data);</code>

Mmc_Read_Cid

Prototype	function Mmc_Read_Cid(var data_for_registers : array [512] of byte) : 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 <i>data_for_registers</i> .
Requires	Library needs to be initialized, see Mmc_Init.
Example	<code>error := Mmc_Read_Cid(data);</code>

Mmc_Read_Csd

Prototype	function Mmc_Read_Csd(var data_for_registers : array [512] of byte) : 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	function Mmc_Fat_Init(var mmcport : byte; mmcpin : byte) : byte;
Returns	Returns 0 if MMC card is present and successfully initialized, otherwise returns 1.
Description	<p>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.</p> <p>This function needs to be called before using other functions of MMC FAT library.</p>
Example	success := Mmc_Fat_Init(PORTD, 6);

Mmc_Fat_Assign

Prototype	function Mmc_Fat_Assign(var filename : array [11] of char, create_file : byte) : 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 allways be non-zero Mmc_Fat_Assign('EXAMPLE1TXT', 1); // Assign the file "EXAMPLE2.TXT" in the root directory of MMC. // If the file is not found, routine will NOT create new one. file_found := Mmc_Fat_Assign('EXAMPLE2TXT', 0);</pre>

Mmc_Fat_Delete

Prototype	procedure Mmc_Fat_Delete;
Description	Deletes file from MMC.
Requires	Ports must be initialized for FAT operations with MMC. See <code>Mmc_Fat_Init</code> . File must be assigned. See <code>Mmc_Fat_Assign</code> .
Example	<code>Mmc_Fat_Delete;</code>

Mmc_Fat_Reset

Prototype	procedure Mmc_Fat_Reset(var size : 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	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	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	procedure Mmc_Fat_Read(var data : 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	procedure Mmc_Fat_Write(var fdata : array [256] of char);
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' + #13 + 'world');

Mmc_Fat_Set_File_Date

Prototype	procedure Mmc_Set_File_Date(year : word; month, day, hours, min, sec : 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	procedure Mmc_Fat_Get_File_Date(var year : word; var month, day, hours, min, sec : 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 var yr: word; mnth, dat, hrs, mins: 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	function Mmc_Fat_Get_File_Size : 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 var yr: word; mnth, dat, hrs, mins: byte; ... file_name := "MYFILEXXTXT"; Mmc_Fat_Assign(file_name); cf_size := Mmc_Fat_Get_File_Size();</pre>

Mmc_Fat_Get_Swap_File

Prototype	function Mmc_Fat_Get_Swap_File(sectors_cnt : longint): longint
Returns	No. of start sector for the newly created swap file, if swap file was created; otherwise, the function returns zero.
Description	<p>This function is used to create a swap file on the MMC/SD media. It accepts as sectors_cnt argument the number of consecutive sectors that user wants the swap file to have. During its execution, the function searches for the available consecutive sectors, their number being specified by the sectors_cnt argument. If there is such space on the media, the swap file named MIKROSWP.SYS is created, and that space is designated (in FAT tables) to it. The attributes of this file are: system, archive and hidden, in order to distinct it from other files. If a file named MIKROSWP.SYS already exists on the media, this function deletes it upon creating the new one.</p> <p>The purpose of the swap file is to make reading and writing to MMC/SD media as fast as possible, without potentially damaging the FAT system. Swap file can be considered as a "window" on the media where user can freely write/read the data, in any way (s)he wants to. Its main purpose in mikroPascal library is to be used for fast data acquisition; when the time-critical acquisition has finished, the data can be re-written into a "normal" file, and formatted in the most suitable way.</p>
Requires	Ports must be initialized for FAT operations with MMC. See Mmc_Fat_Init.
Example	<pre>//----- Tries to create a swap file, whose size will be //at least 100 sectors. //If it succeeds, it sends the No. of start sector over USART procedure M_Create_Swap_File; begin size := Mmc_Fat_Get_Swap_File(100); if (size) then begin Usart_Write(\$AA); Usart_Write(Lo(size)); Usart_Write(Hi(size)); Usart_Write(Higher(size)); Usart_Write(Highest(size)); Usart_Write(\$AA); end; end;</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;
var tmp : byte;
    i : word;
    data : array[ 512] of byte;

begin
    Usart_Init(9600);

    // Initialize ports
    tmp := Mmc_Init(PORTC, 2);

    // Fill the buffer with the 'M' character
    for i := 0 to 512 do data[ i] := 'M';

    // Write it to MMC card, sector 55
    tmp := Mmc_Write_Sector(55, data);

    // Fill the buffer with the 'E' character
    for i := 0 to 512 do data[ i] := 'E';

    // Write it to MMC card, sector 56
    tmp := Mmc_Write_Sector(56, data);

    /** Verify: **

    // Read from sector 55
    tmp := Mmc_Read_Sector(55, data);

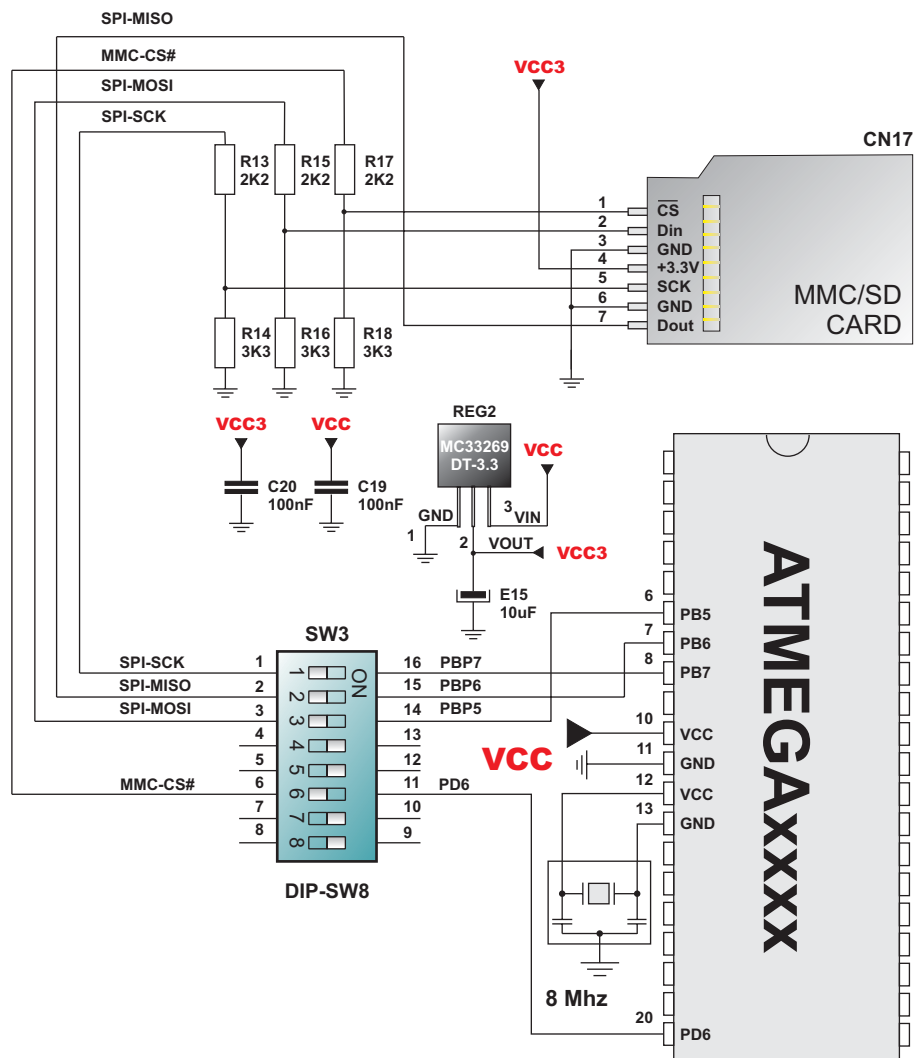
    // Send 512 bytes from buffer to USART
    if tmp = 0 then
        for i := 0 to 512 do Usart_Write(data[ i]);

    // Read from sector 56
    tmp := Mmc_Read_Sector(56, data);

    // Send 512 bytes from buffer to USART
    if tmp = 0 then
        for i := 0 to 512 do Usart_Write(data[ i]);
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	function Ow_Reset(var port : byte; pin : byte) : 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	Nothing.
Example	<code>Ow_Reset(PORTC, 7); // reset DS1820 connected to the PC7 pin</code>

Ow_Read

Prototype	function Ow_Read : 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;</code>

Ow_Write

Prototype	procedure Ow_Write(par : byte);
Description	Writes one byte of data (argument <code>par</code>) via OneWire bus.
Example	<code>Ow_Write(\$CC);</code>

Library Example

The example reads the temperature from DS1820 sensor connected to PC7. Temperature value is continually displayed on LCD.

```

program onewire_test;
var i, j1, j2,tmp_sign: byte;
    text : array[6] of char;
begin
    Usart1_init(9600);
    repeat
        begin
            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;
            if j2 = $FF then
                begin
                    tmp_sign:='-';      // temperature sign
                    j1:= j1 or $FF;     // complement of two
                    j1:= j1 + $01;
                end
            else tmp_sign:='+';

            j2:=(j1 and $01)*5;         // Get decimal value
            j1:=j1 shr 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');

            continues on next page...
        end
    until false;
end

```

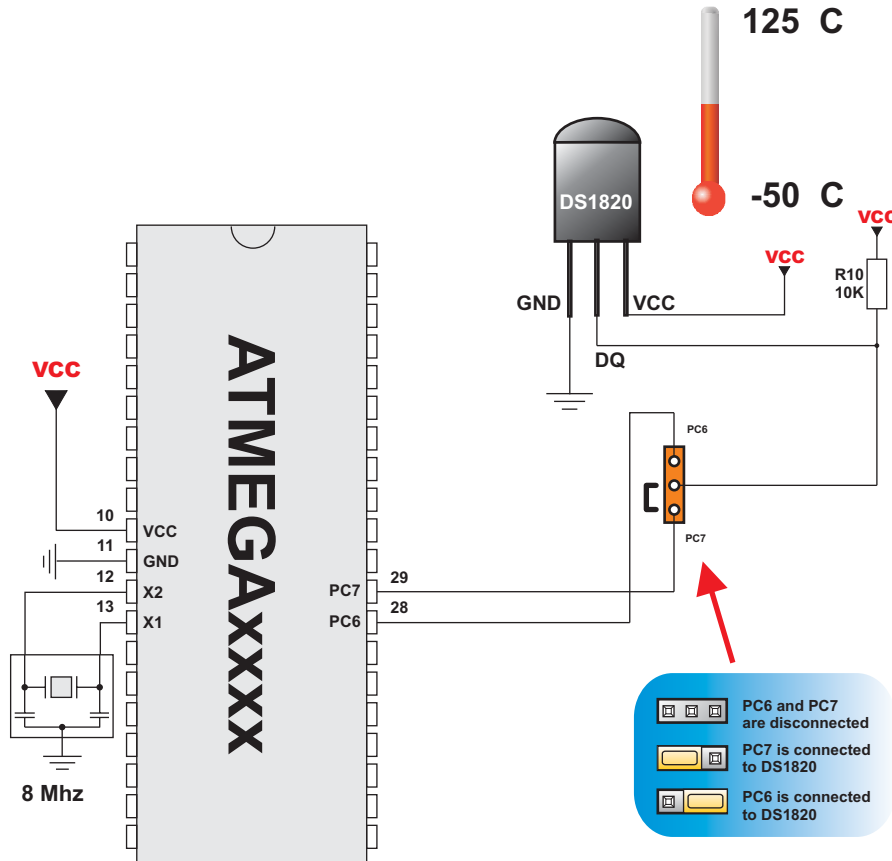
Hardware Connection

```

Usart1_write_char(10);      // next row
Usart1_write_char(14);

Delay_ms(100);
end;
until false;               // endless loop
end.

