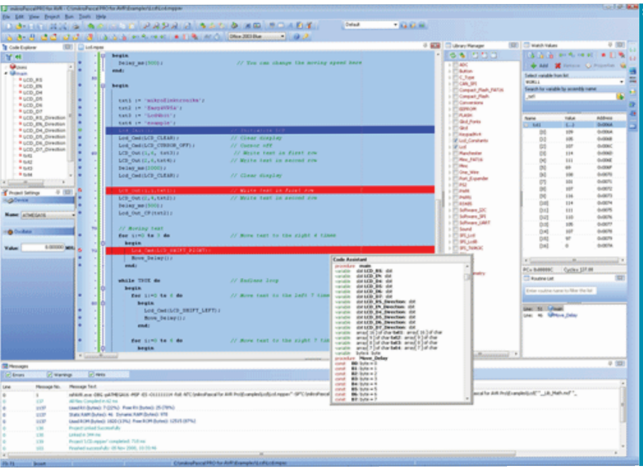
MikroElektronika
SOFTWARE AND HARDWARE SOLUTIONS FOR EMBEDDED WORLD ...making it simple

mikroPASCAL PRO for AVR



```
begin
  Delay_ms(500);                        // You can change the moving speed here
end;

begin
  txt1 := 'mikroElektronika';
  txt2 := 'EasyAVR6a';
  txt3 := 'Lcd4bit';
  txt4 := 'example';
  Lcd_Init();                           // Initialize LCD
  Lcd_Cmd(LCD_CLEAR);                   // Clear display
  Lcd_Cmd(LCD_CURSOR_OFF);              // Cursor off
  LCD_Out(1,6, txt3);                   // Write text in first row
  LCD_Out(2,6,txt4);                    // Write text in second row
  Delay_ms(2000);
  Lcd_Cmd(LCD_CLEAR);                   // Clear display

  LCD_Out(1,1,txt1);                    // Write text in first row
  LCD_Out(2,4,txt2);                    // Write text in second row
  Delay_ms(500);
  Lcd_Out_CP(txt2);

  // Moving text
  for i:=0 to 3 do                      // Move text to the right 4 times
    begin
      Lcd_Cmd(LCD_SHIFT_RIGHT);
      Move_Delay();
    end;

  while TRUE do                         // Endless loop
    begin
      for i:=0 to 6 do                  // Move text to the left 7 times
        begin
          Lcd_Cmd(LCD_SHIFT_LEFT);
          Move_Delay();
        end;

      for i:=0 to 6 do                  // Move text to the right 7 times
        begin
```

USER MANUAL

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

May 2009.

Reader’s note

DISCLAIMER:
mikroPASCAL PRO for AVR 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 mikroEelktronika. 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 PRO for AVR* 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.

Reader’s Note

LICENSE AGREEMENT:
By using the *mikroPASCAL PRO for AVR* compiler, you agree to the terms of this agreement. Only one person may use licensed version of *mikroPascal PRO for AVR* compiler at a time.

This manual covers *mikroPASCAL PRO for AVR* version 1.2.5 and the related topics. Newer 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 next information in your bug report:

- Your operating system
- Version of *mikroPASCAL PRO for AVR*
- Code sample
- Description of a bug

CONTACT US:
mikroElektronika

Voice:	+ 381 (11) 36 28 830
Fax:	+ 381 (11) 36 28 831
Web:	www.mikroe.com
E-mail:	office@mikroe.com

Table of Contents

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

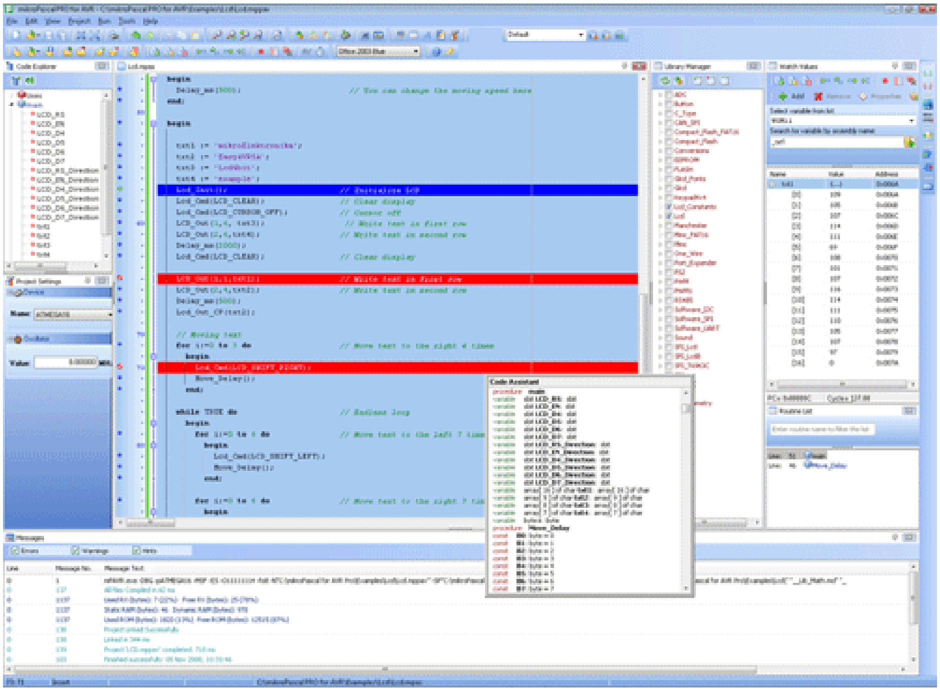
CHAPTER 6

CHAPTER 1

Introduction to mikroPascal PRO for AVR

Help version: 2009/05/18

The mikroPascal PRO for AVR is a powerful, feature-rich development tool for AVR microcontrollers. It is designed to provide the programmer with the easiest possible solution to developing applications for embedded systems, without compromising performance or control.



Introduction to mikroPascal PRO for AVR

Features

- mikroPascal PRO for AVR allows you to quickly develop and deploy complex applications:
- Write your Pascal source code using the built-in Code Editor (Code and Parameter Assistants, Code Folding, Syntax Highlighting, Auto Correct, Code Templates, and more.)
- Use included mikroPascal PRO for AVR libraries to dramatically speed up the development: data acquisition, memory, displays, conversions, communication etc.
- Monitor your program structure, variables, and functions in the Code Explorer.
- Generate commented, human-readable assembly, and standard HEX compatible with all programmers.
- Inspect program flow and debug executable logic with the integrated Software Simulator.
- Get detailed reports and graphs: RAM and ROM map, code statistics, assembly listing, calling tree, and more.
- mikroPascal PRO for AVR provides plenty of examples to expand, develop, and use as building bricks in your projects. Copy them entirely if you deem fit – that's why we included them with the compiler.

Where to Start

- In case that you’re a beginner in programming AVR microcontrollers, read carefully the AVR Specifics chapter. It might give you some useful pointers on AVR constraints, code portability, and good programming practices.

- If you are experienced in Pascal programming, you will probably want to consult mikroPascal PRO for AVR Specifics first. For language issues, you can always refer to the comprehensive Language Reference. A complete list of included libraries is available at mikroPascal PRO for AVR Libraries.

- If you are not very experienced in Pascal programming, don’t panic! mikroPascal PRO for AVR provides plenty of examples making it easy for you to go quickly. We suggest that you first consult Projects and Source Files, and then start browsing the examples that you're the most interested in.

MIKROELEKTRONIKA ASSOCIATES LICENSE STATEMENT AND LIMITED WARRANTY

IMPORTANT - READ CAREFULLY

This license statement and limited warranty constitute a legal agreement (“License Agreement”) between you (either as an individual or a single entity) and mikroElektronika (“mikroElektronika Associates”) for software product (“Software”) identified above, including any software, media, and accompanying on-line or printed documentation.

BY INSTALLING, COPYING, OR OTHERWISE USING SOFTWARE, YOU AGREE TO BE BOUND BY ALL TERMS AND CONDITIONS OF THE LICENSE AGREEMENT.

Upon your acceptance of the terms and conditions of the License Agreement, mikroElektronika Associates grants you the right to use Software in a way provided below.

This Software is owned by mikroElektronika Associates and is protected by copyright law and international copyright treaty. Therefore, you must treat this Software like any other copyright material (e.g., a book).

You may transfer Software and documentation on a permanent basis provided. You retain no copies and the recipient agrees to the terms of the License Agreement. Except as provided in the License Agreement, you may not transfer, rent, lease, lend, copy, modify, translate, sublicense, time-share or electronically transmit or receive Software, media or documentation. You acknowledge that Software in the source code form remains a confidential trade secret of mikroElektronika Associates and therefore you agree not to modify Software or attempt to reverse engineer, decompile, or disassemble it, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.

If you have purchased an upgrade version of Software, it constitutes a single product with the mikroElektronika Associates software that you upgraded. You may use the upgrade version of Software only in accordance with the License Agreement.

LIMITED WARRANTY

Respectfully excepting the Redistributables, which are provided “as is”, without warranty of any kind, mikroElektronika Associates warrants that Software, once updated and properly used, will perform substantially in accordance with the accompanying documentation, and Software media will be free from defects in materials and workmanship, for a period of ninety (90) days from the date of receipt. Any implied warranties on Software are limited to ninety (90) days.

mikroElektronika Associates’ and its suppliers’ entire liability and your exclusive remedy shall be, at mikroElektronika Associates’ option, either (a) return of of the price paid, or (b) repair or replacement of Software that does not meet mikroElektronika Associates' Limited Warranty and which is returned to mikroElektronika Associates with a copy of your receipt. DO NOT RETURN ANY PRODUCT UNTIL YOU HAVE CALLED MIKROELEKTRONIKA ASSOCIATES FIRST AND OBTAINED A RETURN AUTHORIZATION NUMBER. This Limited Warranty is void if failure of Software has resulted from an accident, abuse, or misapplication. Any replacement of Software will be warranted for the rest of the original warranty period or thirty (30) days, whichever is longer.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, MIKROELEKTRONIKA ASSOCIATES AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESSED OR IMPLIED, INCLUDED, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT, WITH REGARD TO SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES.

IN NO EVENT SHALL MIKROELEKTRONIKA ASSOCIATES OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS AND BUSINESS INFORMATION, BUSINESS INTERRUPTION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MIKROELEKTRONIKA ASSOCIATES HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MIKROELEKTRONIKA ASSOCIATES' ENTIRE LIABILITY UNDER ANY PROVISION OF THIS LICENSE AGREEMENT SHALL BE LIMITED TO THE AMOUNT ACTUALLY PAID BY YOU FOR SOFTWARE PRODUCT PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MIKROELEKTRONIKA ASSOCIATES SUPPORT SERVICES AGREEMENT, MIKROELEKTRONIKA ASSOCIATES' ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT.

HIGH RISK ACTIVITIES

Software 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 Software could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities"). mikroElektronika Associates and its suppliers specifically disclaim any expressed or implied warranty of fitness for High Risk Activities.

GENERAL PROVISIONS

This statement may only be modified in writing signed by you and an authorised officer of mikroElektronika Associates. If any provision of this statement is found void or unenforceable, the remainder will remain valid and enforceable according to its terms. If any remedy provided is determined to have failed for its essential purpose, all limitations of liability and exclusions of damages set forth in the Limited Warranty shall remain in effect.

This statement gives you specific legal rights; you may have others, which vary, from country to country. mikroElektronika Associates reserves all rights not specifically granted in this statement.

mikroElektronika
Visegradska 1A,
11000 Belgrade,
Europe.

Phone: + 381 11 36 28 830
Fax: +381 11 36 28 831
Web: www.mikroe.com
E-mail: office@mikroe.com

TECHNICAL SUPPORT

In case you encounter any problem, you are welcome to our support forums at www.mikroe.com/forum/. Here, you may also find helpful information, hardware tips, and practical code snippets. Your comments and suggestions on future development of the mikroPascal PRO for AVR are always appreciated — feel free to drop a note or two on our Wishlist.

In our Knowledge Base www.mikroe.com/en/kb/ you can find the answers to Frequently Asked Questions and solutions to known problems. If you can not find the solution to your problem in Knowledge Base then report it to Support Desk www.mikroe.com/en/support/. In this way, we can record and track down bugs more efficiently, which is in our mutual interest. We respond to every bug report and question in a suitable manner, ever improving our technical support.

HOW TO REGISTER

The latest version of the mikroPascal PRO for AVR is always available for downloading from our website. It is a fully functional software libraries, examples, and comprehensive help included.

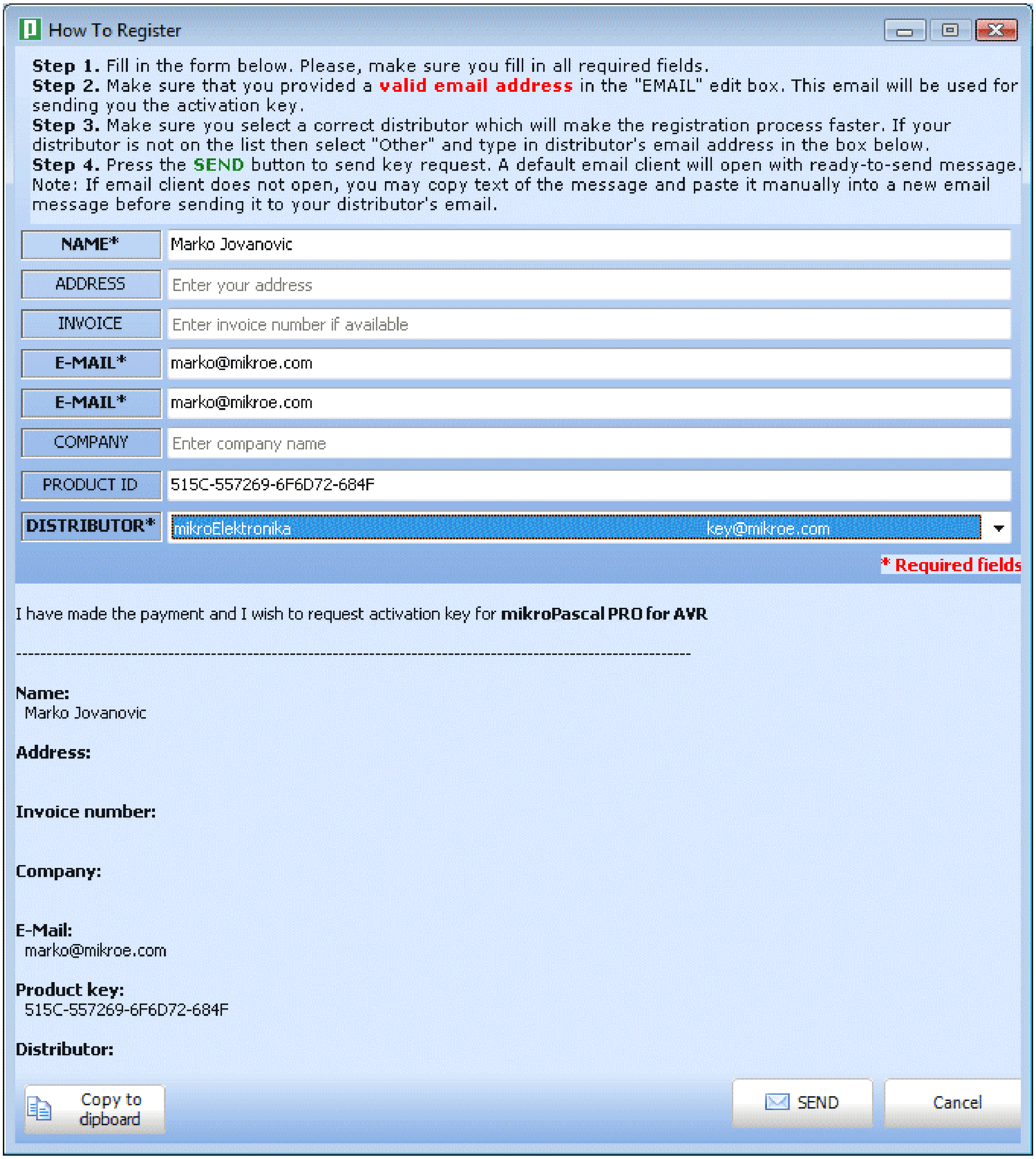
The only limitation of the free version is that it cannot generate hex output over 2 KB. Although it might sound restrictive, this margin allows you to develop practical, working applications with no thinking of demo limit. If you intend to develop really complex projects in the mikroPascal PRO for AVR, then you should consider the possibility of purchasing the license key.