```



PS/2 Library

mikroPascal 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	procedure Ps2_Init(var port : byte; clock, data : 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 at 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, 5, 6);

Ps2_Key_Read

Prototype	function Ps2_Key_Read(var value, special, pressed : byte) : byte;
Returns	Returns 1 if reading of a key from the keyboard was successful, otherwise 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, value 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, special 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 Ps2_Init.
Example	<pre>// Press Enter to continue: repeat if Ps2_Key_Read(val, spec, press) = 1 then if (val = 13) and (spec = 1) then break; until FALSE;</pre>

Library Example

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

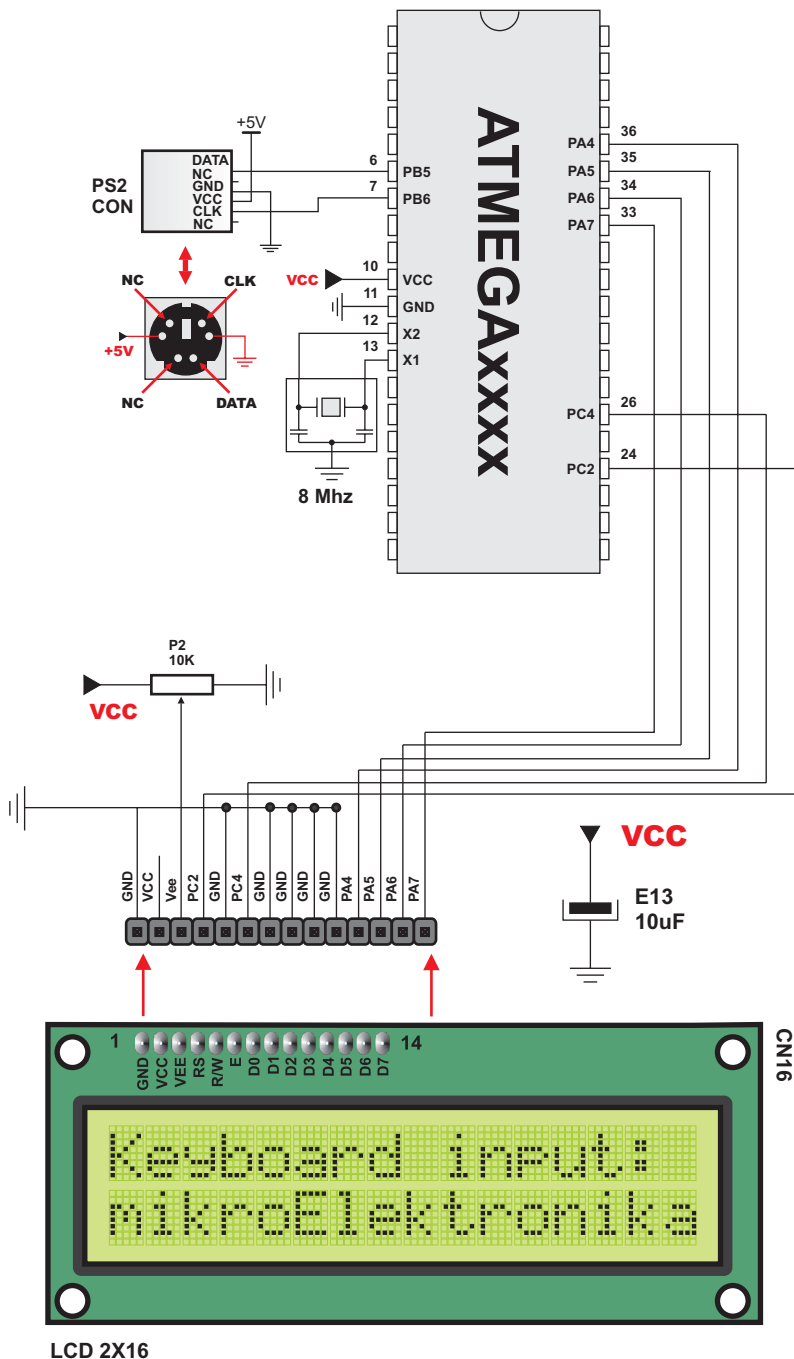
```
program ps2_test;
var keydata, special, down : byte;

begin
  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

  Ps2_Init(PORTB,5,6); // Init PS/2 Keyboard on PORTB
  Delay_ms(100);      // Wait for keyboard to finish

  repeat
  begin
    if Ps2_Key_Read(keydata, special, down) = 1 then
      begin
        if (down = 1) and (special = 0) and (keydata <> 0) then
          Lcd_Out_CP(keydata);
        end;
        Delay_ms(10);    // debounce
      end;
    until FALSE;
  end.
```

Hardware Connection



PWM Library

PWM unit is available with a number of AVR micros. mikroPascal provides library which simplifies using PWM HW Module.

Note: These routines support module on RB0, RB3, RB4 and won't work with modules on other ports. You can find examples for AVR micros with unit on other ports in mikroPascal installation folder, subfolder "Examples".

Library Routines

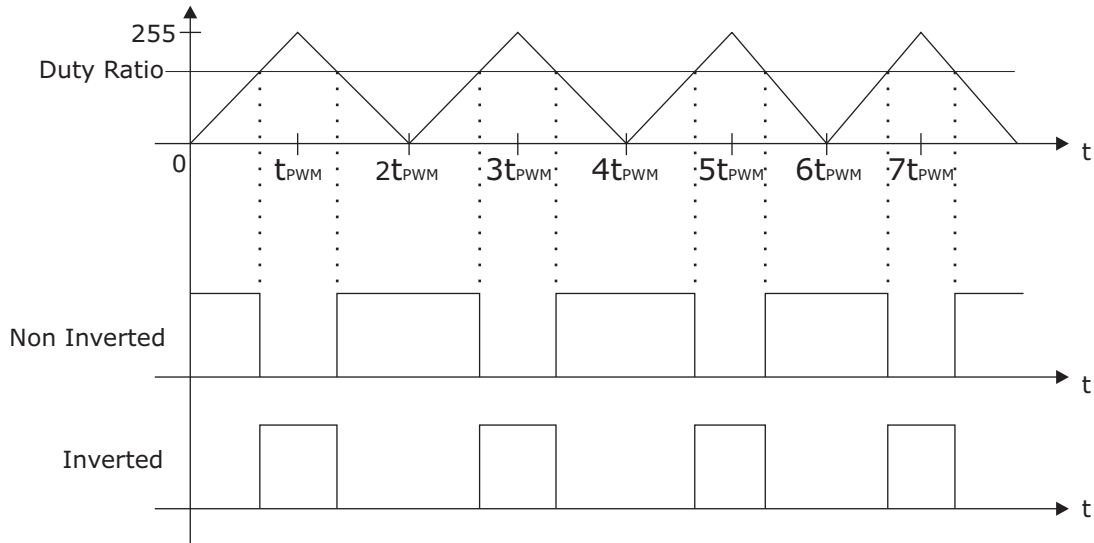
```
Pwm_Init
Pwm_Change_Duty
Pwm_Start
Pwm_Stop
```

Pwm_Init

Prototype	procedure Pwm_Init(wave_mode, prescaler, inverted , duty : 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 mikroPascal installation folder, subfolder "Examples", for alternate solutions.
Example	<p>Initialize PWM module:</p> <pre>Pwm_Init (PWM_PHASE_CORRECT_MODE, PWM_PRESCALER_1024, PWM_NON_INVERTED, duty);</pre>

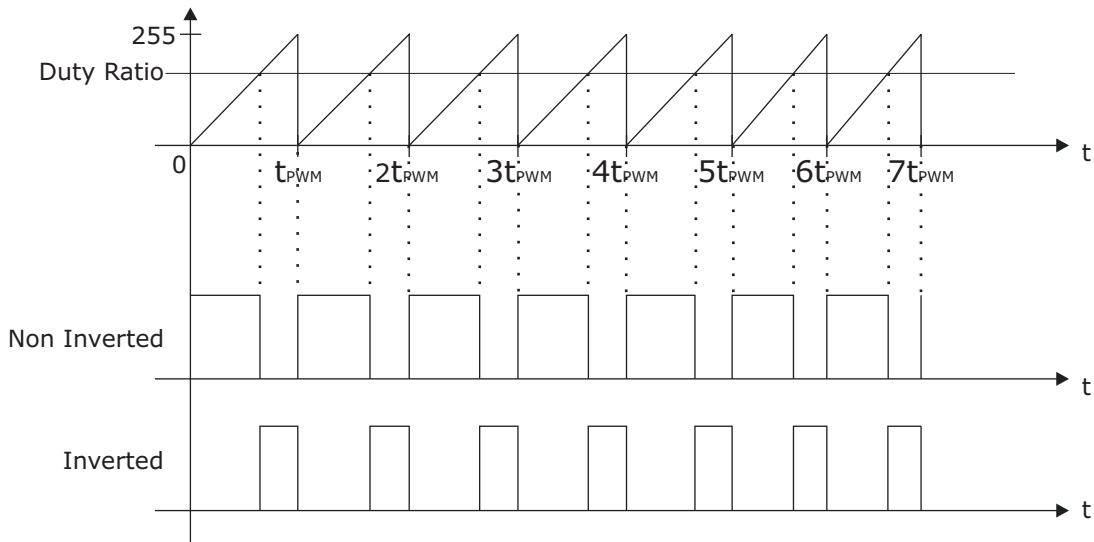
PHASE MODE

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



FAST MODE

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



Pwm_Change_Duty

Prototype	procedure Pwm_Change_Duty(duty_ratio : byte);
Description	Changes PWM duty ratio. Parameter duty_ratio takes values from 0 to 255, where 0 is 0%, 127 is 50%, and 255 is 100% duty ratio. Other specific values for duty ratio can be calculated as $(\text{Percent} * 255) / 100$.
Requires	You need a CMO module on PORTB to use this library. Check mikroPascal installation folder, subfolder “Examples”, for alternate solutions.
Example	<code>Pwm_Change_Duty(192); // Set duty ratio to 75%</code>

Pwm_Start

Prototype	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 mikroPascal 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 mikroPascal installation folder, subfolder “Examples”, for alternate solutions.
Example	<code>Pwm_Stop();</code>

Library Example

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

```

program pwm_test;

var text : string[ 7];
    duty : byte;

begin

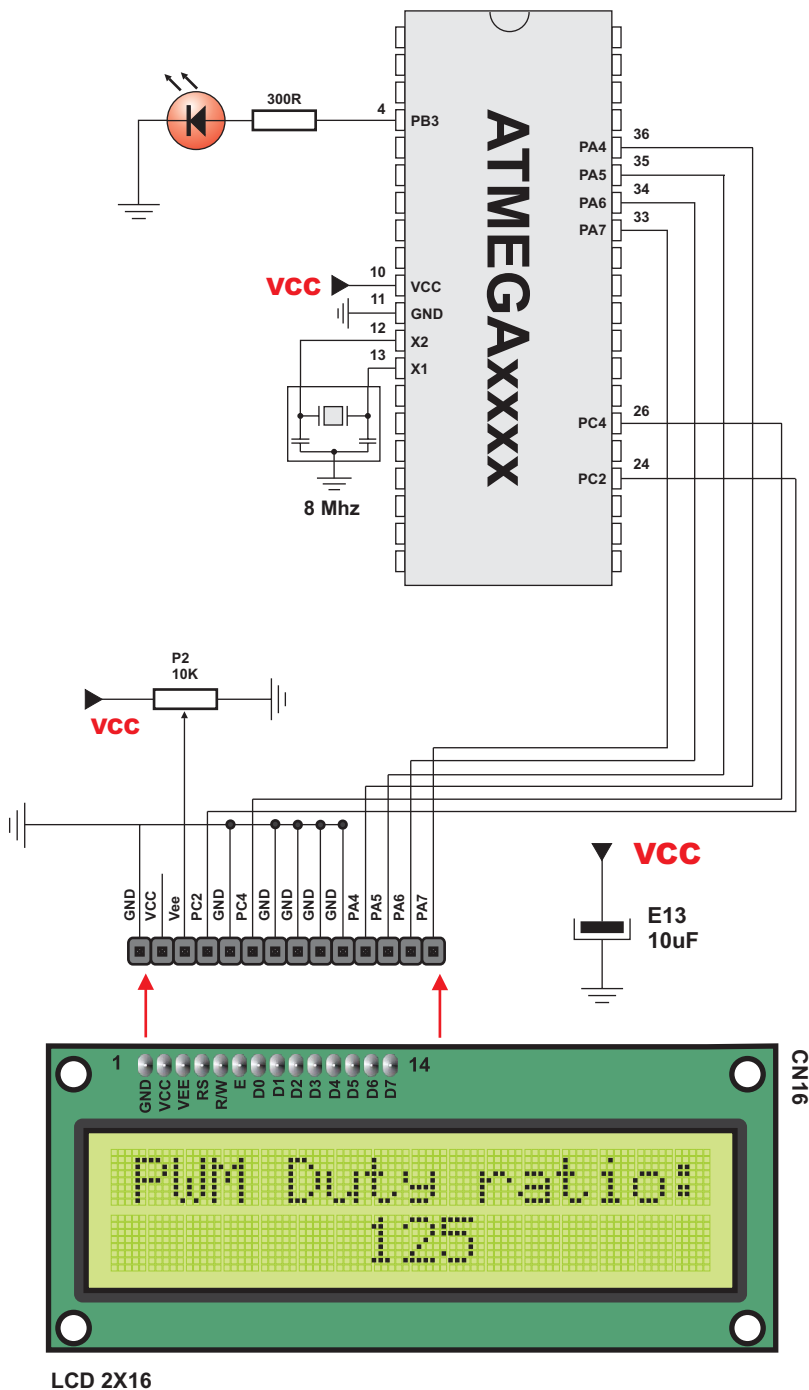
    //look on portb.3 to see how it is changing
    DDRB.3:=1; // set direction register for pwm
    Lcd_Init(PORTC, 4, 2, PORTA, LCD_HI_NIBBLE);

    Lcd_Cmd(LCD_CURSOR_OFF);
    Lcd_Out(1,1,'Phase correct');
    Lcd_Out(2,1,'mode duty :');

    duty := 20;
    Pwm_Init(PWM_PHASE_CORRECT_MODE, PWM_PRESCALER_1024, PWM_NON_INVERTED, duty);
    while true do
        begin
            ByteToStr(duty, text);
            Lcd_Out(2,14, text);
            Delay_ms(100);
            duty := duty + 1;
            Pwm_Set_Duty(duty);
        end;
    end.