Who Gets the License Key

Buyers of the mikroPascal PRO for AVR are entitled to the license key. After you have completed the payment procedure, you have an option of registering your mikroPascal. In this way you can generate hex output without any limitations.

How to Get License Key

After you have completed the payment procedure, start the program. Select **Help › How to Register** from the drop-down menu or click the How To Register Icon 🔑 Fill out the registration form (figure below), select your distributor, and click the Send button.



How To Register

Step 1. Fill in the form below. Please, make sure you fill in all required fields.
Step 2. Make sure that you provided a **valid email address** in the "EMAIL" edit box. This email will be used for sending you the activation key.
Step 3. Make sure you select a correct distributor which will make the registration process faster. If your distributor is not on the list then select "Other" and type in distributor's email address in the box below.
Step 4. Press the **SEND** button to send key request. A default email client will open with ready-to-send message.
Note: If email client does not open, you may copy text of the message and paste it manually into a new email message before sending it to your distributor's email.

NAME*	Marko Jovanovic
ADDRESS	Enter your address
INVOICE	Enter invoice number if available
E-MAIL*	marko@mikroe.com
E-MAIL*	marko@mikroe.com
COMPANY	Enter company name
PRODUCT ID	515C-557269-6F6D72-684F
DISTRIBUTOR*	mikroElektronika key@mikroe.com

*** Required fields**

I have made the payment and I wish to request activation key for **mikroPascal PRO for AVR**

Name:
Marko Jovanovic

Address:

Invoice number:

Company:

E-Mail:
marko@mikroe.com

Product key:
515C-557269-6F6D72-684F

Distributor:

This will start your e-mail client with message ready for sending. Review the information you have entered, and add the comment if you deem it necessary. Please, do not modify the subject line.

Upon receiving and verifying your request, we will send the license key to the e-mail address you specified in the form.

After Receving the License Key

The license key comes as a small autoextracting file – just start it anywhere on your computer in order to activate your copy of compiler and remove the demo limit. You do not need to restart your computer or install any additional components. Also, there is no need to run the mikroPascal PRO for AVR at the time of activation.

Notes:

- The license key is valid until you format your hard disk. In case you need to format the hard disk, you should request a new activation key.

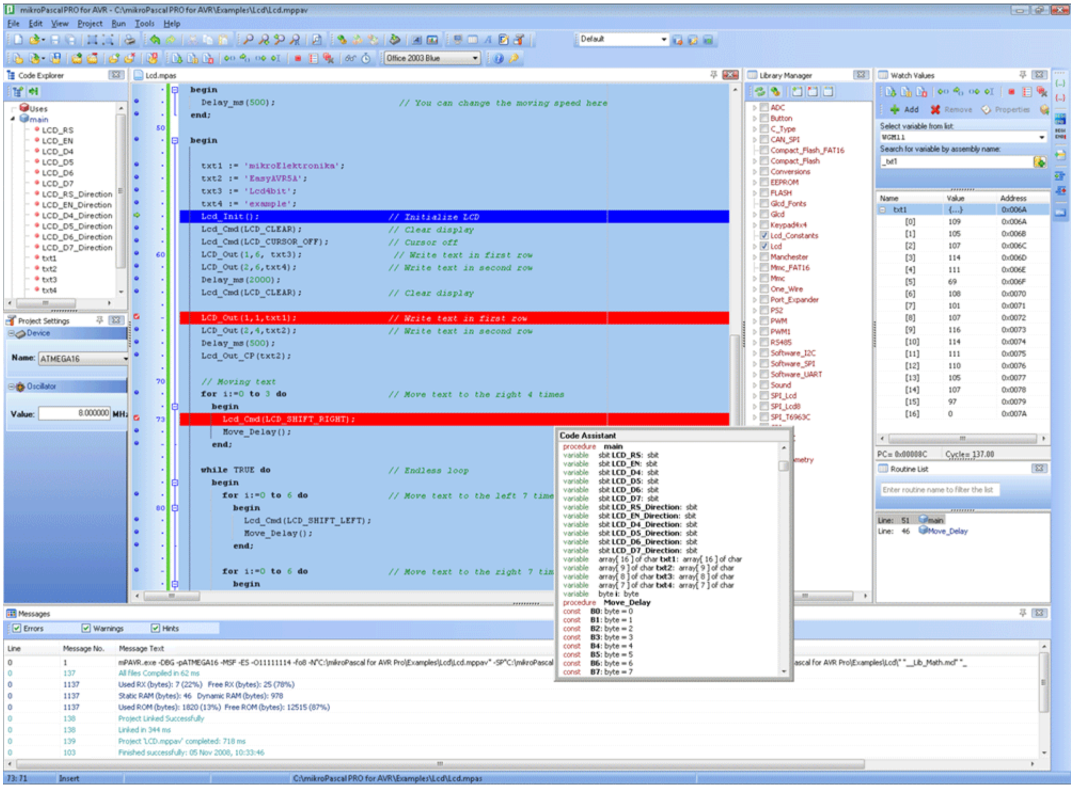
- Please keep the activation program in a safe place. Every time you upgrade the compiler you should start this program again in order to reactivate the license.

CHAPTER 2

mikroPascal PRO for AVR Environment

IDE OVERVIEW

The mikroPascal PRO for AVR is an user-friendly and intuitive environment:



Name: ATMEGA16

Value: 8.000000 MHz

```
begin
  Delay_ms(500);                    // You can change the moving speed here
end;

begin

  txt1 := 'mikroElektronika';
  txt2 := 'EasyAVR5A';
  txt3 := 'Lcd4bit';
  txt4 := 'example';
  Lcd_Init();                      // Initialize LCD
  Lcd_Cmd(LCD_CLEAR);              // Clear display
  Lcd_Cmd(LCD_CURSOR_OFF);         // Cursor off
  LCD_Out(1,6, txt3);               // Write text in first row
  LCD_Out(2,6,txt4);               // Write text in second row
  Delay_ms(2000);
  Lcd_Cmd(LCD_CLEAR);              // Clear display

  LCD_Out(1,1,txt1);               // Write text in first row
  LCD_Out(2,4,txt2);               // Write text in second row
  Delay_ms(500);
  Lcd_Out_CP(txt2);

  // Moving text
  for i:=0 to 3 do                 // Move text to the right 4 times
    begin
      Lcd_Cmd(LCD_SHIFT_RIGHT);
      Move_Delay();
    end;

  while TRUE do                    // Endless loop
    begin
      for i:=0 to 6 do             // Move text to the left 7 time
        begin
          Lcd_Cmd(LCD_SHIFT_LEFT);
          Move_Delay();
        end;

      for i:=0 to 6 do             // Move text to the right 7 tim
        begin
```

Code Assistant

```
procedure  main
variable   sbit LCD_RS: sbit
variable   sbit LCD_EN: sbit
variable   sbit LCD_D4: sbit
variable   sbit LCD_D5: sbit
variable   sbit LCD_D6: sbit
variable   sbit LCD_D7: sbit
variable   sbit LCD_RS_Direction: sbit
variable   sbit LCD_EN_Direction: sbit
variable   sbit LCD_D4_Direction: sbit
variable   sbit LCD_D5_Direction: sbit
variable   sbit LCD_D6_Direction: sbit
variable   sbit LCD_D7_Direction: sbit
variable   array[ 16 ] of char txt1: array[ 16 ] of char
variable   array[ 9 ] of char txt2: array[ 9 ] of char
variable   array[ 8 ] of char txt3: array[ 8 ] of char
variable   array[ 7 ] of char txt4: array[ 7 ] of char
variable   byte i: byte
procedure  Move_Delay
const      B0: byte = 0
const      B1: byte = 1
const      B2: byte = 2
const      B3: byte = 3
const      B4: byte = 4
const      B5: byte = 5
const      B6: byte = 6
const      B7: byte = 7
```

Select variable from list: WGM11

Search for variable by assembly name: _txt1

Name	Value	Address
txt1	{...}	0x006A
[0]	109	0x006A
[1]	105	0x006B
[2]	107	0x006C
[3]	114	0x006D
[4]	111	0x006E
[5]	69	0x006F
[6]	108	0x0070
[7]	101	0x0071
[8]	107	0x0072
[9]	116	0x0073
[10]	114	0x0074
[11]	111	0x0075
[12]	110	0x0076
[13]	105	0x0077
[14]	107	0x0078
[15]	97	0x0079
[16]	0	0x007A

PC= 0x00008C Cycle= 137.00

Enter routine name to filter the list

Line: 51 main
Line: 46 Move_Delay

☑ Errors ☑ Warnings ☑ Hints

```
Line  Message No.  Message Text
0     1            mPAVR.exe -DBG -pATMEGA16 -MSF -ES -O11111114 -fo8 -N"C:\mikroPascal for AVR Pro\Examples\Lcd\Lcd.mppav" -SP"C:\mikroPascal ... ascal for AVR Pro\Examples\Lcd\" "__Lib_Math.mcl" "_
0     137          All files Compiled in 62 ms
0     1137         Used RX (bytes): 7 (22%)  Free RX (bytes): 25 (78%)
0     1137         Static RAM (bytes): 46  Dynamic RAM (bytes): 978
0     1137         Used ROM (bytes): 1820 (13%)  Free ROM (bytes): 12515 (87%)
0     138          Project Linked Successfully
0     138          Linked in 344 ms
0     139          Project 'LCD.mppav' completed: 718 ms
0     103          Finished successfully: 05 Nov 2008, 10:33:46
```

73: 71 Insert C:\mikroPascal PRO for AVR\Examples\Lcd\Lcd.mpas

- The Code Editor features adjustable Syntax Highlighting, Code Folding, Code Assistant, Parameters Assistant, Spell Checker, Auto Correct for common typos and Code Templates (Auto Complete).
- The Code Explorer (with Keyboard shortcut browser and Quick Help browser) is at your disposal for easier project management.
- The Project Manager allows multiple project management
- General project settings can be made in the Project Settings window
- Library manager enables simple handling libraries being used in a project
- The Error Window displays all errors detected during compiling and linking.
- The source-level Software Simulator lets you debug executable logic step-by-step by watching the program flow.
- The New Project Wizard is a fast, reliable, and easy way to create a project.
- Help files are syntax and context sensitive.
- Like in any modern Windows application, you may customize the layout of mikroPascal PRO for AVR to suit your needs best.

- Spell checker underlines identifiers which are unknown to the project. In this way it helps the programmer to spot potential problems early, much before the project is compiled.
- Spell checker can be disabled by choosing the option in the Preferences dialog (F12).

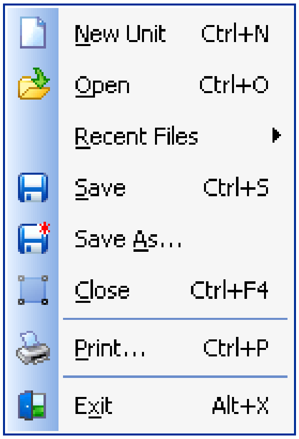
MAIN MENU OPTIONS

Available Main Menu options are:

Related topics: Keyboard shortcuts

File Menu Options

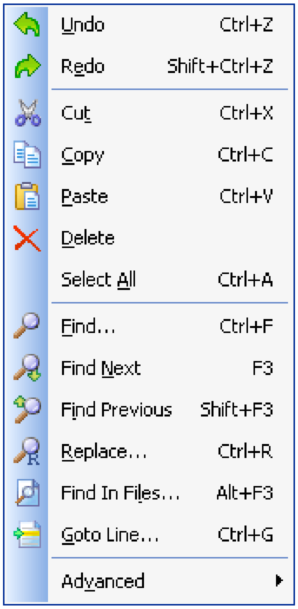
The File menu is the main entry point for manipulation with the source files.



File	Description
New Unit Ctrl+N	Open a new editor window.
Open Ctrl+O	Open source file for editing or image file for viewing.
Recent Files ▸	Reopen recently used file.
Save Ctrl+S	Save changes for active editor.
Save As...	Save the active source file with the different name or change the file type.
Close Alt+F4	Close active source file.
Print... Ctrl+P	Print Preview.
Exit Alt+X	Exit IDE.

Related topics: Keyboard shortcuts, File Toolbar, Managing Source Files

Edit Menu Options



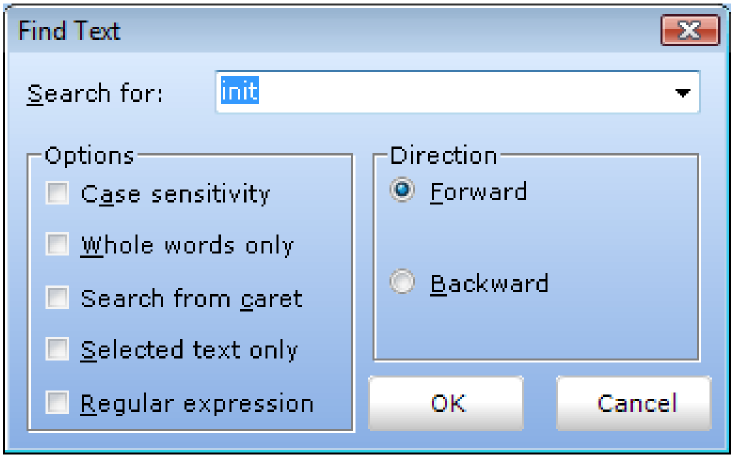
File	Description
Undo Ctrl+Z	Undo last change.
Redo Shift+Ctrl+Z	Redo last change.
Cut Ctrl+X	Cut selected text to clipboard.
Copy Ctrl+C	Copy selected text to clipboard.
Paste Ctrl+V	Paste text from clipboard.
Delete	Delete selected text.
Select All Ctrl+A	Select all text in active editor.
Find... Ctrl+F	Find text in active editor.
Find Next F3	Find next occurence of text in active editor.
Find Previous Shift+F3	Find previous occurence of text in active editor.
Replace... Ctrl+R	Replace text in active editor.

Find In Files... Alt+F3	Find text in current file, in all opened files, or in files from desired folder.
Goto Line... Ctrl+G	Goto to the desired line in active editor.
Advanced ▸	Advanced Code Editor options

Advanced »	Description
{..} Comment Shift+Ctrl+.	Comment selected code or put single line comment if there is no selection.
{..} Uncomment Shift+Ctrl+,	Uncomment selected code or remove single line comment if there is no selection.
Indent Shift+Ctrl+I	Indent selected code.
Outdent Shift+Ctrl+U	Outdent selected code.
Lowercase Ctrl+Alt+L	Changes selected text case to lowercase.
Uppercase Ctrl+Alt+U	Changes selected text case to uppercase.
Titlecase Ctrl+Alt+T	Changes selected text case to titlercase.

Find Text

Dialog box for searching the document for the specified text. The search is performed in the direction specified. If the string is not found a message is displayed.



Find Text

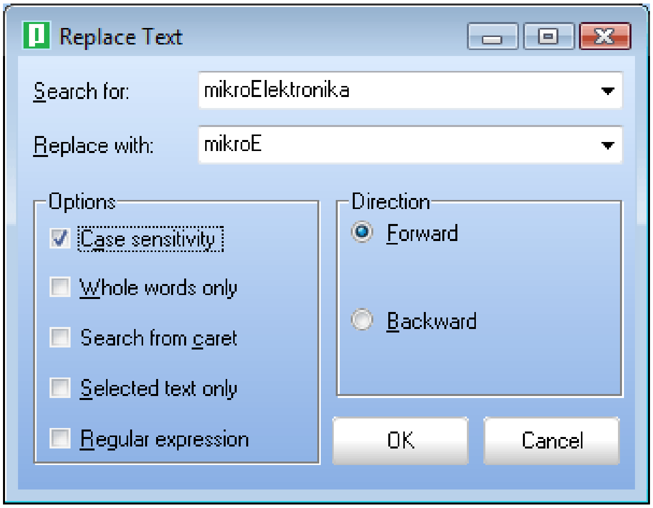
Search for: init

Options
- ☐ Case sensitivity
- ☐ Whole words only
- ☐ Search from caret
- ☐ Selected text only
- ☐ Regular expression

Direction
- ● Forward
- ○ Backward

Replace Text

Dialog box for searching for a text string in file and replacing it with another text string.



Replace Text

Search for: mikroElektronika

Replace with: mikroE

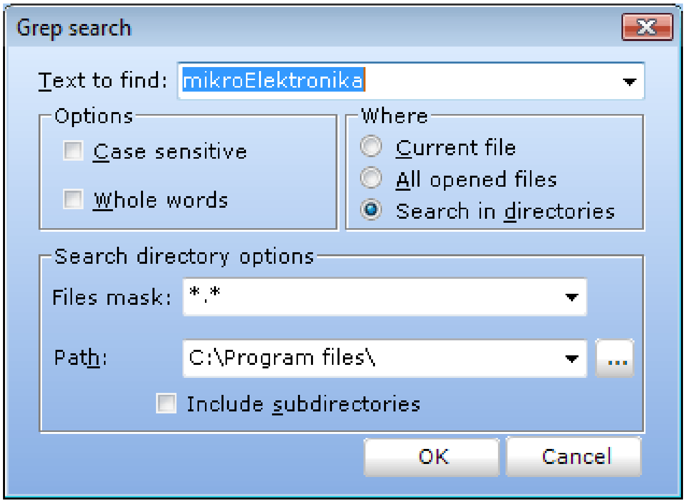
Options
☑ Case sensitivity
□ Whole words only
□ Search from caret
□ Selected text only
□ Regular expression

Direction
● Forward
○ Backward

Find In Files

Dialog box for searching for a text string in current file, all opened files, or in files on a disk.

The string to search for is specified in the **Text to find** field. If Search in directories option is selected, The files to search are specified in the **Files mask** and **Path** fields.



Grep search

Text to find: mikroElektronika

Options
□ Case sensitive
□ Whole words

Where
○ Current file
○ All opened files
● Search in directories

Search directory options

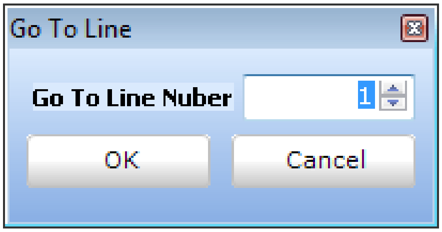
Files mask: *.*

Path: C:\Program files\

□ Include subdirectories

Go To Line

Dialog box that allows the user to specify the line number at which the cursor should be positioned.

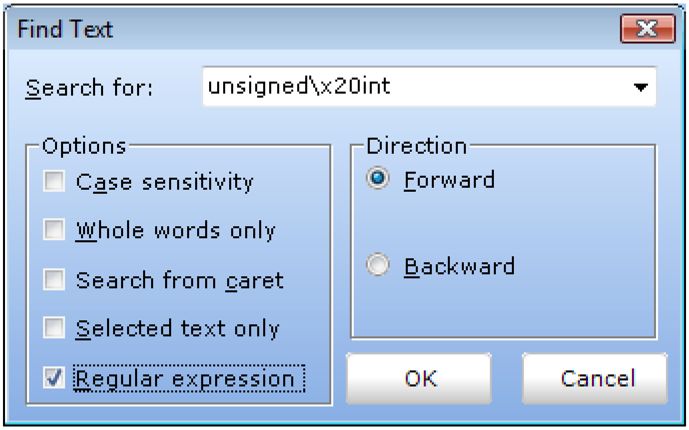


Go To Line

Go To Line Nuber: 1

Regular expressions

By checking this box, you will be able to advance your search, through Regular expressions.



Find Text

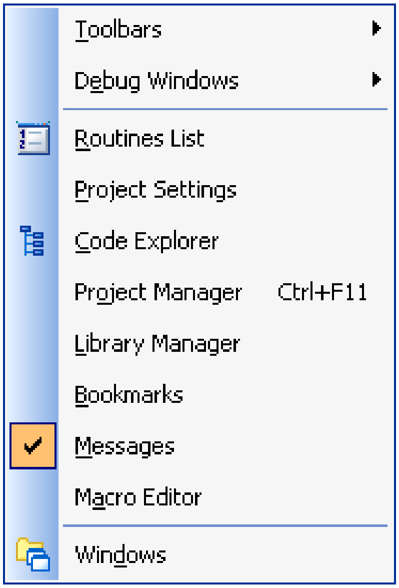
Search for: unsigned\x20int

Options
- □ Case sensitivity
- □ Whole words only
- □ Search from caret
- □ Selected text only
- ☑ Regular expression

Direction
- ● Forward
- ○ Backward

Related topics: Keyboard shortcuts, Edit Toolbar, Advanced Edit Toolbar

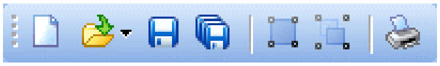
View Menu Options



File	Description
Toolbars ▸	Show/Hide toolbars.
Debug Windows	Show/Hide debug windows.
Routines List	Show/Hide Routine List in active editor.
Project Settings	Show/Hide Project Settings window.
Code Explorer	Show/Hide Code Explorer window.
Project Manager Shift+Ctrl+F11	Show/Hide Project Manager window.
Library Manager	Show/Hide Library Manager window.
Bookmarks	Show/Hide Bookmarks window.
Messages	Show/Hide Error Messages window.
Macro Editor	Show/Hide Macro Editor window.
Windows	Show Window List window.

TOOLBARS

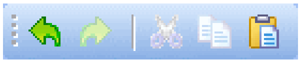
File Toolbar



File Toolbar is a standard toolbar with following options:

Icon	Description
🗋	Opens a new editor window.
📂	Open source file for editing or image file for viewing.
💾	Save changes for active window.
💾	Save changes in all opened windows.
☐	Close current editor.
⧉	Close all editors.
🖨	Print Preview.

Edit Toolbar



Edit Toolbar is a standard toolbar with following options:

Icon	Description
↶	Undo last change.
↷	Redo last change.
✂	Cut selected text to clipboard.
⎘	Copy selected text to clipboard.
📋	Paste text from clipboard.

Advanced Edit Toolbar

Advanced Edit Toolbar comes with following options:

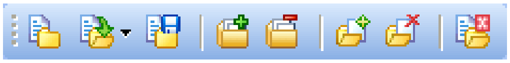
Icon	Description
{..}	Comment selected code or put single line comment if there is no selection
{..}	Uncomment selected code or remove single line comment if there is no selection.
BEGI END	Select text from starting delimiter to ending delimiter.
BEGI END	Go to ending delimiter.
➜	Go to line.
⇥	Indent selected code lines.
⇤	Outdent selected code lines.
HTML	Generate HTML code suitable for publishing current source code on the web.

Find/Replace Toolbar

Find/Replace Toolbar is a standard toolbar with following options:

Icon	Description
🔍	Find text in current editor.
🔍	Find next occurence.
🔍	Find previous occurence.
R	Replace text.
🔍	Find text in files.

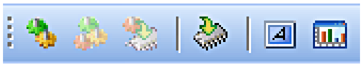
Project Toolbar



Project Toolbar comes with following options:

Icon	Description
	Open new project wizard. wizard.
	Open Project
	Save Project
	Add existing project to project group.
	Remove existing project from project group.
	Add File To Project
	Remove File From Project
	Close current project.

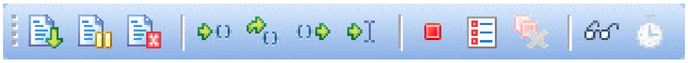
Build Toolbar



Build Toolbar comes with following options:

Icon	Description
	Build current project.
	Build all opened projects.
	Build and program active project.
	Start programmer and load current HEX file.
	Open assembly code in editor.
	View statistics for current project.

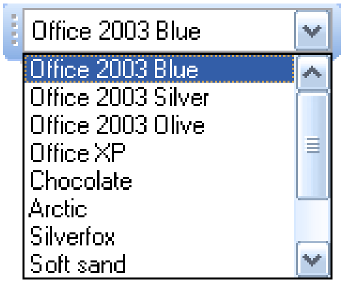
Debugger



Debugger Toolbar comes with following options:

Icon	Description
📄	Start Software Simulator.
📄	Run/Pause debugger.
📄	Stop debugger.
➔()	Step into.
↷()	Step over.
()➔	Step out.
➔I	Run to cursor.
■	Toggle breakpoint.
☰	Toggle breakpoints.
✕	Clear breakpoints.
👓	View watch window
⏱	View stopwatch window

Styles Toolbar



Styles toolbar allows you to easily customize your workspace.

Tools Toolbar

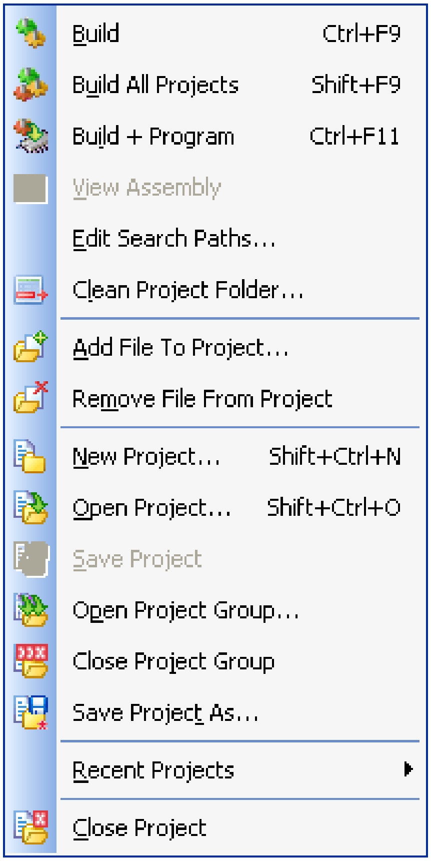
Tools Toolbar comes with following default options:

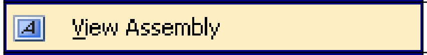
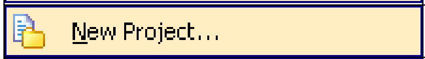
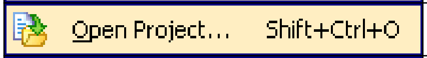
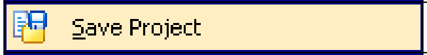
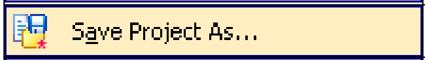
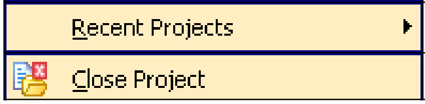
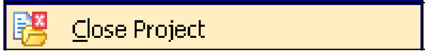
Icon	Description
🖥	Run USART Terminal
▦	EEPROM
A	ASCII Chart
✎	Seven segment decoder tool.

The Tools toolbar can easily be customized by adding new tools in Options(F12) window.

Related topics: Keyboard shortcuts, Integrated Tools, Debugger Windows

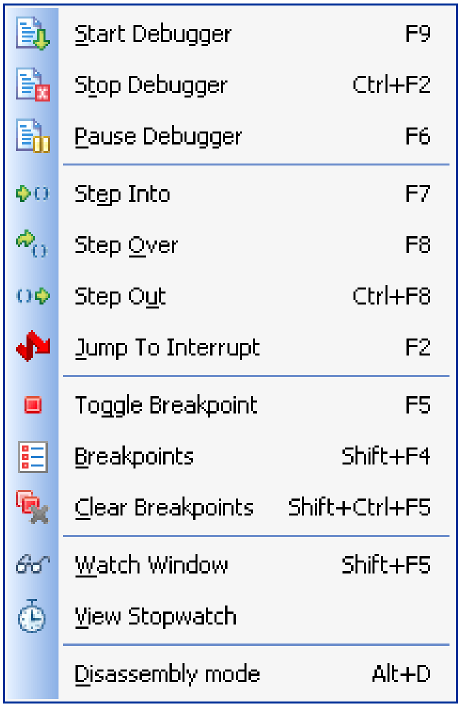
PROJECT MENU OPTIONS

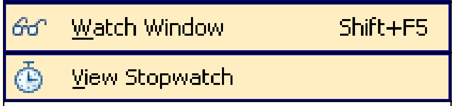
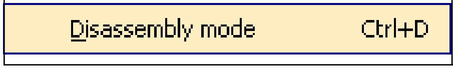


Project	Description
Build Ctrl+F9	Build active project.
Build All Shift+F9	Build all projects.
Build + Program Ctrl+F11	Build and program active project.
 View Assembly	View Assembly.
Edit Search Paths...	Edit search paths.
Clean Project Folder...	Clean Project Folder
Add File To Project...	Add file to project.
Remove File From Project	Remove file from project.
 New Project...	Open New Project Wizard
 Open Project... Shift+Ctrl+O	Open existing project.
 Save Project	Save current project.
Open Project Group...	Open project group.
Close Project Group	Close project group.
 Save Project As...	Save active project file with the different name.
 Recent Projects ▸	Open recently used project.
 Close Project	Close active project.