```

Hardware Connection



Software I2C Library

mikroPascal 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 I2C.

Library Routines

```
Soft_I2c_Config
Soft_I2c_Start
Soft_I2c_Read
Soft_I2c_Write
Soft_I2c_Stop
```

Soft_I2c_Config

Prototype	procedure Soft_I2c_Config(var port : byte; SDA, SCL : byte);
Description	<p>Configures software I2C. Parameter <code>port</code> 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.</p> <p>Soft_I2c_Config needs to be called before using other functions from Soft I2c Library.</p>
Example	Soft_I2c_Config(PORTB, 1, 2);

Soft_I2c_Start

Prototype	procedure Soft_I2c_Start;
Description	Issues START signal. Needs to be called prior to sending and receiving data.
Requires	Soft I2C must be configured before using this function. See Soft_I2c_Config.
Example	Soft_I2c_Start;

Soft_I2c_Read

Prototype	function Soft_I2C_Read(ack : byte) : 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);

Soft_I2c_Write

Prototype	function Soft_I2c_Write(data : byte) : byte;
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter data) via I2C bus.
Requires	START signal needs to be issued in order to use this function. See Soft_I2c_Start.
Example	Soft_I2c_Write(\$A3);

Soft_I2c_Stop

Prototype	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;

var   ee_adr, ee_data : byte;
       jj : word;

begin
    Soft_I2c_Config(PORTD, 3, 4);           // Initialize full master mode
    TRISC := 0;                             // PORTC is output
    PORTC := $FF;                           // Initialize PORTC
    Soft_I2c_Start();                       // Issue I2c start signal
    Soft_I2c_Write($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(data that will be written)
    Soft_I2c_Stop();                       // Issue I2c stop signal

    for jj := 0 to 65500 do                 // Pause while EEPROM writes data
        nop;

    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_stop signal
    PORTC := ee_data;                      // Display data on PORTC

while TRUE do nop;
end.

```


Software SPI Library

mikroPascal 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	procedure Soft_Spi_Config(var port : byte; SDI, SDO, SCK : byte);
Description	<p>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.</p> <p>Soft_Spi_Config needs to be called before using other functions from Soft SPI Library.</p>
Example	<p>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:</p> <pre>Soft_Spi_Config(PORTB, 1, 2, 3);</pre>

Soft_Spi_Read

Prototype	function Soft_Spi_Read(buffer : byte) : 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	procedure Soft_Spi_Write(data : 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.

```
program Soft_SPI_test;

var value : word;

procedure Dac_Output(out_value: word);
var loc : byte;
begin
    PORTD := PORTD and $DF; // clear CS
    loc := out_value shr 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 := PORTD or $20; // set CS
end;

begin
    DDRD := DDRD or $20; // 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 do
    begin
        value := 1;
        while value < $FFF do
        begin
            Dac_Output(value); // changing the voltage from zero to AREF.
            Delay_ms(5);
            inc(value);
        end;
    end;
end.
```

Software UART Library

mikroPascal 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_Write_Text

Soft_Uart_Init

Prototype	<code>procedure Soft_Uart_Init(var port : byte; const rx, tx, baud_rate, inverted : 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	function Soft_Uart_Read(var error : byte) : 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; repeat data := Soft_Uart_Read(error); until error = 0;</pre>

Soft_Uart_Write

Prototype	procedure Soft_Uart_Write(data : byte);
Description	Function transmits a byte (data) 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>

Soft_Uart_Write_Text

Prototype	<code>procedure Soft_Uart_Write_Text(var uart_text : string[20]);</code>
Description	Sends text (parameter <code>uart_text</code>) via soft UART. Text should be zero terminated.
Requires	Soft UART must be initialized and communication established before using this function. See <code>Soft_Uart_Init</code> . Be aware that during transmission, software UART is incapable of receiving data – data transfer protocol must be set in such a way to prevent loss of information.
Example	<code>Soft_Uart_Write_Text(text);</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 mikroPascal terminal for RS232 communication, menu choice **Tools > Terminal**. Hardware connection is given in USART library section.

```

program soft_uart_test;

var received_byte, er : byte;

begin
    Soft_Uart_Init(PORTB, 1, 2, 2400, 0);           // Initialize soft UART
    er := 1;
    while true do
        begin
            repeat
                received_byte := Soft_Uart_Read(er);    // Read received data
            until er = 0;
            Soft_Uart_Write(received_byte);             // Send data via UART
        end;
    end.

```

Sound Library

mikroPascal 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	procedure Sound_Init(var port: byte; pin: 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	procedure Sound_Play(freq_in_hz, period_ms: 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	procedure Sound_Play_Khz(freq_in_khz, period_ms: 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 milliseconds (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;

procedure Tone1;
begin
    Sound_Play(6000, 400);    // frequency is 6kHz and duration is 400ms
end;

procedure Tone2;
begin
    Sound_Play(7000, 400);    // frequency is 7kHz and duration is 400ms
end;

procedure Tone3;
begin
    Sound_Play(8000, 400);    // frequency is 8kHz and duration is 400ms
end;

// continues on next page...

```


//continues here...

```
procedure Melody;           // Plays the melody "Yellow house"
begin
    Tone1; Tone2; Tone3; Tone3;
    Tone1; Tone2; Tone3; Tone3;
    Tone1; Tone2; Tone3; Tone1;
    Tone2; Tone3; Tone3; Tone1;
    Tone2; Tone3; Tone3; Tone3;
    Tone2; Tone1;
end;

begin
    Sound_Init(PORTB,3);      // put piezo on portb.3
    Sound_Play_Khz(3, 200);

    while true do
        begin
            if Button(PORTB,7,1,1) then           // RB7 plays Tone1
                Tone1;
            while PINB.7 = 1 do nop;    // Wait for button to be released

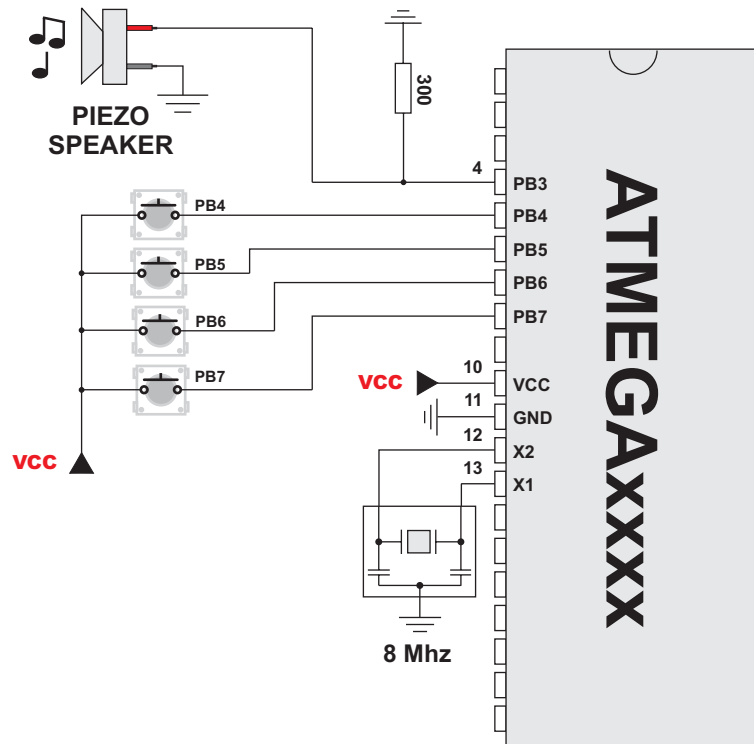
            if Button(PORTB,6,1,1) then           // RB6 plays Tone2
                Tone2;
            while PINB.6 = 1 do nop;    // Wait for button to be released

            if Button(PORTB,5,1,1) then           // RB5 plays Tone3
                Tone3;
            while PINB.5 = 1 do nop;    // Wait for button to be released

            if Button(PORTB,4,1,1) then           // RB4 plays Melody
                Melody;

            while PINB.4 = 1 do nop;    // Wait for button to be released
        end;
    end.
```

Hardware Connection



SPI Library

SPI unit is available with a number of AVR MCU models. mikroPascal 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: Examples for AVRmicros with unit on other ports can be found in your mikroPascal installation folder, subfolder “Examples”.

```
Spi_Init
Spi_Init_Advanced
Spi_Read
Spi_Write
```

Library Routines

Spi_Init

Prototype	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	<p>You need AVR MCU with hardware integrated SPI.</p> <p>DDR must be set before calling this routine like this:</p> <pre>DDRB:=DDRB and \$A0; // seting to input direction registers DDRB:=DDRB or \$BF; // seting to output direction registers</pre>
Example	Spi_Init;

Spi_Init_Advanced

Prototype	procedure Spi_Init_Advanced(master, data_sample, clock_idle, transmit_edge : 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	function Spi_Read(buffer : byte) : 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	procedure Spi_Write(data : byte) : byte;
Description	Writes byte <code>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;

var value : word;

procedure Dac_Output(out_value: word);
var loc : byte;
begin
    PORTD.5:=0;           // clear CS
    Spil_Write(hi(out_value) or $30);
    Delay_us(50);
    Spil_Write(lo(loc));
    Delay_us(50);
    PORTD.5 := 1;        // set CS
end;

begin
    DDRD.5 := 1;         // set direction of CS line to be output.