Related topics: Keyboard shortcuts, Project Toolbar, Creating New Project, Project Manager, Project Settings

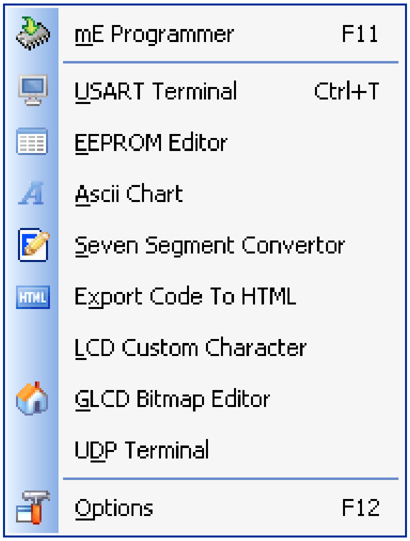
RUN MENU OPTIONS



Run	Description
Start Debugger F9	Start Software Simulator.
Stop Debugger Ctrl+F2	Stop debugger.
Pause Debugger F6	Pause Debugger.
Step Into F7	Step Into.
Step Over F8	Step Over.
Step Out Ctrl+F8	Step Out.
Jump To Interrupt F2	Jump to interrupt in current project.
Toggle Breakpoint F5	Toggle Breakpoint.
Show/Hide Breakpoints Shift+F4	Breakpoints.
Clear Breakpoints Shift+Ctrl+F5	Clear Breakpoints.
 Watch Window Shift+F5	Show/Hide Watch Window
View Stopwatch	Show/Hide Stopwatch Window
 Disassembly mode Ctrl+D	Toggle between Pascal source and disassembly.

Related topics: Keyboard shortcuts, Debug Toolbar

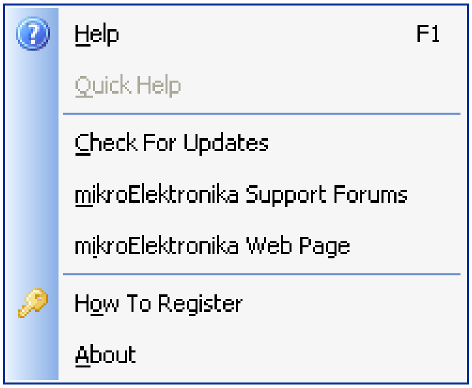
TOOLS MENU OPTIONS



Tools	Description
mE Programmer F11	Run mikroElektronika Programmer
USART Terminal Ctrl+T	Run USART Terminal
EEPROM Editor	Run EEPROM Editor
Ascii Chart	Run ASCII Chart
Seven Segment Convertor	Run 7 Segment Display Decoder
Export Code To HTML	Generate HTML code suitable for publishing source code on the web.
LCD Custom Character	Generate your own custom Lcd characters
GLCD Bitmap Editor	Generate bitmap pictures for Glcd
UDP Terminal	UDP communication terminal.
Options F12	Open Options window

Related topics: Keyboard shortcuts, Tools Toolbar

HELP MENU OPTION



Help	Description
Help F1	Open Help File.
Quick Help	Quick Help.
Check For Updates	Check if new compiler version is available.
mikroElektronika Support Forums	Open mikroElektronika Support Forums in a default browser.
mikroElektronika Web Page	Open mikroElektronika Web Page in a default browser.
How To Register	Information on how to register
About	Open About window.

Related Topics:Keyboard shortcuts

KEYBOARD SHORTCUTS

Below is a complete list of keyboard shortcuts available in mikroPascal PRO for AVR IDE. You can also view keyboard shortcuts in the Code Explorer window, tab Keyboard.

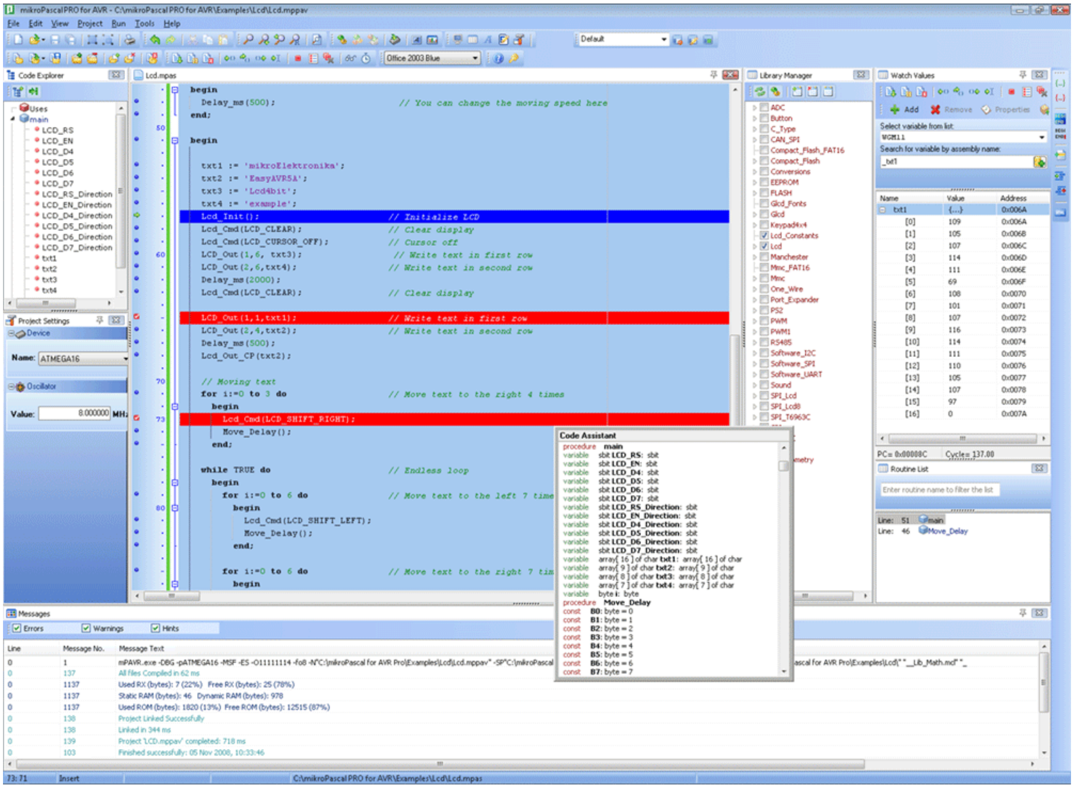
IDE Shortcuts	
F1	Help
Ctrl+N	New Unit
Ctrl+O	Open
Ctrl+Shift+O	Open Project
Ctrl+Shift+N	Open New Project
Ctrl+K	Close Project
Ctrl+F9	Compile
Shift+F9	Compile All
Ctrl+F11	Compile and Program
Shift+F4	Compile and Program
Ctrl+Shift+F5	Clear breakpoints
F11	Start AVRFlash Programmer
F12	Preferences
Basic Editor Shortcuts	
F3	Find, Find Next
Shift+F3	Find Previous
Alt+F3	Grep Search, Find in Files
Ctrl+A	Select All
Ctrl+C	Copy
Ctrl+F	Find
Ctrl+R	Replace
Ctrl+P	Print
Ctrl+S	Save unit
Ctrl+Shift+S	Save All
Ctrl+Shift+V	Paste

Ctrl+X	Cut
Ctrl+Y	Delete entire line
Ctrl+Z	Undo
Ctrl+Shift+Z	Redo
Advanced Editor Shortcuts	
Ctrl+Space	Code Assistant
Ctrl+Shift+Space	Parameters Assistant
Ctrl+D	Find declaration
Ctrl+E	Incremental Search
Ctrl+L	Routine List
Ctrl+G	Goto line
Ctrl+J	Insert Code Template
Ctrl+Shift+.	Comment Code
Ctrl+Shift+,	Uncomment Code
Ctrl+number	Goto bookmark
Ctrl+Shift+number	Set bookmark
Ctrl+Shift+I	Indent selection
Ctrl+Shift+U	Unindent selection
TAB	Indent selection
Shift+TAB	Unindent selection
Alt+Select	Select columns
Ctrl+Alt+Select	Select columns
Ctrl+Alt+L	Convert selection to lowercase
Ctrl+Alt+U	Convert selection to uppercase
Ctrl+Alt+T	Convert to Titlecase

Software Simulator Shortcuts	
F2	Jump To Interrupt
F4	Run to Cursor
F5	Toggle Breakpoint
F6	Run/Pause Debugger
F7	Step into
F8	Step over
F9	Debug
Ctrl+F2	Reset
Ctrl+F5	Add to Watch List
Ctrl+F8	Step out
Alt+D	Dissassembly view
Shift+F5	Open Watch Window

IDE OVERVIEW

The mikroPascal PRO for AVR is an user-friendly and intuitive environment:



```
  begin
    Delay_ms(500);                    // You can change the moving speed here
  end;

  begin

    txt1 := 'mikroElektronika';
    txt2 := 'EasyAVR5A';
    txt3 := 'Lcd4bit';
    txt4 := 'example';
    Lcd_Init();                     // Initialize LCD
    Lcd_Cmd(LCD_CLEAR);             // Clear display
    Lcd_Cmd(LCD_CURSOR_OFF);        // Cursor off
    LCD_Out(1,6, txt3);              // Write text in first row
    LCD_Out(2,6,txt4);              // Write text in second row
    Delay_ms(2000);
    Lcd_Cmd(LCD_CLEAR);             // Clear display

    LCD_Out(1,1,txt1);              // Write text in first row
    LCD_Out(2,4,txt2);              // Write text in second row
    Delay_ms(500);
    Lcd_Out_CP(txt2);

    // Moving text
    for i:=0 to 3 do                // Move text to the right 4 times
      begin
        Lcd_Cmd(LCD_SHIFT_RIGHT);
        Move_Delay();
      end;

    while TRUE do                   // Endless loop
      begin
        for i:=0 to 6 do            // Move text to the left 7 time
          begin
            Lcd_Cmd(LCD_SHIFT_LEFT);
            Move_Delay();
          end;

        for i:=0 to 6 do            // Move text to the right 7 tim
          begin
```

```
procedure  main
variable   sbit LCD_RS: sbit
variable   sbit LCD_EN: sbit
variable   sbit LCD_D4: sbit
variable   sbit LCD_D5: sbit
variable   sbit LCD_D6: sbit
variable   sbit LCD_D7: sbit
variable   sbit LCD_RS_Direction: sbit
variable   sbit LCD_EN_Direction: sbit
variable   sbit LCD_D4_Direction: sbit
variable   sbit LCD_D5_Direction: sbit
variable   sbit LCD_D6_Direction: sbit
variable   sbit LCD_D7_Direction: sbit
variable   array[ 16 ] of char txt1: array[ 16 ] of char
variable   array[ 9 ] of char txt2: array[ 9 ] of char
variable   array[ 8 ] of char txt3: array[ 8 ] of char
variable   array[ 7 ] of char txt4: array[ 7 ] of char
variable   byte i: byte
procedure  Move_Delay
const      B0: byte = 0
const      B1: byte = 1
const      B2: byte = 2
const      B3: byte = 3
const      B4: byte = 4
const      B5: byte = 5
const      B6: byte = 6
const      B7: byte = 7
```

Name	Value	Address
txt1	{...}	0x006A
[0]	109	0x006A
[1]	105	0x006B
[2]	107	0x006C
[3]	114	0x006D
[4]	111	0x006E
[5]	69	0x006F
[6]	108	0x0070
[7]	101	0x0071
[8]	107	0x0072
[9]	116	0x0073
[10]	114	0x0074
[11]	111	0x0075
[12]	110	0x0076
[13]	105	0x0077
[14]	107	0x0078
[15]	97	0x0079
[16]	0	0x007A

Line	Message No.	Message Text
0	1	mPAVR.exe -DBG -pATMEGA16 -MSF -ES -O11111114 -fo8 -N"C:\mikroPascal for AVR Pro\Examples\Lcd\Lcd.mppav" -SP"C:\mikroPascal ... ascal for AVR Pro\Examples\Lcd\" "__Lib_Math.mcl" "_
0	137	All files Compiled in 62 ms
0	1137	Used RX (bytes): 7 (22%) Free RX (bytes): 25 (78%)
0	1137	Static RAM (bytes): 46 Dynamic RAM (bytes): 978
0	1137	Used ROM (bytes): 1820 (13%) Free ROM (bytes): 12515 (87%)
0	138	Project Linked Successfully
0	138	Linked in 344 ms
0	139	Project 'LCD.mppav' completed: 718 ms
0	103	Finished successfully: 05 Nov 2008, 10:33:46

- The Code Editor features adjustable Syntax Highlighting, Code Folding, Code Assistant, Parameters Assistant, Spell Checker, Auto Correct for common typos and Code Templates (Auto Complete).
- The Code Explorer (with Keyboard shortcut browser and Quick Help browser) is at your disposal for easier project management.
- The Project Manager alows multiple project management
- General project settings can be made in the Project Settings window
- Library manager enables simple handling libraries being used in a project
- The Error Window displays all errors detected during compiling and linking.
- The source-level Software Simulator lets you debug executable logic step-by-step by watching the program flow.
- The New Project Wizard is a fast, reliable, and easy way to create a project.
- Help files are syntax and context sensitive.

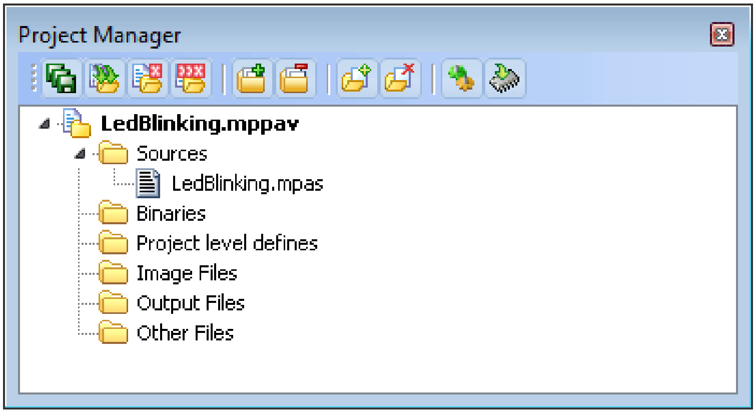
- Like in any modern Windows application, you may customize the layout of mikroPascal PRO for AVR to suit your needs best.
- Spell checker underlines identifiers which are unknown to the project. In this way it helps the programmer to spot potential problems early, much before the project is compiled.
- Spell checker can be disabled by choosing the option in the Preferences dialog (F12).