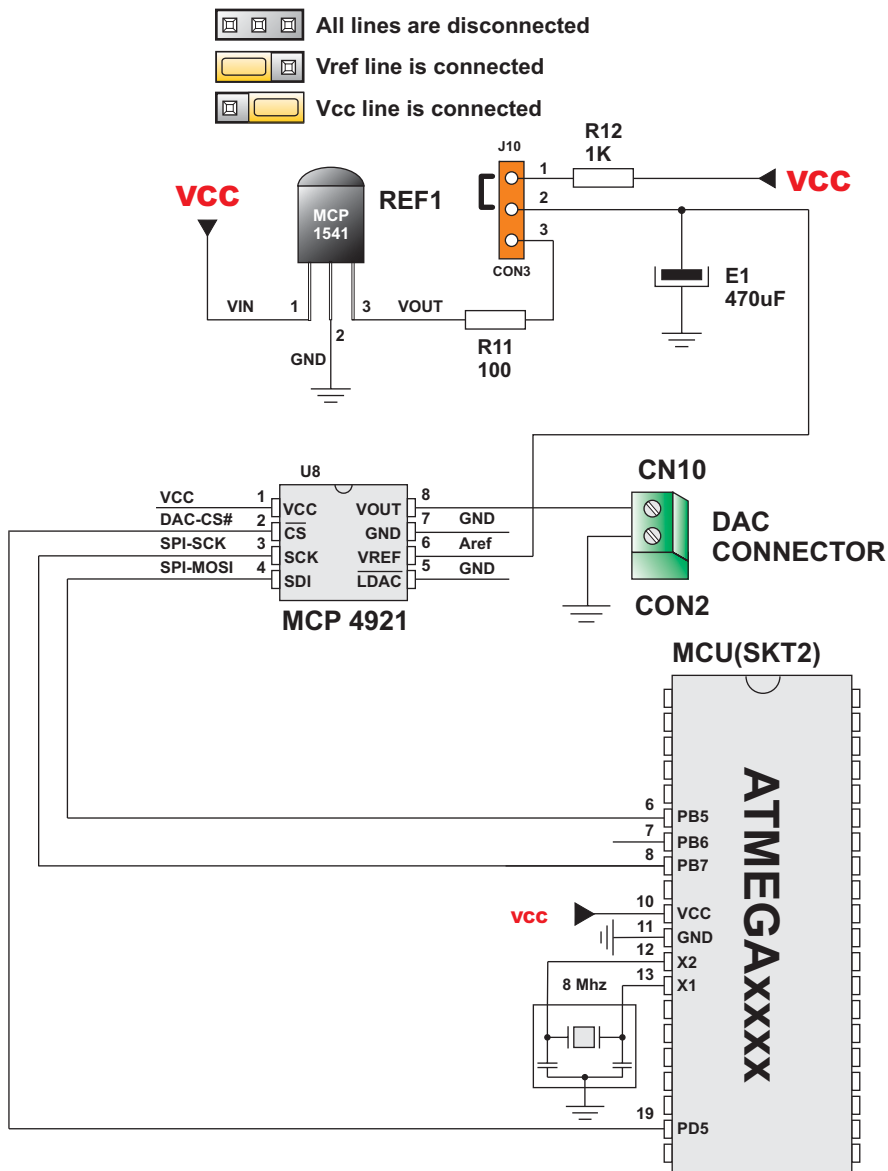
    DDRB:=DDRB and $A0; // seting to input direction registers
    DDRB:=DDRB or $BF;  // seting to output direction registers

    Spil_Init;

    while TRUE do
    begin
        value := 1;
        while value < $FFF do
        begin
            Dac_Output(value);
            Delay_ms(10);
            value:=value+1;
        end;
    end;

end.
```

HW Connection



USART Library

USART hardware unit is available with a number of AVR micros. mikroPascal 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. Then, simply use the functions listed below.

Note: Examples for AVRmicros with unit on other ports can be found in “Examples” in mikroPascal installation folder.

Library Routines

Usart1_Init	Usart2_Init
Usart1_Data_Ready	Usart2_Data_Ready
Usart1_Read	Usart2_Read
Usart1_Read_Text	Usart2_Read_Text
Usart1_Write_Char	Usart2_Write_Char
Usart1_Write_Text	Usart2_Write_Text

Note: Certain AVR micros with two USART units, such as ATmega128, require you to specify the unit you want to use. Simply append the number 1 or 2 to a function name. For example, Usart_Write2.

Usart1_Init

Prototype	procedure Usart1_Init(const baud_rate : longint);
Description	<p>Initializes hardware USART unit 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.</p> <p>Usart_Init needs to be called before using other functions from USART Library.</p>
Requires	You need AVR MCU with hardware USART.
Example	Usart1_Init(2400); // Establish communication at 2400 bps

Usart1_Data_Ready

Prototype	function Usart1_Data_Ready : 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 unit must be initialized and communication established before using this function. See Usart1_Init.
Example	<pre>// If data is ready, read it: if Usart1_Data_Ready() = 1 then receive := Usart1_Read;</pre>

Usart1_Read

Prototype	function Usart1_Read : byte;
Returns	Returns the received byte. If byte is not received, returns 0.
Description	Function receives a byte via USART. Use the function Usart1_Data_Ready to test if data is ready first.
Requires	USART HW unit must be initialized and communication established before using this function. See Usart_Init.
Example	<pre>// If data is ready, read it: if Usart1_Data_Ready() = 1 then receive := Usart1_Read;</pre>

Usart1_Read_Text

Prototype	procedure Usart1_Read_Text(var output, delimiter : string [20]);
Description	<p>Reads characters received via USART until the delimiter sequence is detected. The read sequence is stored in the parameter output; delimiter sequence is stored in the parameter delimiter.</p> <p>This is a blocking call: the delimiter sequence is expected, otherwise the procedure exits after 20 recieved characters.</p>
Requires	USART HW unit must be initialized and communication established before using this function. See Usart1_Init.
Example	<pre> Usart1_Init(9600); delim := 'OK'; while TRUE do begin if Usart1_Data_Ready() = 1 then begin Usart1_Read_Text(txt, delim); Usart1_Write_Text(txt); end; end; </pre>

Usart1_Write_Char

Prototype	procedure Usart1_Write_Char(data : byte);
Description	Function transmits a byte (data) via USART.
Requires	USART HW unit must be initialized and communication established before using this function. See Usart1_Init.
Example	Usart1_Write_Char(\$1E); // send chunk via USART

Usart1_Write_Text

Prototype	procedure Usart1_Write_Text(var uart_text : string [20]);
Description	Sends text (parameter uart_text) via USART. Text should be zero terminated.
Requires	USART HW unit must be initialized and communication established before using this function. See Usart1_Init.
Example	<pre> Usart1_Init(9600); delim := 'OK'; while TRUE do begin if Usart1_Data_Ready() = 1 then begin Usart1_Read_Text(txt, delim); Usart1_Write_Text(txt); end; end; </pre>

Usart2_Init

Prototype	procedure Usart2_Init(const baud_rate : longint);
Description	<p>Initializes hardware USART unit 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.</p> <p>Usart2_Init needs to be called before using other functions from USART Library.</p>
Requires	You need AVR MCU with hardware USART.
Example	<pre> Usart2_Init(2400); // Establish communication at 2400 bps </pre>

Usart2_Data_Ready

Prototype	function Usart2_Data_Ready : 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 unit must be initialized and communication established before using this function. See Usart1_Init.
Example	<pre>// If data is ready, read it: if Usart2_Data_Ready() = 1 then receive := Usart2_Read;</pre>

Usart2_Read

Prototype	function Usart2_Read : byte;
Returns	Returns the received byte. If byte is not received, returns 0.
Description	Function receives a byte via USART. Use the function Usart1_Data_Ready to test if data is ready first.
Requires	USART HW unit must be initialized and communication established before using this function. See Usart_Init.
Example	<pre>// If data is ready, read it: if Usart2_Data_Ready() = 1 then receive := Usart2_Read;</pre>

Usart2_Read_Text

Prototype	procedure Usart2_Read_Text(var output, delimiter : string [20]);
Description	<p>Reads characters received via USART until the delimiter sequence is detected. The read sequence is stored in the parameter output; delimiter sequence is stored in the parameter delimiter.</p> <p>This is a blocking call: the delimiter sequence is expected, otherwise the procedure exits after 20 recieved characters.</p>
Requires	USART HW unit must be initialized and communication established before using this function. See Usart2_Init.
Example	<pre> Usart2_Init(9600); delim := 'OK'; while TRUE do begin if Usart2_Data_Ready() = 1 then begin Usart2_Read_Text(txt, delim); Usart2_Write_Text(txt); end; end; </pre>

Usart2_Write_Char

Prototype	procedure Usart2_Write_Char(data : byte);
Description	Function transmits a byte (data) via USART.
Requires	USART HW unit must be initialized and communication established before using this function. See Usart2_Init.
Example	Usart2_Write_Char(\$1E); // send chunk via USART

Usart2_Write_Text

Prototype	<code>procedure Usart2_Write_Text(var uart_text : string[20]);</code>
Description	Sends text (parameter <code>uart_text</code>) via USART. Text should be zero terminated.
Requires	USART HW unit must be initialized and communication established before using this function. See <code>Usart1_Init</code> .
Example	<pre> Usart2_Init(9600); delim := 'OK'; while TRUE do begin if Usart2_Data_Ready() = 1 then begin Usart2_Read_Text(txt, delim); Usart2_Write_Text(txt); end; end; </pre>

Library Example

The example demonstrates simple data exchange via USART. When AVR receives the data, it immediately sends it back. If AVR is connected to the PC (see the figure below), you can test the example from mikroPascal terminal for RS232 communication, menu choice **Tools > Terminal**.

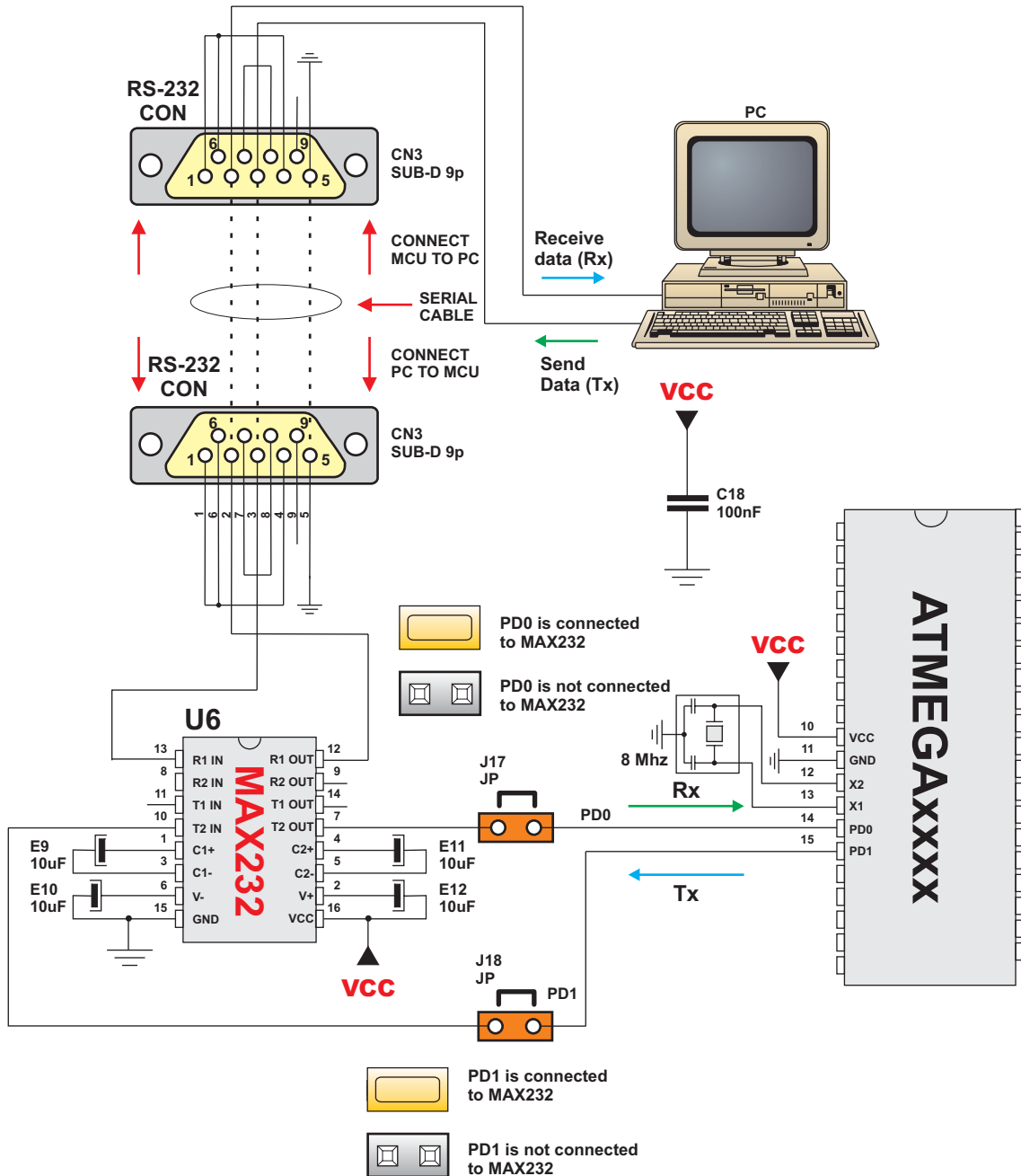
```

program rs232_com_test;
var  received_byte : byte;

begin
  Usart1_Init(2400);           // Initialize USART unit
  while true do
  begin
    if Usart1_Data_Ready = 1 then  // If data is received
    begin
      received_byte := Usart1_Read;  // Read received data
      Usart1_Write(received_byte);  // Send data via USART
    end;
  end;
end.

```

Hardware Connection



Util Library

Util library contains miscellaneous routines useful for project development.

Button

Prototype	function Button(var port : byte; pin, time, active_state : byte) : 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 RB0, to which the button is connected; on transition from 1 to 0 (release of button), PORTD is inverted:</p> <pre> while true do begin if Button(PORTB, 0, 1, 1) then oldstate := 255; if oldstate and Button(PORTB, 0, 1, 0) then begin PORTD := not(PORTD); oldstate := 0; end; end; end; end; </pre>

Conversions Library

mikroPascal 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
WordToHex

The following functions convert decimal values to BCD (Binary Coded Decimal) and vice versa:

Bcd2Dec
Dec2Bcd
Bcd2Dec16
Dec2Bcd16

ByteToStr

Prototype	<code>procedure ByteToStr(number : byte; var output : string[3]);</code>
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	<code>var t : word; txt : string[3] ; //... t := 24; ByteToStr(t, txt); // txt is ' 24' (one blank here)</code>

ShortToStr

Prototype	procedure ShortToStr(number : short; var output : 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> var t : short; txt : string[4] ; //... t := -24; ShortToStr(t, txt); // txt is ' -24' (one blank here) </pre>

WordToStr

Prototype	procedure WordToStr(number : word; var output : 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> var t : short; txt : string[4] ; //... t := -24; ShortToStr(t, txt); // txt is ' -24' (one blank here) </pre>

IntToStr

Prototype	procedure IntToStr(number : integer; var output : 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> var j : integer; txt : string[6] ; //... j := -4220; IntToStr(j, txt); // txt is ' -4220' (one blank here) </pre>

LongintToStr

Prototype	procedure LongintToStr(number: longint; var output : 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> var jj : longint; txt : string[11]; //... jj := -3700000; LongintToStr(jj, txt); // txt is ' -3700000' (three blanks here) </pre>

WordToHex

Prototype	procedure WordToHex(input : word ; var output : string [4]);
Description	Procedure creates string of hexadecimal digits out of input (numerical value of word type). Parameter output accepts the created string. Output string has fixed width of 4 characters, right aligned; remaining positions on the left (if any) are filled with zeroes.
Example	<pre> var input : word; output : string[4]; //... begin input := 26; WordToHex(input, output); Lcd_Out_Cp(output); // Print '001A' on LCD </pre>

Bcd2Dec

Prototype	function Bcd2Dec(bcdnum : byte) : byte;
Returns	Returns converted decimal value.
Description	Converts 8-bit BCD numeral bcdnum to its decimal equivalent.
Example	<pre> var a, b : byte; a := \$52; b := Bcd2Dec(a); // b equals 52 </pre>

Dec2Bcd16

Prototype	function Dec2Bcd16(decnum : byte) : byte;
Returns	Returns converted BCD value.
Description	Converts 16-bit decimal value decnum to BCD.
Example	<pre> var a, b : word; a := 4660; b := Dec2Bcd16(a); // b equals 1234 </pre>

Dec2Bcd

Prototype	function Dec2Bcd(decnum : byte) : byte;
Returns	Returns converted BCD value.
Description	Converts 8-bit decimal value decnum to BCD.
Example	<pre> var a, b : byte; a := 52; b := Dec2Bcd(a); // b equals \$52 </pre>

Bcd2Dec16

Prototype	function Bcd2Dec16(bcdnum : byte) : byte;
Returns	Returns converted decimal value.
Description	Converts 16-bit BCD numeral bcdnum to its decimal equivalent.
Example	<pre> var a, b : word; a := 1234; b := Bcd2Dec16(a); // b equals 4660 </pre>

Math Library

Math Library implements a number of common mathematical functions.

Library Routines

Acos
Asin
Atan
Atan2
Ceil
Cos
CosE3
Cosh
Exp
Fabs
Floor
Frexp
Fmod
Ldexp
Log
Log10
Modf
Pow
Sin
SinE3
Sinh
Sqrt
Tan
Tanh

Acos

Prototype	function Acos(<i>x</i> : real) : real;
Description	Function returns the arc cosine of parameter <i>x</i> ; that is, the value whose cosine is <i>x</i> . Input parameter <i>x</i> must be between -1 and 1 (inclusive). The return value is in radians, between 0 and pi (inclusive).

Asin

Prototype	function Asin(<i>x</i> : real) : real;
Description	Function returns the arc sine of parameter <i>x</i> ; that is, the value whose sine is <i>x</i> . Input parameter <i>x</i> must be between -1 and 1 (inclusive). The return value is in radians, between -pi/2 and pi/2 (inclusive).

Atan

Prototype	function Atan(<i>x</i> : real) : real;
Description	Function computes the arc tangent of parameter <i>x</i> ; that is, the value whose tangent is <i>x</i> . The return value is in radians, between -pi/2 and pi/2 (inclusive).

Atan2

Prototype	function Atan2(<i>x</i> , <i>y</i> : real) : real;
Description	This is the two argument arc tangent function. It is similar to computing the arc tangent of <i>y/x</i> , except that the signs of both arguments are used to determine the quadrant of the result, and <i>x</i> is permitted to be zero. The return value is in radians, between -pi and pi (inclusive).

Ceil

Prototype	function Ceil(<i>x</i> : real) : real;
Description	Function returns value of parameter <i>x</i> rounded up to the next whole number.

Cos

Prototype	function Cos(<i>x</i> : real) : real;
Description	Function returns the cosine of <i>x</i> in radians. The return value is from -1 to 1.

CosE3

Prototype	function CosE3(<i>x</i> : word) : integer;
Description	<p>Function takes parameter <i>x</i> which represents angle in degrees, and returns its cosine multiplied by 1000 and rounded up to the nearest integer:</p> <pre>result := round_up(cos(<i>x</i>)*1000)</pre> <p>The function is implemented as a lookup table; maximum error obtained is ± 1.</p>

Cosh

Prototype	function Cosh(<i>x</i> : real) : real;
Description	Function returns the hyperbolic cosine of <i>x</i> , defined mathematically as $(e^x + e^{-x}) / 2$. If the value of <i>x</i> is too large (if overflow occurs), the function fails.

Exp

Prototype	function Exp(<i>x</i> : real) : real;
Description	Function returns the value of <i>e</i> — the base of natural logarithms — raised to the power of <i>x</i> (i.e. e^x).

Fabs

Prototype	function Fabs(<i>x</i> : real) : real;
Description	Function returns the absolute (i.e. positive) value of <i>x</i> .

Floor

Prototype	function Floor(<i>x</i> : real) : real;
Description	Function returns value of parameter <i>x</i> rounded down to the nearest integer.

Fmod

Prototype	function Fmod(<i>x</i> , <i>y</i> : real) : real;
Description	Function computes the floating point remainder of <i>x</i> / <i>y</i> . Function returns the value $x - i * y$ for some integer <i>i</i> such that, if <i>y</i> is nonzero, the result has the same sign as <i>x</i> and magnitude less then the magnitude of <i>y</i> . If <i>y</i> is zero, the <code>fmod</code> function returns zero.

Frexp

Prototype	function Frexp(num : real; n : ^integer) : real;
Description	Function splits a floating-point value <code>num</code> into a normalized fraction and an integral power of 2. Return value is the normalized fraction, and the integer exponent is stored in the object pointed to by <code>n</code> .

Ldexp

Prototype	function Ldexp(num : real; n : integer) : real;
Description	Function returns the result of multiplying the floating-point number <code>num</code> by 2 raised to the power <code>n</code> (i.e. returns $x * 2^n$).

Log

Prototype	function Log(x : real) : real;
Description	Function returns the natural logarithm of <code>x</code> (i.e. $\log_e(x)$).

Log10

Prototype	function Log10(x : real) : real;
Description	Function returns the base-10 logarithm of <code>x</code> (i.e. $\log_{10}(x)$).

Modf

Prototype	function Modf(num : real; whole : ^real) : real;
Description	Function returns the signed fractional component of num, placing its whole number component into the variable pointed to by whole.

Pow

Prototype	function Pow(x, y: real) : real;
Description	Function returns the value of x raised to the power of y (i.e. x^y). If the x is negative, function will automatically cast the y into longint.

Sin

Prototype	function Sin(x : real) : real;
Description	Function returns the sine of x in radians. The return value is from -1 to 1.

SinE3

Prototype	function SinE3(x : word) : integer;
Description	<p>Function takes parameter x which represents angle in degrees, and returns its sine multiplied by 1000 and rounded up to the nearest integer:</p> <pre>result := round_up(sin(x)*1000)</pre> <p>The function is implemented as a lookup table; maximum error obtained is ± 1.</p>

Sinh

Prototype	function Sinh(x : real) : real;
Description	Function returns the hyperbolic sine of x, defined mathematically as $(e^x - e^{-x}) / 2$. If the value of x is too large (if overflow occurs), the function fails.

Sqrt

Prototype	function Sqrt(x : real) : real;
Description	Function returns the non negative square root of num.

Tan

Prototype	function Tan(x : real) : real;
Description	Function returns the tangent of x in radians. The return value spans the allowed range of floating point in mikroPascal.

Tanh

Prototype	function Tanh(x : real) : real;
Description	Function returns the hyperbolic tangent of x, defined mathematically as $\sinh(x) / \cosh(x)$.

Delay Library

mikroPascal 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	procedure Delay_us(const time_in_us : 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	procedure Delay_ms(const time_in_ms : 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	procedure Delay_Cyc(number_in_cycles : 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

Memchr
Memcmp
Memcpy
Memmove
Memset
Strcat
Strchr
Strcmp
Strcpy
Strcspn
Strlen
Strncat
Strncmp
Strncpy
Strpbrk
Strrchr
Strspn
Strstr
strAppendSuf
strAppendPre

Memchr

Prototype	function Memchr(p : word; ch : char; n : word) : word;
Description	<p>Function locates the first occurrence of byte <code>ch</code> in the initial <code>n</code> bytes of memory area starting at the address <code>p</code>. Function returns the offset of this occurrence from the memory address <code>p</code> or <code>\$FFFF</code> if the character was not found.</p> <p>For parameter <code>p</code> you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example <code>@mystring</code> or <code>@PORTB</code>.</p>

Memcmp

Prototype	function Memcmp(p1, p2, n : word) : integer;								
Description	<p>Function returns a positive, negative, or zero value indicating the relationship of first <code>n</code> bytes of memory areas starting at addresses <code>p1</code> and <code>p2</code>.</p> <p>The <code>Memcmp</code> function compares two memory areas starting at addresses <code>p1</code> and <code>p2</code> for <code>n</code> bytes and returns a value indicating their relationship as follows:</p> <table> <tr> <td>Value</td><td>Meaning</td></tr> <tr> <td>< 0</td><td>p1 "less than" p2</td></tr> <tr> <td>= 0</td><td>p1 "equal to" p2</td></tr> <tr> <td>> 0</td><td>p1 "greater than" p2</td></tr> </table> <p>The value returned by function is determined by the difference between the values of the first pair of bytes that differ in the strings being compared.</p> <p>For parameters <code>p1</code> and <code>p2</code> you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example <code>@mystring</code> or <code>@PORTB</code>.</p>	Value	Meaning	< 0	p1 "less than" p2	= 0	p1 "equal to" p2	> 0	p1 "greater than" p2
Value	Meaning								
< 0	p1 "less than" p2								
= 0	p1 "equal to" p2								
> 0	p1 "greater than" p2								

Memcpy

Prototype	procedure Memcpy(p1, p2, n : word);
Description	<p>Function copies n bytes from the memory area starting at the address p2 to the memory area starting at p1. If these memory buffers overlap, the memcpy function cannot guarantee that bytes are copied before being overwritten. If these buffers do overlap, use the Memmove function.</p> <p>For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.</p>

Memmove

Prototype	procedure Memmove(p1, p2, n : word);
Description	<p>Function copies n bytes from the memory area starting at the address p2 to the memory area starting at p1. If these memory buffers overlap, the Memmove function ensures that bytes in p2 are copied to p1 before being overwritten.</p> <p>For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.</p>

Memset

Prototype	procedure Memset(p : word; ch : char; n : word);
Description	<p>Function fills the first n bytes in the memory area starting at the address p with the value of byte ch.</p> <p>For parameter p you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example @mystring or @PORTB.</p>

Strcat

Prototype	procedure Strcat(var s1, s2 : string [100]);
Description	Function appends the value of string s2 to string s1 and terminates s1 with a null character.

Strchr

Prototype	function Strchr(var s : string [100]; ch : char) : byte ;
Description	<p>Function searches the string s for the first occurrence of the character ch. The null character terminating s is not included in the search.</p> <p>Function returns the position (index) of the first character ch found in s; if no matching character was found, function returns \$FF.</p>

Strcmp

Prototype	function Strcmp(var s1, s2 : string [100]) : integer ;
Description	<p>Function lexicographically compares the contents of strings s1 and s2 and returns a value indicating their relationship:</p> <p>Value Meaning</p> <p>< 0 s1 "less than" s2</p> <p>= 0 s1 "equal to" s2</p> <p>> 0 s1 "greater than" s2</p> <p>The value returned by function is determined by the difference between the values of the first pair of bytes that differ in the strings being compared.</p>

Strcpy

Prototype	procedure Strcpy(var s1, s2 : string [100]);
Description	Function copies the value of string s2 to the string s1 and appends a null character to the end of s1.

Strcspn

Prototype	function Strcspn(var s1, s2 : string [100]) : byte;
Description	The strcspn function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of characters not from the string pointed to by s2. Function returns the length of the segment.

Strlen

Prototype	function Strlen(var s : string [100]) : byte;
Description	Function returns the length, in bytes, of the string s. The length does not include the null terminating character.

Strncat

Prototype	procedure Strncat(var s1, s2 : string [100] ; n : byte);
Description	Function appends at most n characters from the string s2 to the string s1 and terminates s1 with a null character. If s2 is shorter than n characters, s2 is copied up to and including the null terminating character.

Strncmp

Prototype	function Strncmp(var s1, s2 : string [100] ; n : byte) : integer;								
Description	<p>Function lexicographically compares the first n bytes of the strings s1 and s2 and returns a value indicating their relationship:</p> <table> <tr> <td>Value</td><td>Meaning</td></tr> <tr> <td>< 0</td><td>s1 "less than" s2</td></tr> <tr> <td>= 0</td><td>s1 "equal to" s2</td></tr> <tr> <td>> 0</td><td>s1 "greater than" s2</td></tr> </table> <p>The value returned by function is determined by the difference between the values of the first pair of bytes that differ in the strings being compared (within first n bytes).</p>	Value	Meaning	< 0	s1 "less than" s2	= 0	s1 "equal to" s2	> 0	s1 "greater than" s2
Value	Meaning								
< 0	s1 "less than" s2								
= 0	s1 "equal to" s2								
> 0	s1 "greater than" s2								

Strncpy

Prototype	procedure Strncpy(var s1, s2 : string [100] ; n : byte);
Description	Function copies at most n characters from the string s2 to the string s1. If s2 contains fewer characters than n, s1 is padded out with null characters up to the total length of n characters.

Strpbrk

Prototype	<code>procedure Strpbrk(var s1, s2 : string[100]);</code>
Description	Function searches s1 for the first occurrence of any character from the string s2. The null terminator is not included in the search. Function returns an index of the matching character in s1. If s1 contains no characters from s2, function returns \$FF.

Strrchr

Prototype	<code>procedure Strrchr(var s : string[100]; ch : byte);</code>
Description	Function searches the string s for the last occurrence of character ch. The null character terminating s is not included in the search. Function returns an index of the last ch found in s; if no matching character was found, function returns \$FF.

Strspn

Prototype	<code>function Strspn(var s1, s2 : string[100]) : byte;</code>
Description	The strspn function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of the characters from the string pointed to by s2. Function returns the length of the segment.

Strstr

Prototype	<code>function Strstr(var s1, s2 : string[100]) : byte;</code>
Description	Function locates the first occurrence of the string s2 in the string s1 (excluding the terminating null character). Function returns a number indicating the position of the first occurrence of s2 in s1; if no string was found, function returns \$FF. If s2 is a null string, the function returns 0.

strAppendSuf

Prototype	<code>procedure strAppendSuf(var s1: string[100]; letter: char);</code>
Description	Adds suffix(letter) to string (s1).
Example	<pre>txt:= '123'; strAppendSuf(txt,'4'); // txt = '1234'</pre>

strAppendPre

Prototype	<code>procedure strAppendPre(letter: char; var s1: string[100]);</code>
Description	Adds prefix(letter) to string (s1).
Example	<pre>txt:= '123'; strAppendPre('0',txt); // txt = '0123'</pre>

LCD Custom Library (4-bit interface)

mikroPascal for AVR provides a library for communicating with commonly used LCD (4-bit interface) with custom defined pinout. 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