CUSTOMIZING IDE LAYOUT

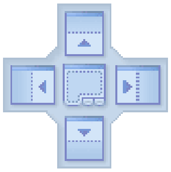
Docking Windows

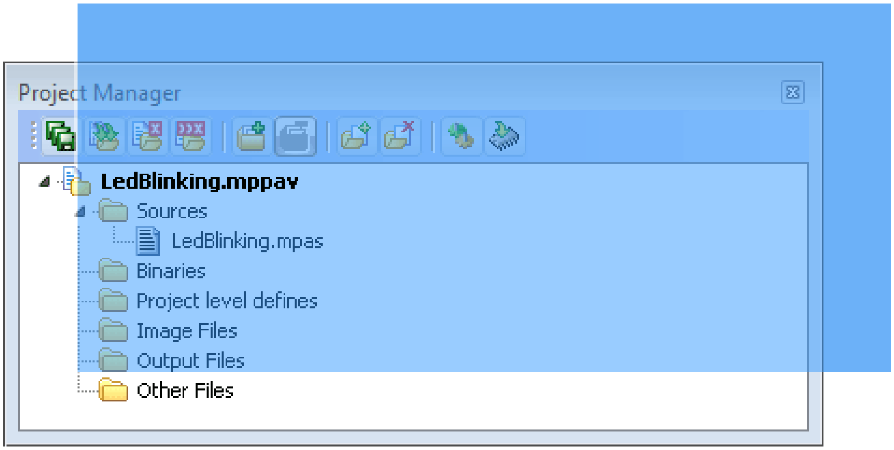
You can increase the viewing and editing space for code, depending on how you arrange the windows in the IDE.

Step 1: Click the window you want to dock, to give it focus.

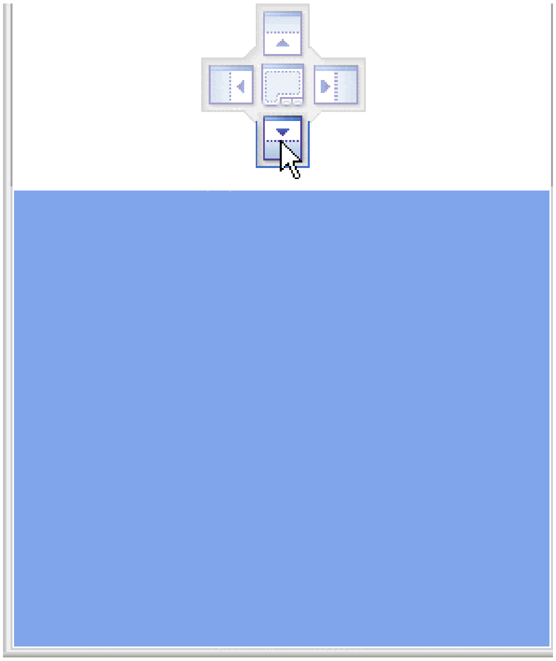


Step 2: Drag the tool window from its current location. A guide diamond appears. The four arrows of the diamond point towards the four edges of the IDE.





Step 3: Move the pointer over the corresponding portion of the guide diamond. An outline of the window appears in the designated area.

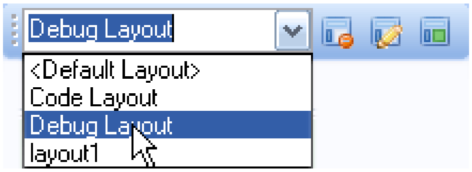


Step 4: To dock the window in the position indicated, release the mouse button.

Tip: To move a dockable window without snapping it into place, press CTRL while dragging it.

Saving Layout

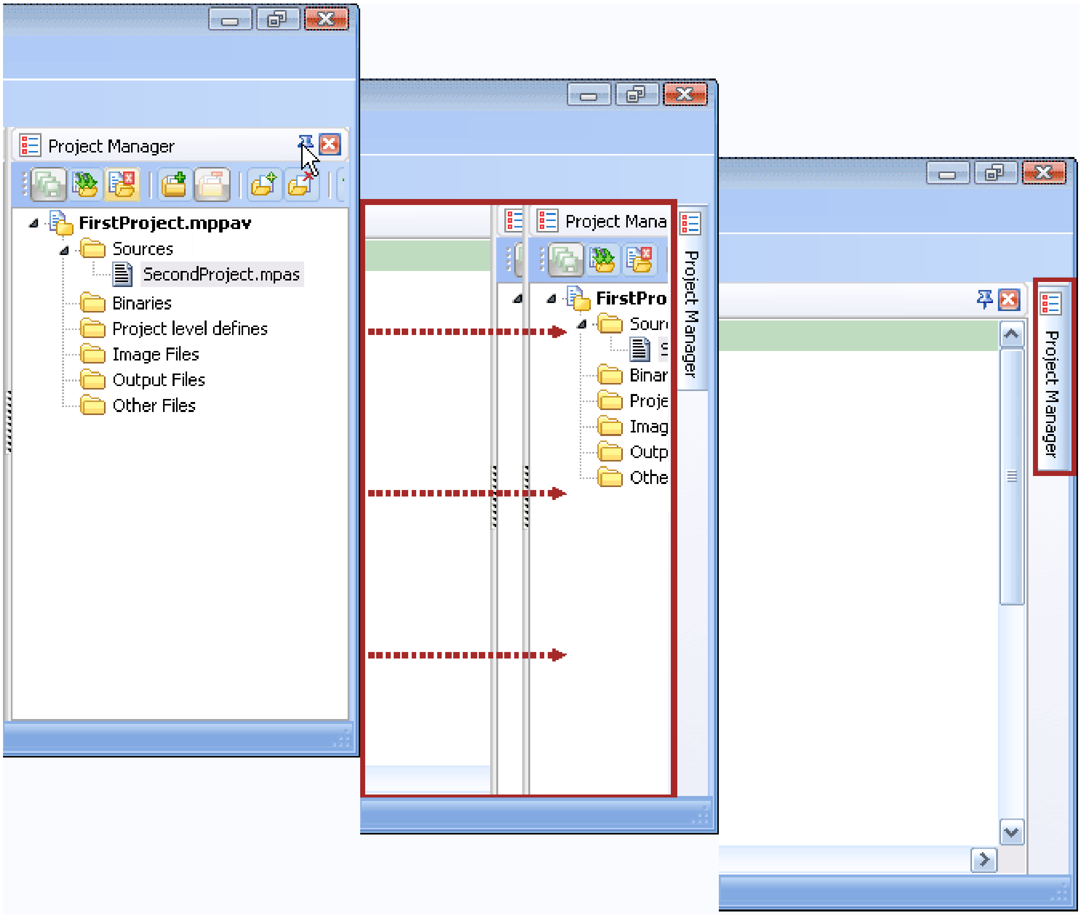
Once you have a window layout that you like, you can save the layout by typing the name for the layout and pressing the Save Layout Icon .
To set the layout select the desired layout from the layout drop-down list and click the Set Layout Icon .
To remove the layout from the drop-down list, select the desired layout from the list and click the Delete Layout Icon .



Auto Hide

Auto Hide enables you to see more of your code at one time by minimizing tool windows along the edges of the IDE when not in use.

- Click the window you want to keep visible to give it focus.
- Click the Pushpin Icon 📌 on the title bar of the window.



When an auto-hidden window loses focus, it automatically slides back to its tab on the edge of the IDE. While a window is auto-hidden, its name and icon are visible on a tab at the edge of the IDE. To display an auto-hidden window, move your pointer over the tab. The window slides back into view and is ready for use.

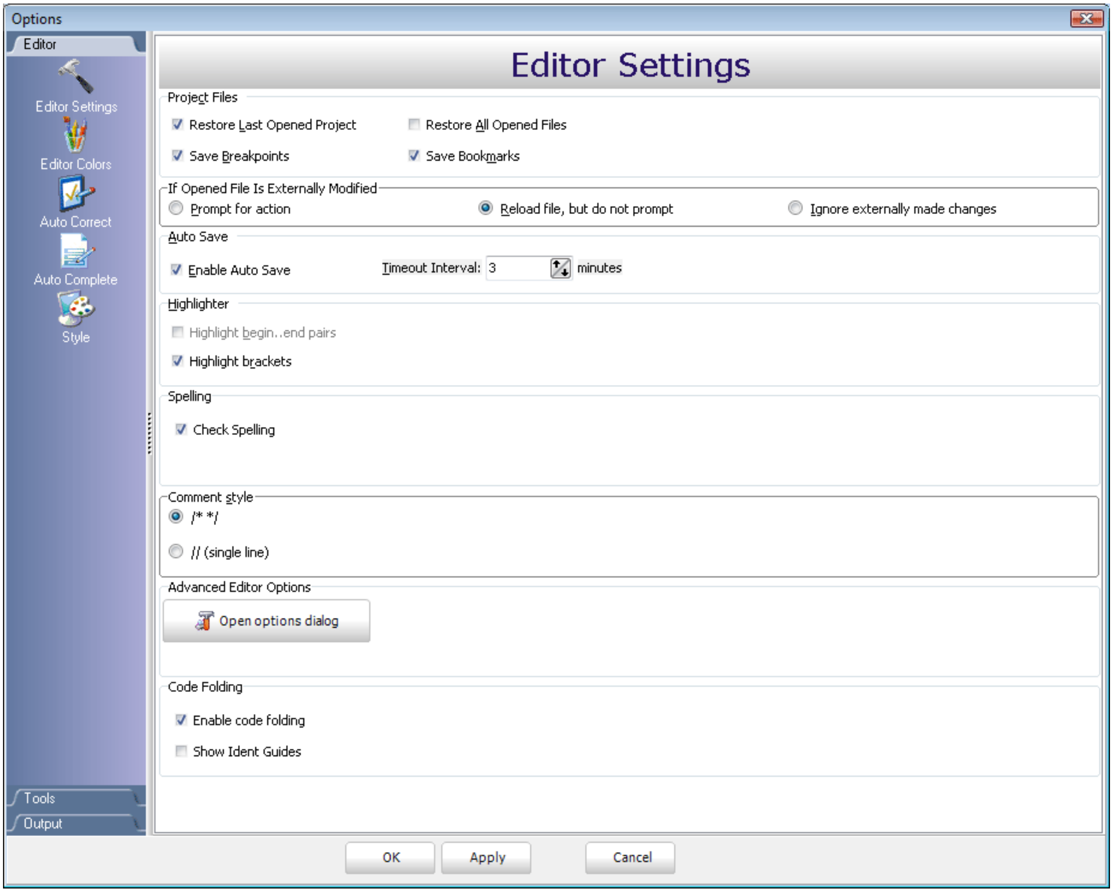
ADVANCED CODE EDITOR

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

Advanced Editor Features

- Adjustable Syntax Highlighting
- Code Assistant
- Code Folding
- Parameter Assistant
- Code Templates (Auto Complete)
- Auto Correct for common typos
- Spell Checker
- Bookmarks and Goto Line
- Comment / Uncomment

You can configure the Syntax Highlighting, Code Templates and Auto Correct from the Editor Settings dialog. To access the Settings, click **Tools › Options** from the drop-down menu, click the Show Options Icon or press F12 key.



Editor Settings

Project Files

☑ Restore Last Opened Project □ Restore All Opened Files

☑ Save Breakpoints ☑ Save Bookmarks

If Opened File Is Externally Modified

○ Prompt for action ● Reload file, but do not prompt ○ Ignore externally made changes

Auto Save

☑ Enable Auto Save Timeout Interval: 3 minutes

Highlighter

□ Highlight begin..end pairs

☑ Highlight brackets

Spelling

☑ Check Spelling

Comment style

● /* */

○ // (single line)

Advanced Editor Options

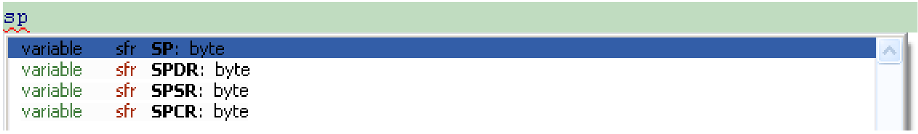
Code Folding

☑ Enable code folding

□ Show Ident Guides

Code Assistant

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

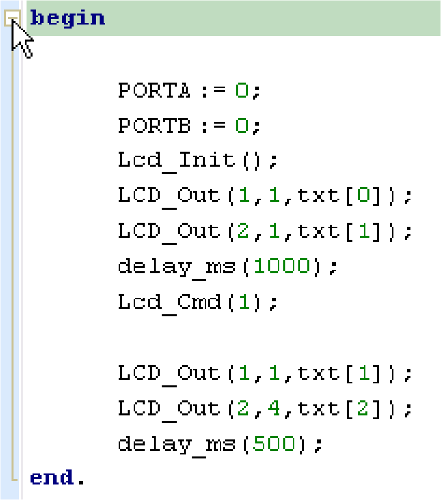


```
sp
variable   sfr   SP: byte
variable   sfr   SPDR: byte
variable   sfr   SPSR: byte
variable   sfr   SPCR: byte
```

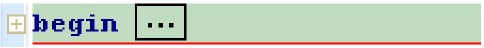
Code Folding

Code folding is IDE feature which allows users to selectively hide and display sections of a source file. In this way it is easier to manage large regions of code within one window, while still viewing only those subsections of the code that are relevant during a particular editing session.

While typing, the code folding symbols (⊞ and ⊟) appear automatically. Use the folding symbols to hide/unhide the code subsections.

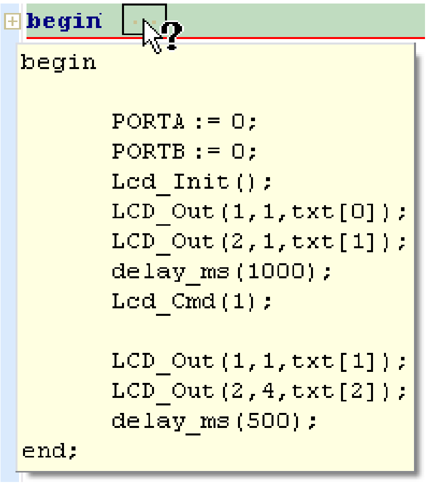


```
begin

        PORTA := 0;
        PORTB := 0;
        Lcd_Init();
        LCD_Out(1,1,txt[0]);
        LCD_Out(2,1,txt[1]);
        delay_ms(1000);
        Lcd_Cmd(1);

        LCD_Out(1,1,txt[1]);
        LCD_Out(2,4,txt[2]);
        delay_ms(500);
end.
```



```
begin ...
```

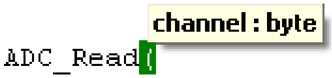
If you place a mouse cursor over the tooltip box, the collapsed text will be shown in a tooltip style box.



```
begin
begin

        PORTA := 0;
        PORTB := 0;
        Lcd_Init();
        LCD_Out(1,1,txt[0]);
        LCD_Out(2,1,txt[1]);
        delay_ms(1000);
        Lcd_Cmd(1);

        LCD_Out(1,1,txt[1]);
        LCD_Out(2,4,txt[2]);
        delay_ms(500);
end;
```

Parameter Assistant

The Parameter Assistant will be automatically invoked when you open parenthesis "(" or press Shift+Ctrl+Space. If the name of a valid function precedes the parenthesis, then the expected parameters will be displayed in a floating panel. As you type the actual parameter, the next expected parameter will become bold.



```
                channel : byte
ADC_Read(
```

Code Templates (Auto Complete)

You can insert the Code Template by typing the name of the template (for instance, `whiles`), then press Ctrl+J and the Code Editor will automatically generate a code.