```
Lcd4_Custom_Init
Lcd4_Custom_Out
Lcd4_Custom_Out_CP
Lcd4_Custom_Chrc
Lcd4_Custom_Chrc_CP
Lcd4_Custom_Cmd
```

Lcd4_Custom_Init

Prototype	procedure Lcd4_Custom_Init(var data_lat: byte ; db3, db2, db1, db0: byte ; var ctrl_lat: byte ; rs, ctrl_rw, enable: byte);
Description	Initializes LCD data port and control port with pin settings you specify.
Example	Lcd4_Custom_Init(PORTA, 7, 6, 5, 4, PORTC, 2, 1, 4);

Lcd4_Custom_Out

Prototype	procedure Lcd4_Custom_Out(row, column: byte ; var text: string [20]);
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 Lcd4_Custom_Init.
Example	Lcd4_Custom_Out(2,1,"mikroElektronika");

Lcd4_Custom_Out_CP

Prototype	procedure Lcd4_Custom_Out_CP(var text: string [20]);
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. Lcd4_Custom_Init.
Example	Lcd4_Custom_Out_CP("Here!");

Lcd4_Custom_Chrc

Prototype	procedure Lcd4_Custom_Chrc(row, column, out_char: 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 Lcd4_Custom_Init.
Example	Lcd4_Custom_Chrc(2, 3, 'i');

Lcd4_Custom_Chrc_CP

Prototype	procedure Lcd4_Custom_Chrc_CP(out_char: 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 Lcd4_Custom_Init.
Example	Lcd4_Custom_Chrc_CP('e');

Lcd4_Custom_Cmd

Prototype	procedure Lcd4_Custom_Cmd(out_char: byte);
Description	Sends command to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown on the page 140.
Requires	Port with LCD must be initialized. See Lcd4_Custom_Init.
Example	Lcd4_Custom_Cmd(Lcd_Clear); <i>// Clear LCD display</i>

Library Example

You can use Lcd4_Custom_Init for custom pin settings. For example (second figure below):

```

program lcd4_custom_test;

begin

    Lcd4_Custom_Init(PORTA, 7, 6, 5, 4, PORTC, 2, 1, 4);

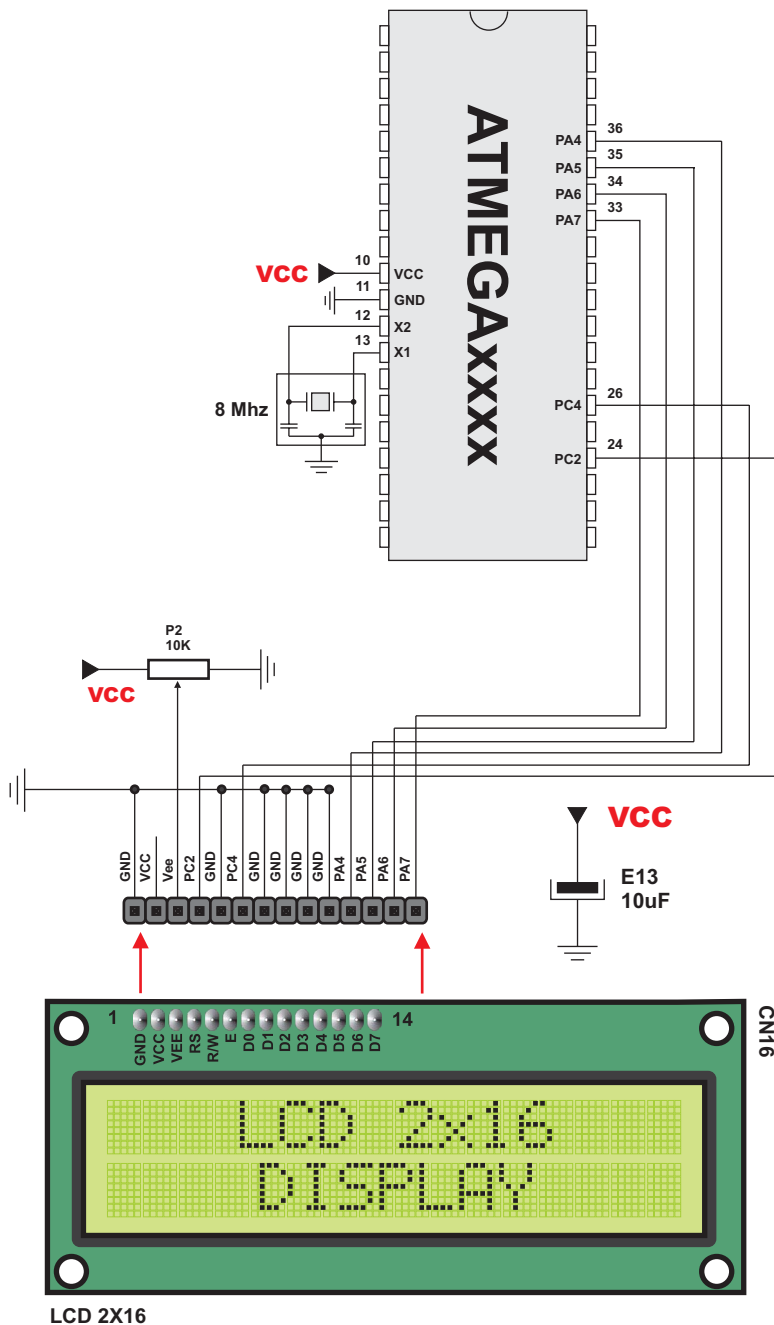
    Lcd4_Custom_Out(2,1,'mikroElektronika');

end.
```

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

Hardware Connection



Port Expander Library

The SPI Expander Library facilitates working with MCP23S17, Microchip's SPI port expander. The chip connects to the AVR according to the scheme presented below.

Note: AVR need to have a hardware SPI module.

Library Routines