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

Autocomplete macros can retreive system and project information:

- `%DATE%` - current system date
- `%TIME%` - current system time
- `%DEVICE%` - device(MCU) name as specified in project settings
- `%DEVICE_CLOCK%` - clock as specified in project settings
- `%COMPILER%` - current compiler version

These macros can be used in template code, see template ptemplate provided with mikroPascal PRO for AVR installation.

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 Show Options Icon and then select the Auto Correct Tab. You can also add your own preferences to the list.

Also, the Code Editor has a feature to comment or uncomment the selected code by simple click of a mouse, using the Comment Icon {..} and Uncomment Icon {..} from the Code Toolbar.

Spell Checker

The Spell Checker underlines unknown objects in the code, so they can be easily noticed and corrected before compiling your project.

Select **Tools › Options** from the drop-down menu, or click the Show Options Icon and then select the Spell Checker Tab.

Bookmarks

Bookmarks make navigation through a large code easier. To set a bookmark, use Ctrl+Shift+number. To jump to a bookmark, use Ctrl+number.

Goto Line

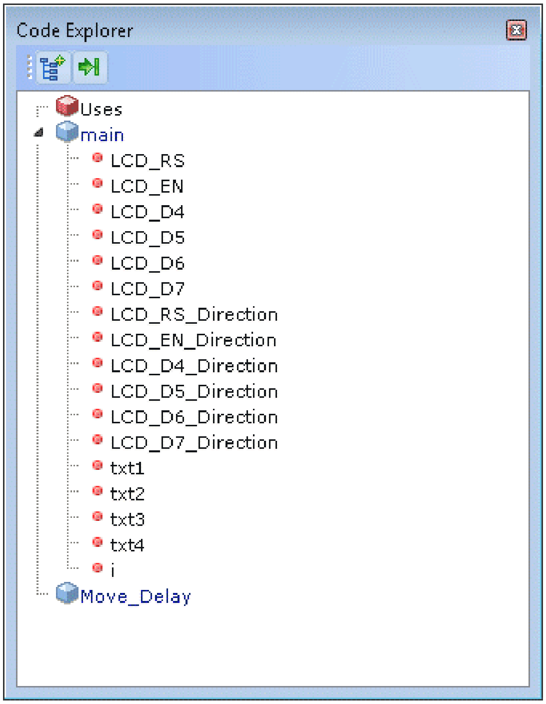
The Goto Line option makes navigation through a large code easier. Use the shortcut Ctrl+G to activate this option.

Comment / Uncomment

Also, the Code Editor has a feature to comment or uncomment the selected code by simple click of a mouse, using the Comment Icon {..} and Uncomment Icon {..} from the Code Toolbar.

CODE EXPLORER

The Code Explorer gives clear view of each item declared inside the source code. You can jump to a declaration of any item by right clicking it. Also, besides the list of defined and declared objects, code explorer displays message about first error and it's location in code.



Code Explorer

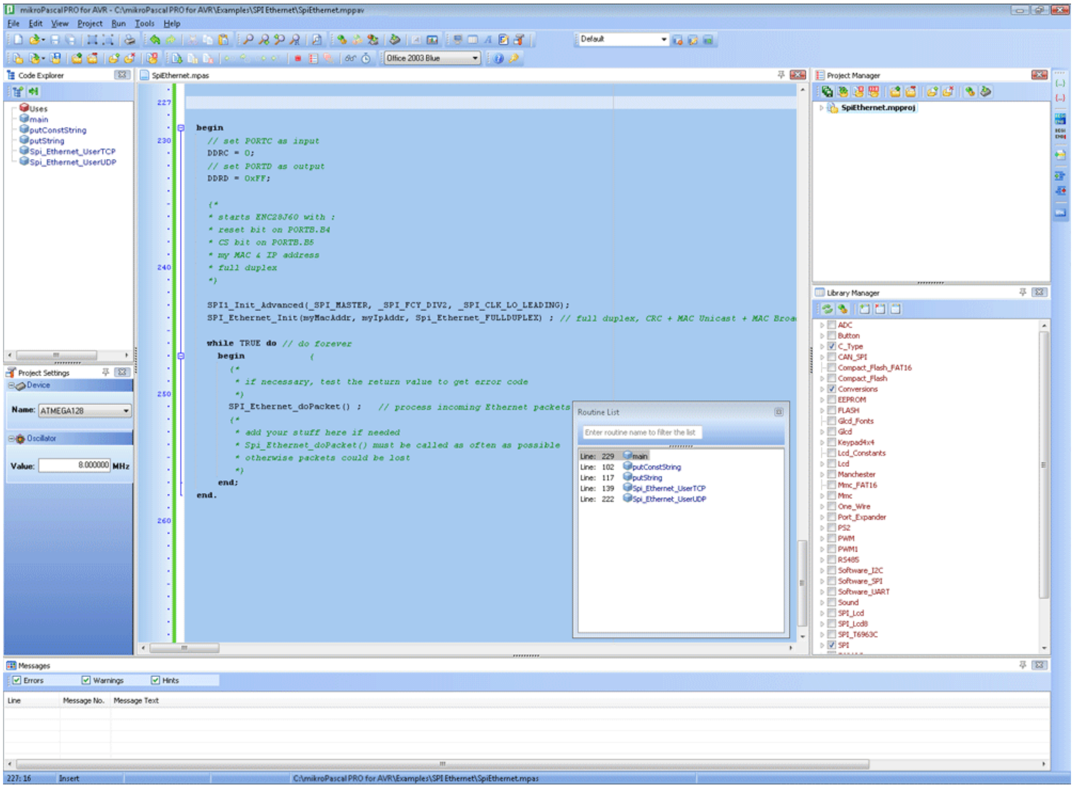
- Uses
- main
  - LCD_RS
  - LCD_EN
  - LCD_D4
  - LCD_D5
  - LCD_D6
  - LCD_D7
  - LCD_RS_Direction
  - LCD_EN_Direction
  - LCD_D4_Direction
  - LCD_D5_Direction
  - LCD_D6_Direction
  - LCD_D7_Direction
  - txt1
  - txt2
  - txt3
  - txt4
  - i
- Move_Delay

Icon	Description
⊞	Expand/Collapse all nodes in tree.
⇥	Locate declaration in code.

ROUTINE LIST

Routine list diplays list of routines, and enables filtering routines by name. Routine list window can be accessed by pressing Ctrl+L.

You can jump to a desired routine by double clicking on it.



Name: ATMEGA128

Value: 8.000000 MHz

```
begin
  // set PORTC as input
  DDRC = 0;
  // set PORTD as output
  DDRD = 0xFF;

  {*
  * starts ENC28J60 with :
  * reset bit on PORTB.B4
  * CS bit on PORTB.B5
  * my MAC & IP address
  * full duplex
  *}

  SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV2, _SPI_CLK_LO_LEADING);
  SPI_Ethernet_Init(myMacAddr, myIpAddr, Spi_Ethernet_FULLDUPLEX) ; // full duplex, CRC + MAC Unicast + MAC Broa

  while TRUE do // do forever
    begin            {
      {*
       * if necessary, test the return value to get error code
       *}
      SPI_Ethernet_doPacket() ;   // process incoming Ethernet packets
      {*
       * add your stuff here if needed
       * Spi_Ethernet_doPacket() must be called as often as possible
       * otherwise packets could be lost
       *}
    end;
end.
```

Routine List

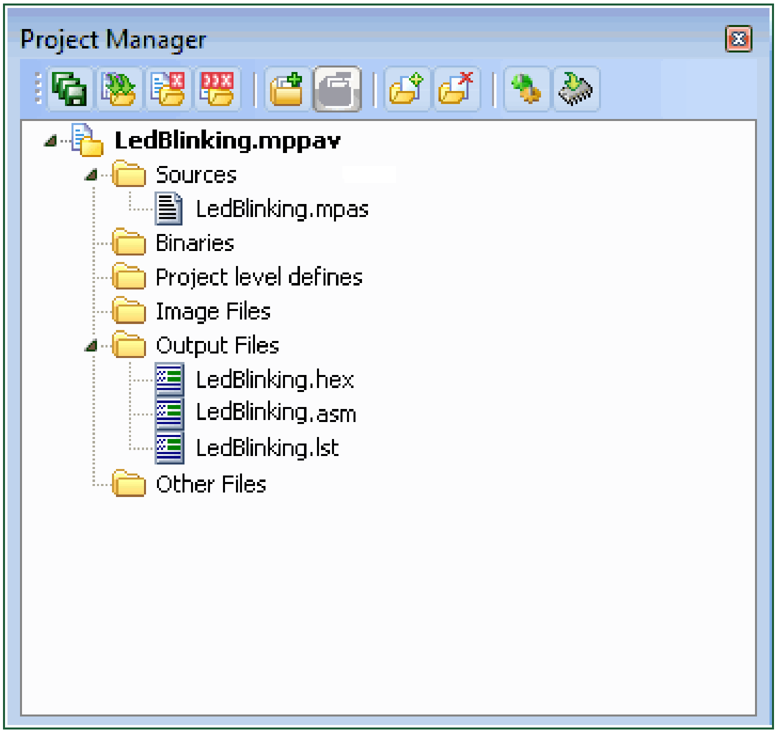
Enter routine name to filter the list

```
Line:  229  main
Line:  102  putConstString
Line:  117  putString
Line:  139  Spi_Ethernet_UserTCP
Line:  222  Spi_Ethernet_UserUDP
```

227: 16   Insert   C:\mikroPascal PRO for AVR\Examples\SPI Ethernet\SpiEthernet.mpas

PROJECT MANAGER

Project Manager is IDE feature which allows users to manage multiple projects. Several projects which together make project group may be open at the same time. Only one of them may be active at the moment.
Setting project in **active** mode is performed by **double click** on the desired project in the Project Manager.



Following options are available in the Project Manager:

Following options are available in the Project Manager:

Icon	Description
	Save project Group.
	Open project group.
	Close the active project.
	Close project group.
	Add project to the project group.
	Remove project from the project group.
	Add file to the active project.
	Remove selected file from the project.
	Build the active project.
	Run mikroElektronika's Flash programmer.

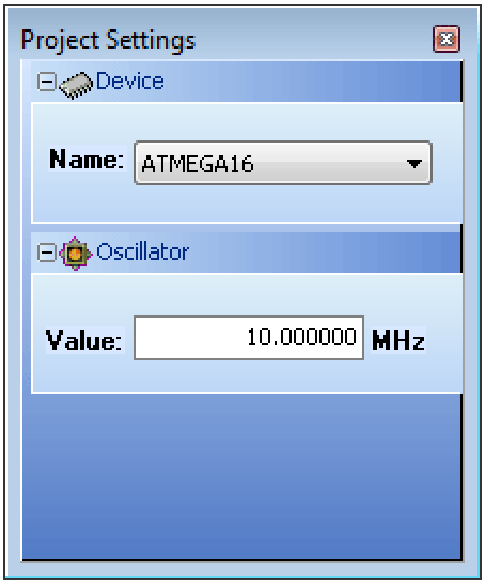
For details about adding and removing files from project see Add/Remove Files from Project.

Related topics: Project Settings, Project Menu Options, File Menu Options, Project Toolbar, Build Toolbar, Add/Remove Files from Project

PROJECT SETTINGS WINDOW

Following options are available in the Project Settings Window:

- Device - select the appropriate device from the device drop-down list.
- Oscillator - enter the oscillator frequency value.



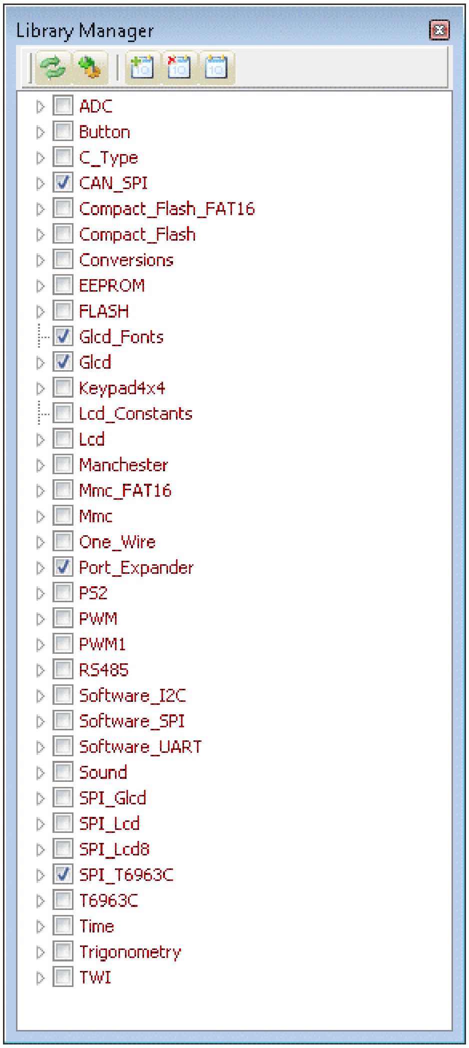
Name: ATMEGA16

Value: 10.000000 MHz

LIBRARY MANAGER

Library Manager enables simple handling libraries being used in a project. Library Manager window lists all libraries (extencion `.mcl`) which are instantly stored in the compiler Uses folder. The desirable library is added to the project by selecting check box next to the library name.

In order to have all library functions accessible, simply press the button **Check All** and all libraries will be selected. In case none library is needed in a project, press the button **Clear All** and all libraries will be cleared from the project. Only the selected libraries will be linked.



Library Manager

- ☐ ADC
- ☐ Button
- ☐ C_Type
- ☑ CAN_SPI
- ☐ Compact_Flash_FAT16
- ☐ Compact_Flash
- ☐ Conversions
- ☐ EEPROM
- ☐ FLASH
- ☑ Glcd_Fonts
- ☑ Glcd
- ☐ Keypad4x4
- ☐ Lcd_Constants
- ☐ Lcd
- ☐ Manchester
- ☐ Mmc_FAT16
- ☐ Mmc
- ☐ One_Wire
- ☑ Port_Expander
- ☐ PS2
- ☐ PWM
- ☐ PWM1
- ☐ RS485
- ☐ Software_I2C
- ☐ Software_SPI
- ☐ Software_UART
- ☐ Sound
- ☐ SPI_Glcd
- ☐ SPI_Lcd
- ☐ SPI_Lcd8
- ☑ SPI_T6963C
- ☐ T6963C
- ☐ Time
- ☐ Trigonometry
- ☐ TWI

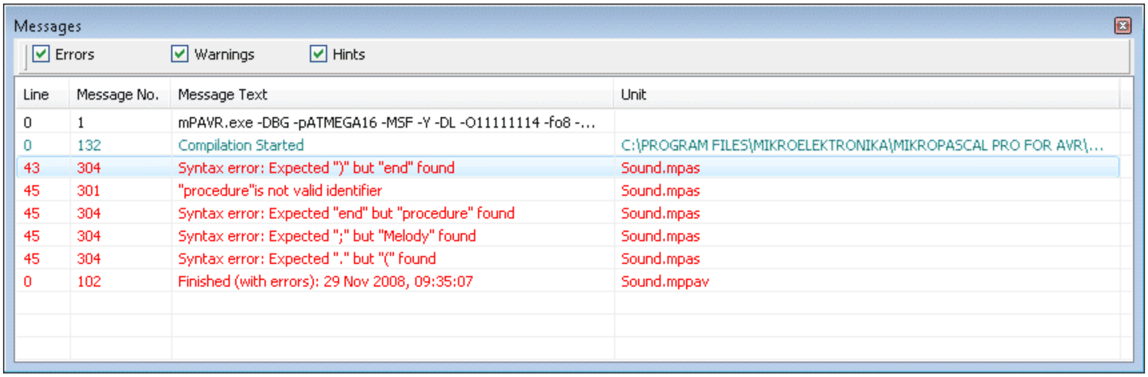
Icon	Description
	Refresh Library by scanning files in "Uses" folder.Useful when new libraries are added by copying files to "Uses" folder.
	Rebuild all available libraries. Useful when library sources are available and need refreshing.
	Include all available libraries in current project.
	No libraries from the list will be included in current project.
	Restore library to the state just before last project saving.

Related topics: mikroPascal PRO for AVR Libraries, Creating New Library

ERROR WINDOW

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

The Error Window is located under message tab, and displays location and type of errors the compiler has encountered. The compiler also reports warnings, but these do not affect the output; only errors can interefere with the generation of hex.



Messages

☑ Errors ☑ Warnings ☑ Hints

Line	Message No.	Message Text	Unit
0	1	mPAVR.exe -DBG -pATMEGA16 -MSF -Y -DL -O11111114 -fo8 -...	
0	132	Compilation Started	C:\PROGRAM FILES\MIKROELEKTRONIKA\MIKROPASCAL PRO FOR AVR\...
43	304	Syntax error: Expected ")" but "end" found	Sound.mpas
45	301	"procedure"is not valid identifier	Sound.mpas
45	304	Syntax error: Expected "end" but "procedure" found	Sound.mpas
45	304	Syntax error: Expected ";" but "Melody" found	Sound.mpas
45	304	Syntax error: Expected "." but "(" found	Sound.mpas
0	102	Finished (with errors): 29 Nov 2008, 09:35:07	Sound.mppav

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

Related topics: Error Messages

STATISTICS

After successful compilation, you can review statistics of your code. Click the Statistics Icon 📊 .

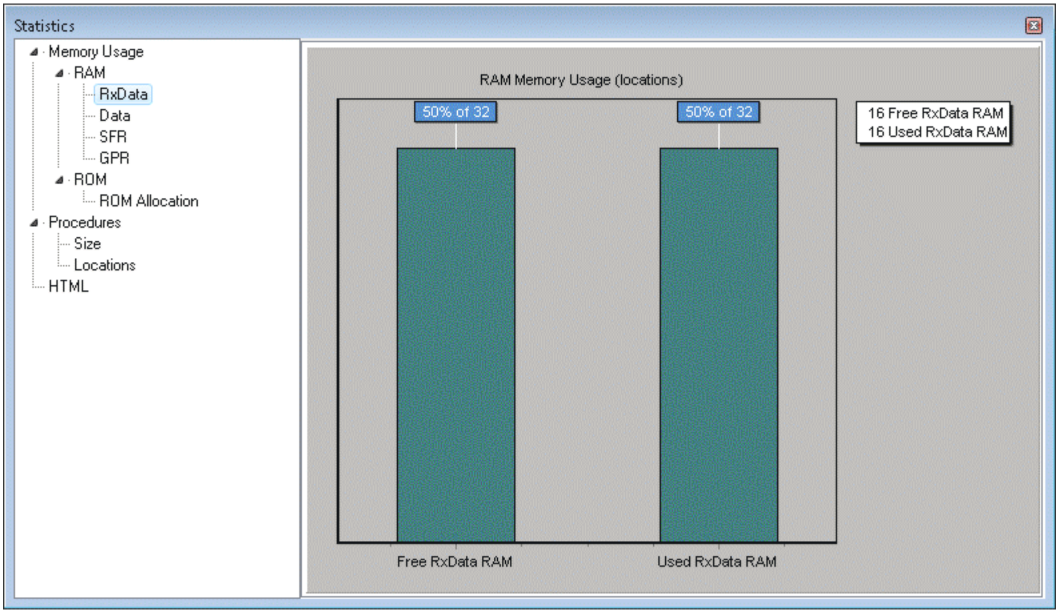
Memory Usage Windows

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

RAM MEMORY

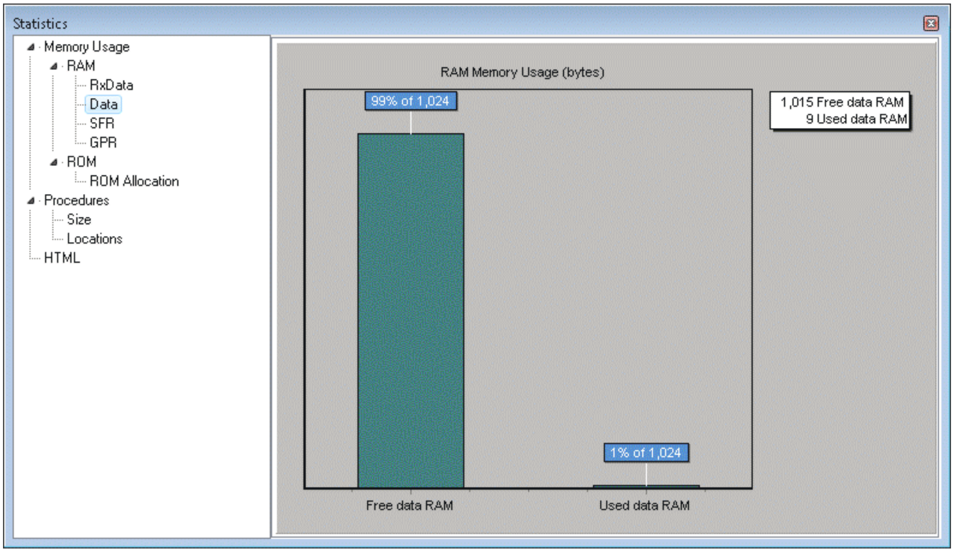
Rx Memory Space

Displays Rx memory space usage in form of histogram.



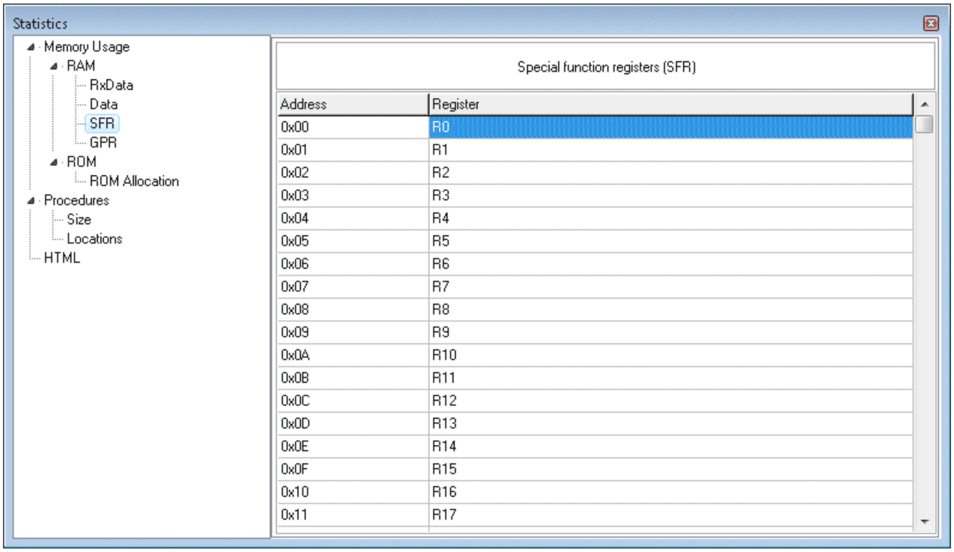
Data Memory Space

Displays Data memory space usage in form of histogram.



Special Function Registers

Summarizes all Special Function Registers and their addresses.

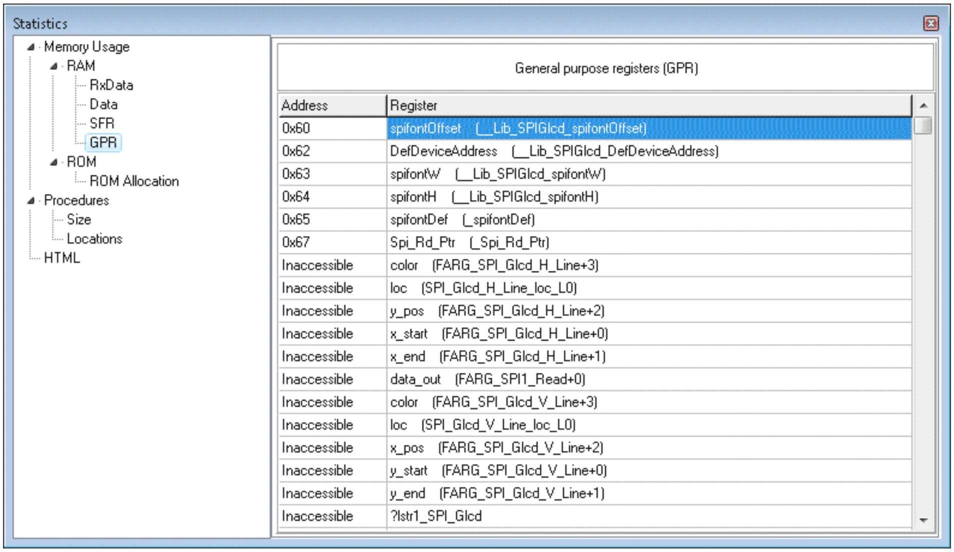


Special function registers (SFR)

Address	Register
0x00	R0
0x01	R1
0x02	R2
0x03	R3
0x04	R4
0x05	R5
0x06	R6
0x07	R7
0x08	R8
0x09	R9
0x0A	R10
0x0B	R11
0x0C	R12
0x0D	R13
0x0E	R14
0x0F	R15
0x10	R16
0x11	R17

General Purpose Registers

Summarizes all General Purpose Registers and their addresses. Also displays symbolic names of variables and their addresses.



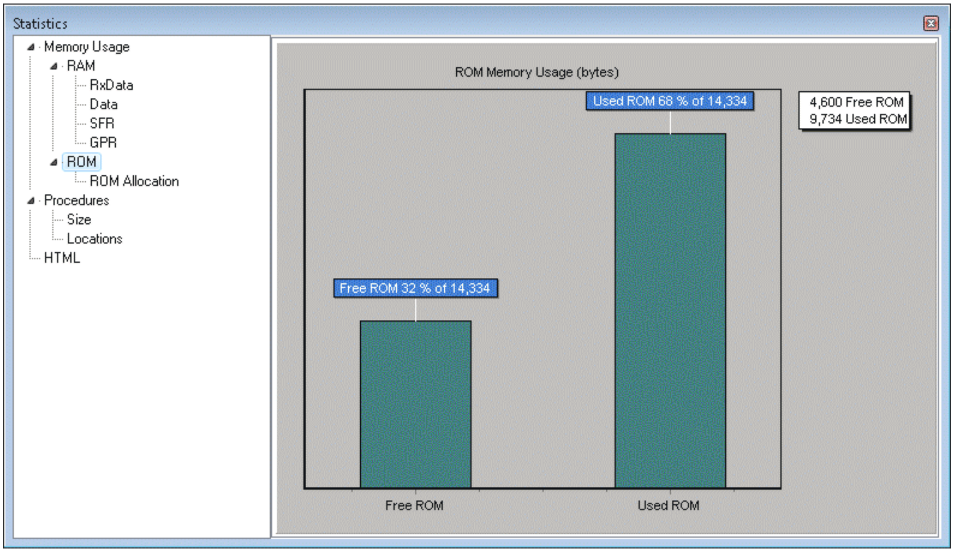
General purpose registers (GPR)

Address	Register
0x60	spifontOffset (__Lib_SPIGlcd_spifontOffset)
0x62	DefDeviceAddress (__Lib_SPIGlcd_DefDeviceAddress)
0x63	spifontW (__Lib_SPIGlcd_spifontW)
0x64	spifontH (__Lib_SPIGlcd_spifontH)
0x65	spifontDef (_spifontDef)
0x67	Spi_Rd_Ptr (_Spi_Rd_Ptr)
Inaccessible	color (FARG_SPI_Glcd_H_Line+3)
Inaccessible	loc (SPI_Glcd_H_Line_loc_L0)
Inaccessible	y_pos (FARG_SPI_Glcd_H_Line+2)
Inaccessible	x_start (FARG_SPI_Glcd_H_Line+0)
Inaccessible	x_end (FARG_SPI_Glcd_H_Line+1)
Inaccessible	data_out (FARG_SPI1_Read+0)
Inaccessible	color (FARG_SPI_Glcd_V_Line+3)
Inaccessible	loc (SPI_Glcd_V_Line_loc_L0)
Inaccessible	x_pos (FARG_SPI_Glcd_V_Line+2)
Inaccessible	y_start (FARG_SPI_Glcd_V_Line+0)
Inaccessible	y_end (FARG_SPI_Glcd_V_Line+1)
Inaccessible	?lstr1_SPI_Glcd

ROM MEMORY

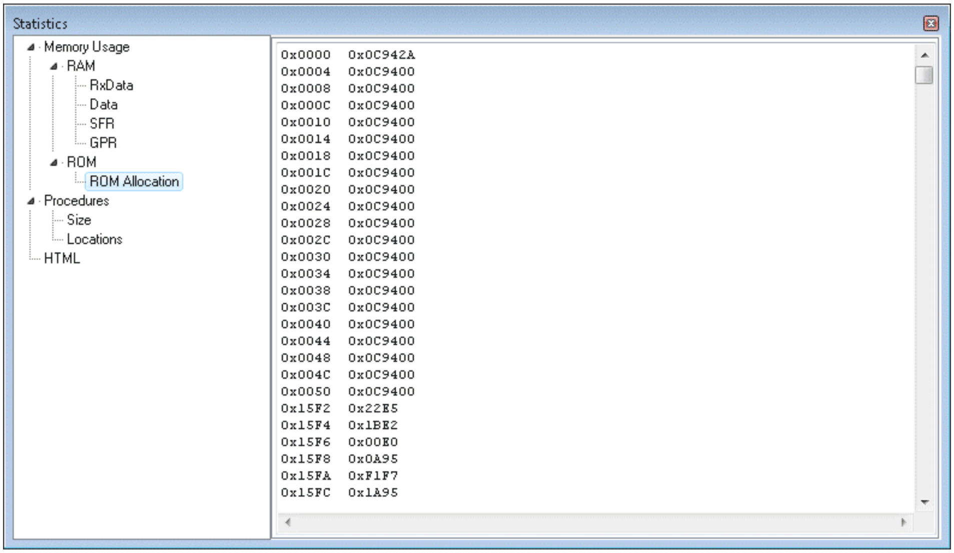
ROM Memory Usage

Displays ROM memory usage in form of histogram.



ROM Memory Allocation

Displays ROM memory allocation.