```
Expander_Init
PortExpanderSelect
PortExpanderUnSelect
Expander_Read_Byte
Expander_Write_Byte
Expander_Set_Mode
Expander_Read_Array
Expander_Write_Array
Expander_Read_PortA
Expander_Read_PortB
Expander_Read_ArrayPortA
Expander_Read_ArrayPortB
Expander_Write_PortA
Expander_Write_PortB
Expander_Set_DirectionPortA
Expander_Set_DirectionPortB
Expander_Set_PullUpsPortA
Expander_Set_PullUpsPortB
```

Expander_Init

Prototype	procedure Expander_Init(var RstPort : byte ; RstPin : byte ; var CSPort : byte ; CSPin, ModuleAddress : byte);
Description	Establishes SPI communication with the expander and initializes the expander. RstPort and RstPin - Sets pin connected on reset pin of spi expander. CSPort and CSPin - Sets pin connected on CS pin of spi expander. moduleaddress - address of spi expander (hardware setting of A0, A1 and A2 pins (connected on VCC or GND) on spi expander). spi_module - Sets SPI1 or SPI2 module to work with SPI expander. For modules that have only one SPI module this setting has no effect.
Requires	This procedure needs to be called before using other routines of PORT Expander library.
Example	Expander_Init(PORTB, 0, PORTB, 1, 0);

PortExpanderSelect

Prototype	procedure PortExpanderSelect;
Description	Selects current port expander.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	PortExpanderSelect;

PortExpanderUnSelect

Prototype	procedure PortExpanderUnSelect;
Description	Un-Selects current port expader.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	PortExpanderUnSelect;

Expander_Read_Byte

Prototype	function Expander_Read_Byte (ModuleAddress, RegAddress : byte) : byte ;
Returns	Byte read from port expander.
Description	Function reads byte from port expander on ModuleAddress and port on RegAddress.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Read_Byte (0,1) ;

Expander_Write_Byte

Prototype	procedure Expander_Write_Byte (ModuleAddress, RegAddress, Data : byte) ;
Returns	Nothing.
Description	This routine writes data to port expander on ModuleAddress and port on RegAddress.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Write_Byte (0,1,\$FF) ;

Expander_Set_Mode

Prototype	procedure Expander_Set_Mode (ModuleAddress, Mode : byte) ;
Returns	Nothing.
Description	Sets port expander mode on ModuleAddress.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Set_Mode (1, 0) ;

Expander_Read_ArrayPortA

Prototype	procedure Expander_Read_ArrayPortA (ModuleAddress, NoBytes : byte ; var DestArray : array [100] of byte) ;
Returns	Nothing.
Description	This routine reads array of bytes (DestArray) from port expander on ModuleAddress and portA. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Read_PortA (0, 1, data) ;

Expander_Read_Array

Prototype	procedure Expander_Read_Array (ModuleAddress, StartAddress, NoBytes : byte ; var DestArray : array [100] of byte) ;
Returns	Nothing.
Description	This routine reads array of bytes (DestArray) from port expander on ModuleAddress and StartAddress. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Read_Array (1, 1, 5, data) ;

Expander_Write_Array

Prototype	procedure Expander_Write_Array(ModuleAddress, StartAddress, NoBytes : byte ; var SourceArray : array [100] of byte);
Returns	Nothing.
Description	This routine writes array of bytes (DestArray) to port expander on ModuleAddress and StartAddress. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Write_Array(1,1,5,data);

Expander_Read_PortA

Prototype	function Expander_Read_PortA(Address : byte) : byte ;
Returns	Read byte.
Description	This routine reads byte from port expander on Address and PortA.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Read_PortA(1);

Expander_Read_ArrayPortB

Prototype	procedure Expander_Read_ArrayPortB(ModuleAddress, NoBytes : byte ; var DestArray : array [100] of byte);
Returns	Nothing.
Description	This routine reads array of bytes (DestArray) from port expander on ModuleAddress and portB. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Read_PortB(0,8,data);

Expander_Write_PortA

Prototype	procedure Expander_Write_PortA(ModuleAddress, Data : byte);
Returns	Nothing.
Description	This routine writes byte to port expander on ModuleAddress and portA.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_write_PortA(3,\$FF);

Expander_Write_PortB

Prototype	procedure Expander_Write_PortB(ModuleAddress, Data : byte);
Returns	Nothing.
Description	This routine writes byte to port expander on ModuleAddress and portB.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_write_PortB(2,\$FF);

Expander_Set_DirectionPortA

Prototype	procedure Expander_Set_DirectionPortA(ModuleAddress, Data : byte) ;
Description	Set port expander PortA pin as input or output.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Set_DirectionPortA(0,\$FF) ;

Expander_Set_DirectionPortB

Prototype	procedure Expander_Set_DirectionPortB(ModuleAddress, Data : byte) ;
Description	Set port expander PortB pin as input or output.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Set_DirectionPortB(0,\$FF) ;

Expander_Set_PullUpsPortA

Prototype	procedure Expander_Set_PullUpsPortA(ModuleAddress, Data : byte) ;
Description	This routine sets port expander PortA pin as pullup or pulldown.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Set_PullUpsPortA(0,\$FF) ;

Expander_Set_PullUpsPortB

Prototype	procedure Expander_Set_PullUpsPortB(ModuleAddress, Data : byte) ;
Description	This routine sets port expander PortB pin as pullup or pulldown.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	Expander_Set_PullUpsPortB(0,\$FF) ;

Library Example

The example demonstrates how to communicate to port expander MCP23S17.

```
program port_expander_test;

var
  i : byte;

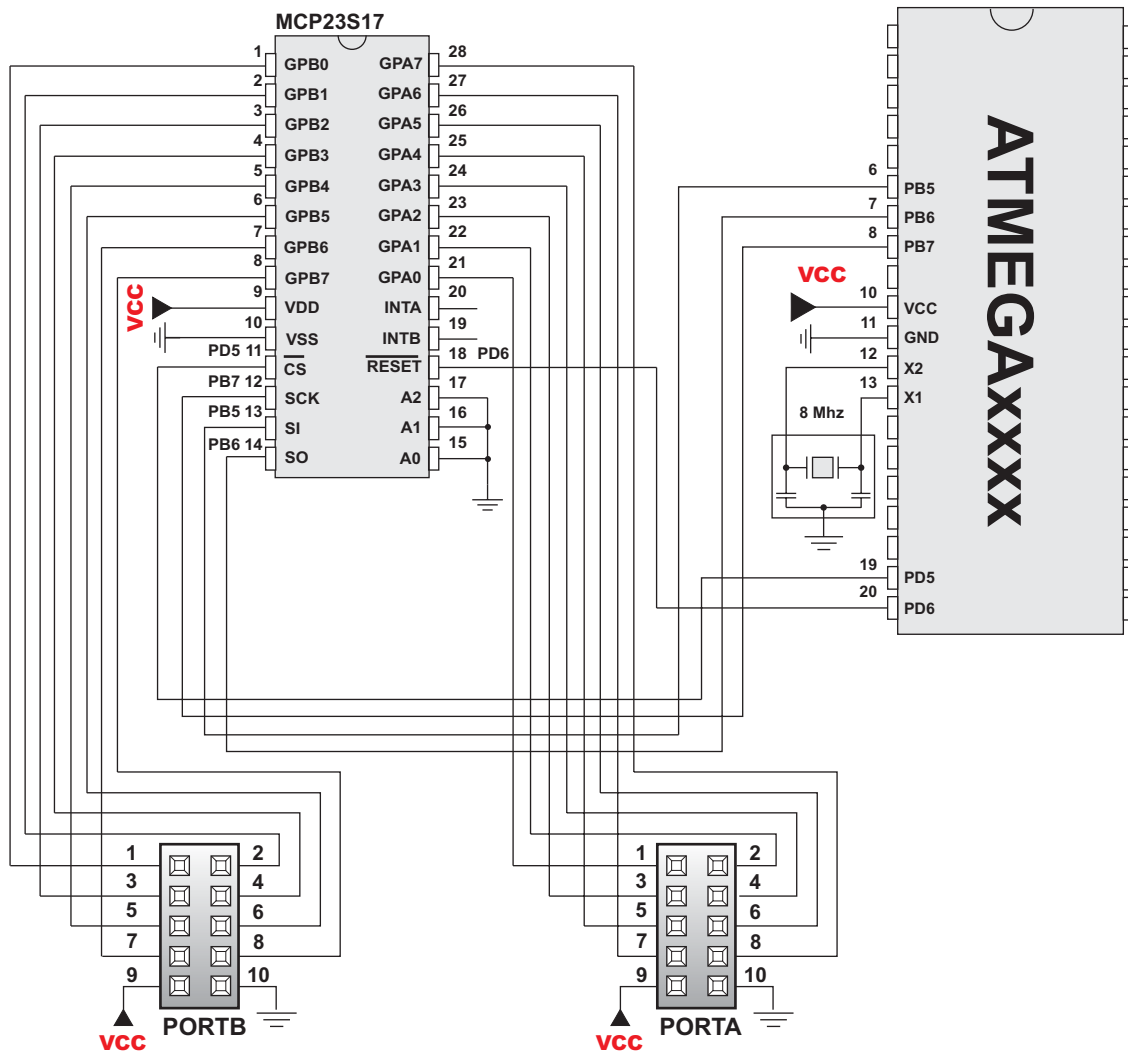
begin
  DDRD := $FF;

  DDRB.7 := 1; // SCK output
  DDRB.5 := 1; // MOSI output
  DDRB.6 := 0; // MISO input
  DDRB.4 := 1; // SS output

  Expander_Init(PORTB, 0, PORTB, 1, 0);
  Expander_Set_DirectionPortA(0, 0);
  Expander_Set_PullUpsPortB(0,$FF);
  Expander_Set_DirectionPortB(0,$FF);

  i := 0;
  while TRUE do
    begin
      Expander_Write_PortA(0, i);
      PORTD:=Expander_Read_PortB(0);
      inc(i);
      Delay_ms(20);
    end;
  end.
```


Hardware Connection



SPI Graphic LCD Library

mikroPascal provides a library for operating the Graphic LCD 128x64 via SPI. These routines work with the common GLCD 128x64 (samsung ks0108).

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

Library Routines

Basic routines:

```
Spi_Glcd_Init  
Spi_Glcd_Disable  
Spi_Glcd_Set_Side  
Spi_Glcd_Set_Page  
Spi_Glcd_Set_X  
Spi_Glcd_Read_Data  
Spi_Glcd_Write_Data
```

Advanced routines:

```
Spi_Glcd_Fill  
Spi_Glcd_Dot  
Spi_Glcd_Line  
Spi_Glcd_V_Line  
Spi_Glcd_H_Line  
Spi_Glcd_Rectangle  
Spi_Glcd_Box  
Spi_Glcd_Circle  
Spi_Glcd_Set_Font  
Spi_Glcd_Write_Char  
Spi_Glcd_Write_Text  
Spi_Glcd_Image
```

Spi_Glcd_Init

Prototype	procedure Spi_Glcd_Init(var RstPort : byte ; RstPin : byte ; var CSPort : byte ; CSPin, DeviceAddress : byte);
Description	Initializes Graphic LCD 128x64 via SPI. RstPort and RstPin - Sets pin connected on reset pin of spi expander. CSPort and CSPin - Sets pin connected on CS pin of spi expander. device address - address of spi expander (hardware setting of A0, A1 and A2 pins (connected on VCC or GND) on spi expander).
Requires	This procedure needs to be called before using other routines of SPI GLCD library.
Example	Spi_Glcd_Init(PORTF, 0, PORTF, 1, 0);

Spi_Glcd_Disable

Prototype	procedure Spi_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 Spi_Glcd_Init.
Example	Spi_Glcd_Disable;

Spi_Glcd_Set_Side

Prototype	procedure Spi_Glcd_Set_Side(x : byte);
Description	Selects side of GLCD, left or right. Parameter x specifies the side: values from 0 to 63 specify the left side, and values higher than 64 specify the right side. Use the functions Spi_Glcd_Set_Side, Spi_Glcd_Set_X, and Spi_Glcd_Set_Page to specify an exact position on GLCD. Then, you can use Spi_Glcd_Write_Data or Spi_Glcd_Read_Data on that location.
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Set_Side(0); // Select the left side of GLCD

Spi_Glcd_Set_Page

Prototype	procedure Spi_Glcd_Set_Page (page : byte);
Description	Selects page of GLCD, technically a line on display; parameter page can be 0..7.
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Set_Page(5);

Spi_Glcd_Set_X

Prototype	procedure Spi_Glcd_Set_X(x : byte);
Description	Positions to x dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Set_X(25);

Spi_Glcd_Read_Data

Prototype	function Spi_Glcd_Read_Data : byte;
Returns	One word from the GLCD memory.
Description	Reads data from from the current location of GLCD memory. Use the functions Spi_Glcd_Set_Side, Spi_Glcd_Set_X, and Spi_Glcd_Set_Page to specify an exact position on GLCD. Then, you can use Spi_Glcd_Write_Data or Spi_Glcd_Read_Data on that location.
Requires	Reads data from from the current location of GLCD memory.
Example	tmp := Spi_Glcd_Read_Data;

Spi_Glcd_Write_Data

Prototype	procedure Spi_Glcd_Write_Data(data : byte);
Description	Writes data to the current location in GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Write_Data(data);

Spi_Glcd_Fill

Prototype	procedure Spi_Glcd_Fill(pattern : byte);
Description	Fills the GLCD memory with byte pattern. To clear the GLCD screen, use Spi_Glcd_Fill(0); to fill the screen completely, use Spi_Glcd_Fill(\$FF).
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Fill(0); <i>// Clear screen</i>

Spi_Glcd_Dot

Prototype	procedure Spi_Glcd_Dot(x, y, color : byte);
Description	Draws a dot on the GLCD at coordinates (x, y). Parameter color determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state.
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Dot(0, 0, 2); <i>// Invert the dot in the upper left corner</i>

Spi_Glcd_Line

Prototype	procedure Spi_Glcd_Line(x1, y1, x2, y2, color : byte);
Description	Draws a line on the GLCD from (x1, y1) to (x2, y2). Parameter color determines the dot state: 0 draws an empty line (clear dots), 1 draws a full line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Line(0, 63, 50, 0, 2);

Spi_Glcd_V_Line

Prototype	procedure Spi_Glcd_V_Line(y1, y2, x, color : byte);
Description	Similar to GLcd_Line, draws a vertical line on the GLCD from (x, y1) to (x, y2).
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_V_Line(0, 63, 0, 1);

Spi_Glcd_H_Line

Prototype	procedure Spi_Glcd_H_Line(x1, x2, y, color : byte);
Description	Similar to GLcd_Line, draws a horizontal line on the GLCD from (x1, y) to (x2, y).
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_H_Line(0, 127, 0, 1);

Spi_Glcd_Rectangle

Prototype	procedure Spi_Glcd_Rectangle(x1, y1, x2, y2, color : byte);
Description	Draws a rectangle on the GLCD. Parameters (x1, y1) set the upper left corner, (x2, y2) set the bottom right corner. Parameter color defines the border: 0 draws an empty border (clear dots), 1 draws a solid border (put dots), and 2 draws a “smart” border (invert each dot).
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Rectangle(10, 0, 30, 35, 1);

Spi_Glcd_Box

Prototype	procedure Spi_Glcd_Box(x1, y1, x2, y2, color : byte);
Description	Draws a box on the GLCD. Parameters (x1, y1) set the upper left corner, (x2, y2) set the bottom right corner. Parameter color defines the fill: 0 draws a white box (clear dots), 1 draws a full box (put dots), and 2 draws an inverted box (invert each dot).
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Box(10, 0, 30, 35, 1);

Spi_Glcd_Circle

Prototype	procedure Spi_Glcd_Circle(x, y, radius, color : integer);
Description	Draws a circle on the GLCD, centered at (x, y) with radius. Parameter color defines the circle line: 0 draws an empty line (clear dots), 1 draws a solid line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Circle(63, 31, 25, 1);

Spi_Glcd_Set_Font

Prototype	procedure Spi_Glcd_Set_Font(font_address : longint; font_width, font_height : byte; font_offset : word);
Description	<p>Sets the font for text display routines, Spi_Glcd_Write_Char and Spi_Glcd_Write_Text. Font needs to be formatted as an array of byte. Parameter font_address specifies the address of the font; you can pass a font name with the @ operator. Parameters font_width and font_height specify the width and height of characters in dots. Font width should not exceed 128 dots, and font height should not exceed 8 dots. Parameter font_offset determines the ASCII character from which the supplied font starts. Demo fonts supplied with the library have an offset of 32, which means that they start with space.</p> <p>If no font is specified, Spi_Glcd_Write_Char and Spi_Glcd_Write_Text will use the default 5x8 font supplied with the library. You can create your own fonts by following the guidelines given in the file “GLCD_Fonts.dpas”. This file contains the default fonts for GLCD, and is located in your installation folder, “Extra Examples” > “GLCD”.</p>
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	<pre>// Use the custom 5x7 font "myfont" which starts with space (32): Spi_Glcd_Set_Font(@myfont, 5, 7, 32);</pre>

Spi_Glcd_Write_Char

Prototype	procedure Spi_Glcd_Write_Char(character, x, page, color : 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 Spi_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 Spi_Glcd_Init. Use the Spi_Glcd_Set_Font to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	<pre>Spi_Glcd_Write_Char('C', 0, 0, 1);</pre>

Spi_Glcd_Write_Text

Prototype	procedure Spi_Glcd_Write_Text(text : string [20] ; x, page, color : byte) ;
Description	<p>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).</p> <p>Use routine Spi_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 Spi_Glcd_Init. Use the Spi_Glcd_Set_Font to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	Spi_Glcd_Write_Text('Hello world!', 0, 0, 1);

Spi_Glcd_Image

Prototype	procedure Spi_Glcd_Image(image : array [0..1023] of byte);
Description	Displays bitmap image on the GLCD. Parameter image should be formatted as an array of 1024 bytes. Use the mikroPascal’s integrated Bitmap-to-LCD editor (menu option Tools > Graphic LCD Editor) to convert image to a constant array suitable for display on GLCD.
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	Spi_Glcd_Image(my_image);

Library Example

The example demonstrates how to communicate to KS0108 GLCD via SPI module, using serial to parallel convertor MCP23S17.

```

program glcd_over_spi;

uses images, fonts;

var
    i: byte;
    someText: string[ 20] ;

procedure wait;
begin
    delay_ms(1200);
end;

begin
    DDRB.7 := 1;  // SCK output
    DDRB.5 := 1;  // MOSI output
    DDRB.6 := 0;  // MISO input
    DDRB.4 := 1;  // SS output

    Spi_Glcd_Init(PORTB, 2, PORTB, 1, 0);
    wait;
    Spi_Glcd_Fill(0xAA);
    wait;

    while TRUE do
        begin
            SPI_Glcd_Image(mikro_logo_bmp);
            wait;
            SPI_Glcd_Image(logo_atmel_bmp);
            wait;
            SPI_Glcd_Fill($00);
            SPI_Glcd_Write_Text('mikroPascal compiler',6,1,1);
            SPI_Glcd_Write_Text('for Atmel AVR MCU'      ,12,3,1);
            SPI_Glcd_Write_Text('by',55,5,1);
            SPI_Glcd_Write_Text('mikroElektronika',15,7,1);
            wait;

            SPI_Glcd_Fill($00);

            //continues..

```

```
        //continues..

SPI_Glcd_Box( 0,50, 10, 63, 1);
SPI_Glcd_Rectangle( 10,20, 20, 63, 1);
SPI_Glcd_Box( 20,40, 30, 63, 1);
SPI_Glcd_Rectangle( 30,10, 40, 63, 1);
SPI_Glcd_Box( 40,30, 50, 63, 1);
SPI_Glcd_Rectangle( 50,50, 60, 63, 1);
SPI_Glcd_Box( 60,40, 70, 63, 1);
SPI_Glcd_Rectangle( 70,10, 80, 63, 1);
SPI_Glcd_Box( 80,20, 90, 63, 1);
SPI_Glcd_Rectangle( 90, 5,100, 63, 1);
SPI_Glcd_Box(100,40,110, 63, 1);
SPI_Glcd_Rectangle(100,50,120, 63, 1);
wait;
SPI_Glcd_Fill($00);
i:=0;
while i<30 do
    begin
        SPI_Glcd_Rectangle(i,i,127-i, 63-i, 1);
        i:=i+5;
    end;

wait;
SPI_Glcd_Fill($00);

i:=1;
while i<31 do
    begin
        SPI_Glcd_Circle(63, 31, i, 1);
        i:=i+5;
    end;

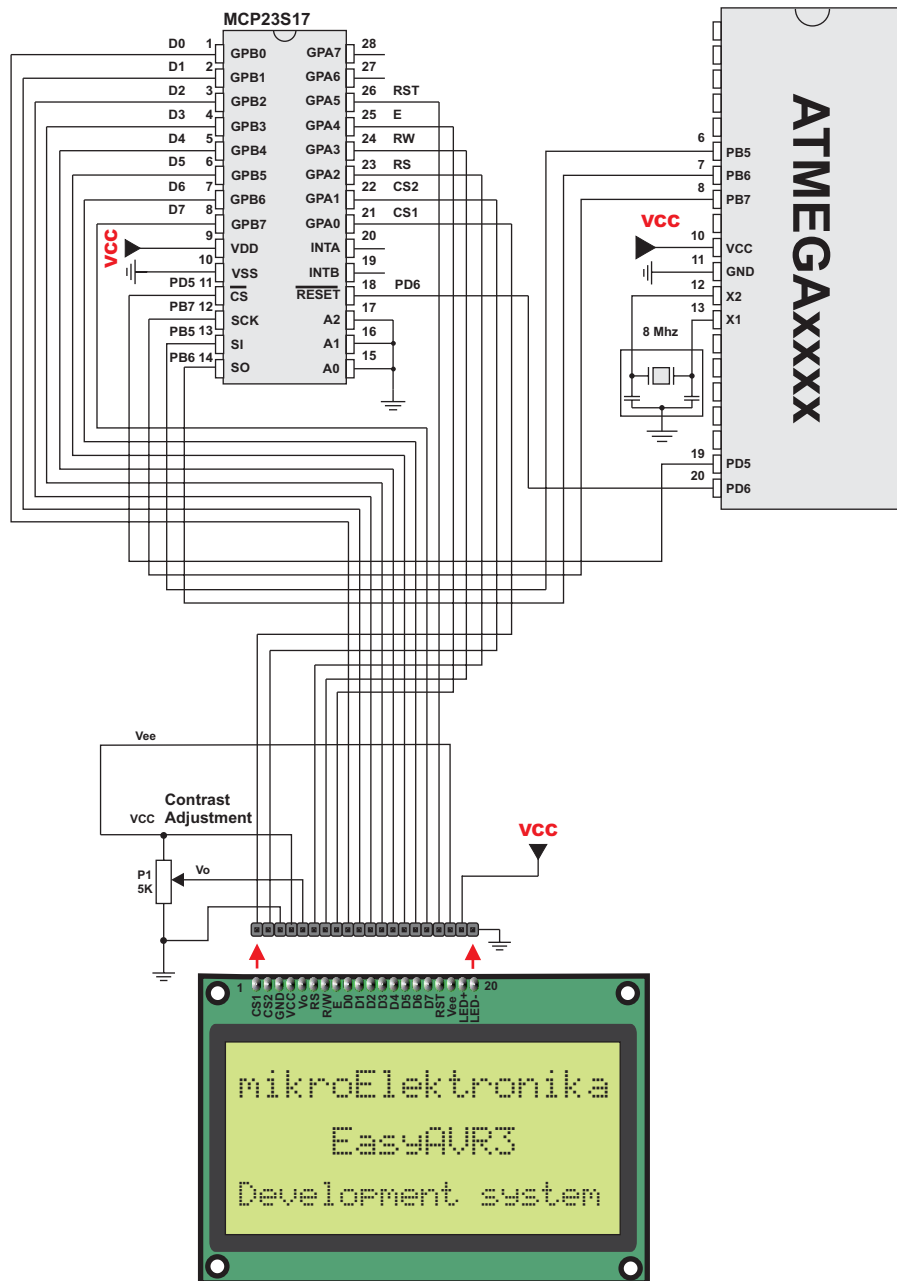
wait;
SPI_Glcd_Fill($FF);
wait;

SPI_Glcd_Line(35,50,63,10,0);
SPI_Glcd_Line(63,10,91,50,0);
SPI_Glcd_Line(35,50,91,50,0);

SPI_Glcd_Line(63,10,30,35,0);
SPI_Glcd_Line(35,50,30,35,0);

wait;
end;
end.
```

Hardware Connection



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 mikroPascal, 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 mikroPascal for purely educational purposes, please contact us.

Problems with transport or delivery

If you want to report a delay in delivery or any other problem concerning distribution of our products, please use the link given below.

Would you like to become mikroElektronika's distributor?

We in mikroElektronika are looking forward to new partnerships. If you would like to help us by becoming distributor of our products, please let us know.

Other

If you have any other question, comment or a business proposal, please contact us:

mikroElektronika
Admirala Geprata 1B
11000 Belgrade
EUROPE

Phone: + 381 (11) 30 66 377, + 381 (11) 30 66 378

Fax: + 381 (11) 30 66 379

E-mail: office@mikroe.com

Web: www.mikroe.com