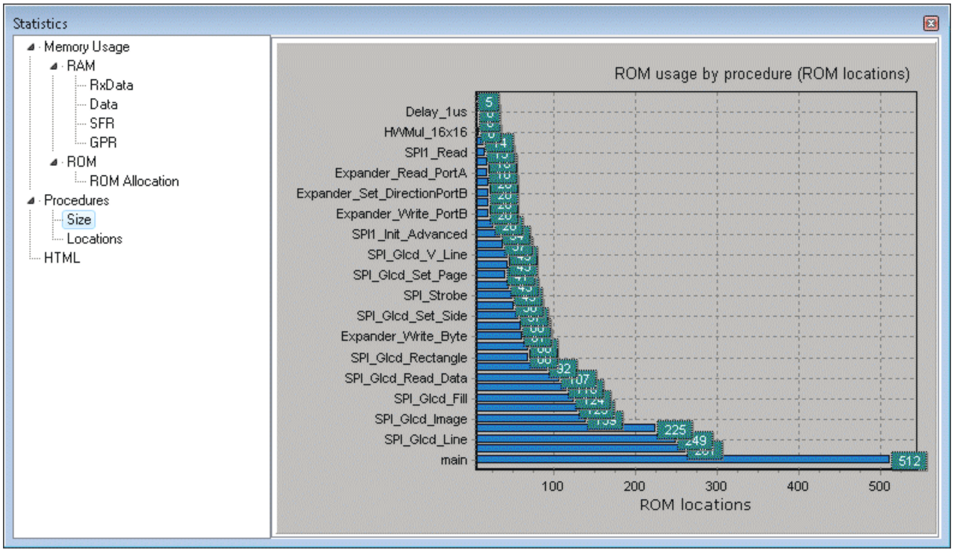
```
0x0000   0x0C942A
0x0004   0x0C9400
0x0008   0x0C9400
0x000C   0x0C9400
0x0010   0x0C9400
0x0014   0x0C9400
0x0018   0x0C9400
0x001C   0x0C9400
0x0020   0x0C9400
0x0024   0x0C9400
0x0028   0x0C9400
0x002C   0x0C9400
0x0030   0x0C9400
0x0034   0x0C9400
0x0038   0x0C9400
0x003C   0x0C9400
0x0040   0x0C9400
0x0044   0x0C9400
0x0048   0x0C9400
0x004C   0x0C9400
0x0050   0x0C9400
0x15F2   0x22E5
0x15F4   0x1BE2
0x15F6   0x00E0
0x15F8   0x0A95
0x15FA   0xF1F7
0x15FC   0x1A95
```

Procedures Windows

Provides overview procedures locations and sizes.

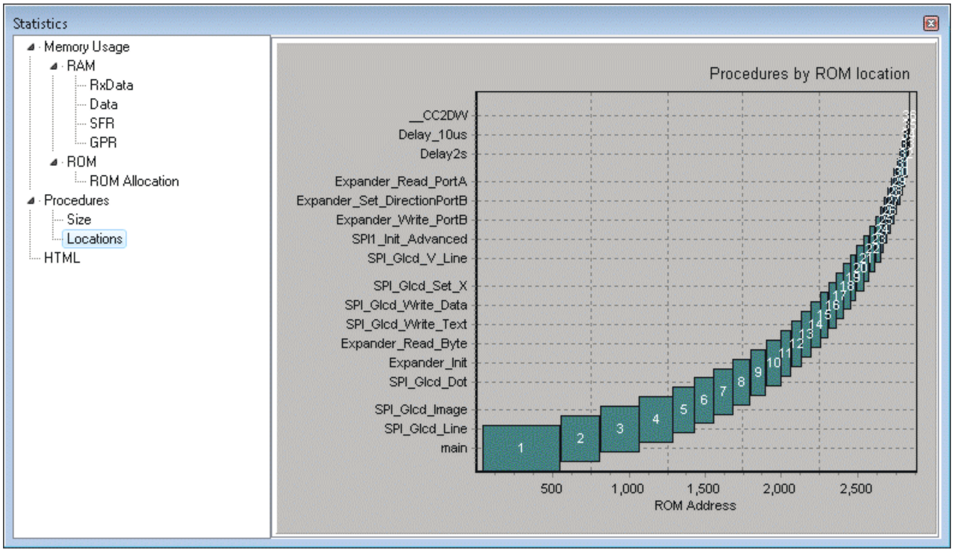
Procedures Size Window

Displays size of each procedure.



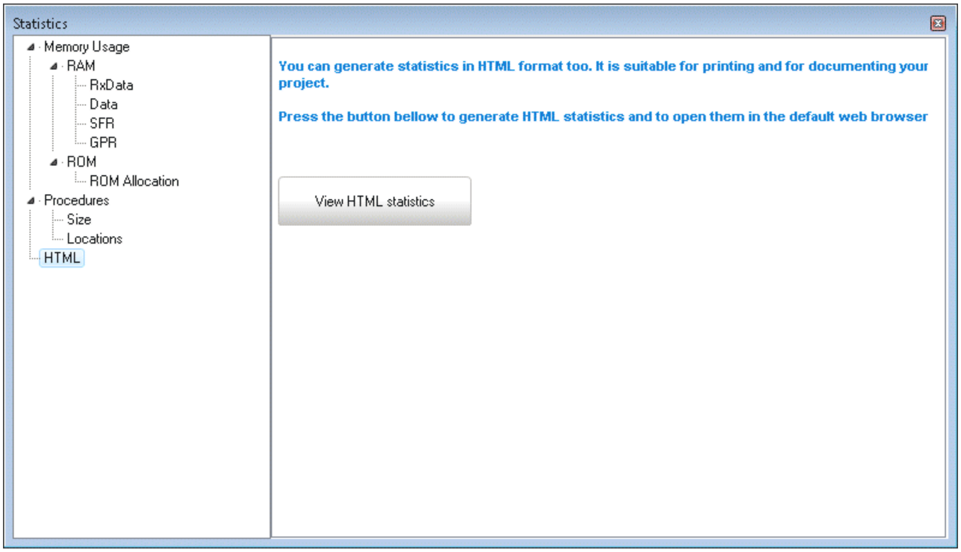
Procedures Locations Window

Displays how functions are distributed in microcontroller's memory.



HTML Window

Display statistics in default web browser.



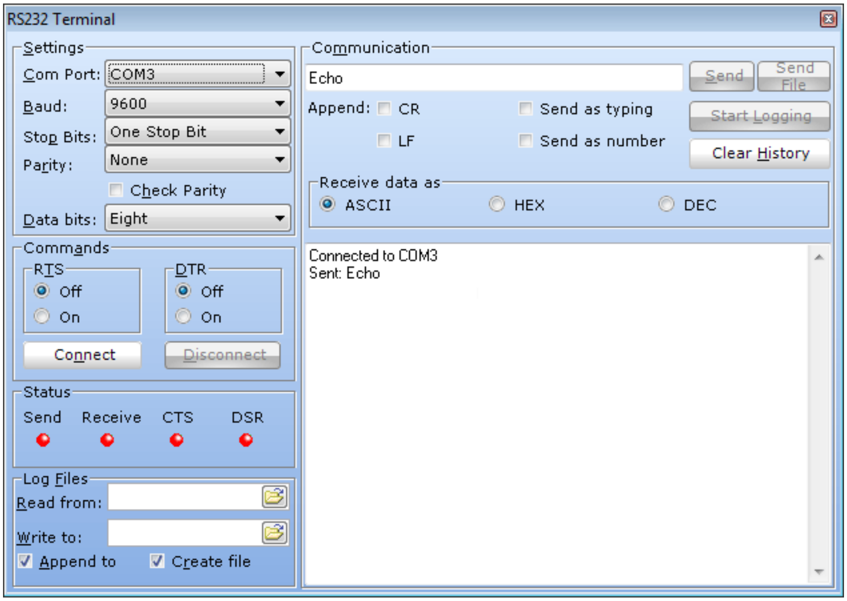
You can generate statistics in HTML format too. It is suitable for printing and for documenting your project.

Press the button bellow to generate HTML statistics and to open them in the default web browser

INTEGRATED TOOLS

USART Terminal

The mikroPascal PRO for AVR includes the USART communication terminal for RS232 communication. You can launch it from the drop-down menu **Tools › USART Terminal** or by clicking the USART Terminal Icon from Tools toolbar.



RS232 Terminal

Settings

Com Port: COM3

Baud: 9600

Stop Bits: One Stop Bit

Parity: None

□ Check Parity

Data bits: Eight

Commands

RTS: ● Off ○ On

DTR: ● Off ○ On

Log Files

Read from:

Write to:

☑ Append to ☑ Create file

Communication

Echo

Append: □ CR □ LF

□ Send as typing

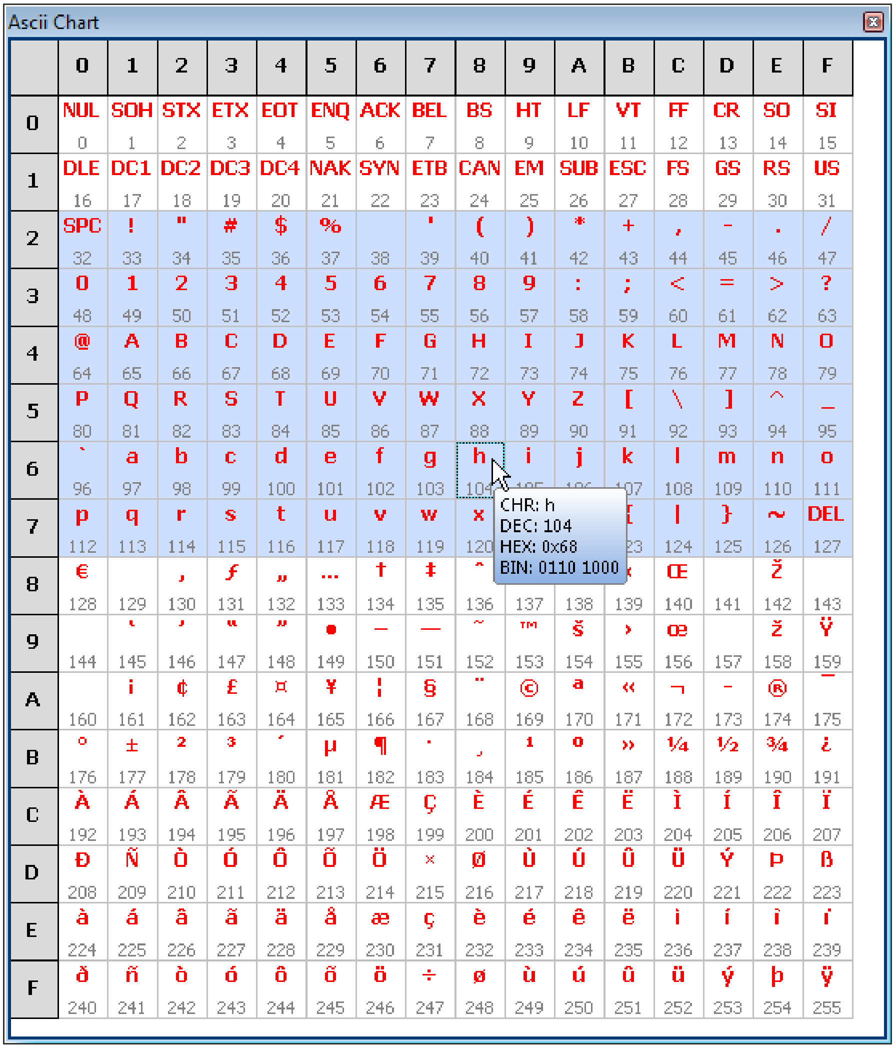
□ Send as number

Receive data as: ● ASCII ○ HEX ○ DEC

```
Connected to COM3
Sent: Echo
```

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** or by clicking the View ASCII Chart Icon from Tools toolbar.



Ascii Chart

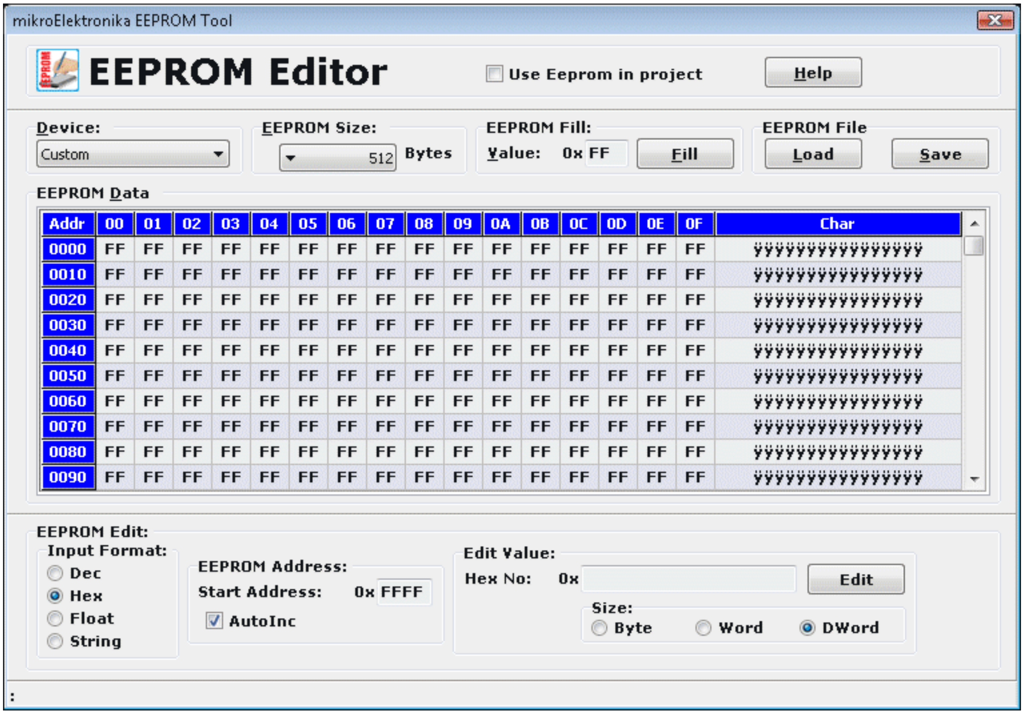
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
2	SPC 32	! 33	" 34	# 35	$ 36	% 37	38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120			{ 123	| 124	} 125	~ 126	DEL 127
8	€ 128	129	‚ 130	ƒ 131	„ 132	… 133	† 134	‡ 135	ˆ 136	137	138	‹ 139	Œ 140	141	Ž 142	143
9	144	‘ 145	’ 146	“ 147	” 148	• 149	– 150	— 151	˜ 152	™ 153	š 154	› 155	œ 156	157	ž 158	Ÿ 159
A	160	¡ 161	¢ 162	£ 163	¤ 164	¥ 165	¦ 166	§ 167	¨ 168	© 169	ª 170	« 171	¬ 172	- 173	® 174	¯ 175
B	° 176	± 177	² 178	³ 179	´ 180	µ 181	¶ 182	· 183	¸ 184	¹ 185	º 186	» 187	¼ 188	½ 189	¾ 190	¿ 191
C	À 192	Á 193	Â 194	Ã 195	Ä 196	Å 197	Æ 198	Ç 199	È 200	É 201	Ê 202	Ë 203	Ì 204	Í 205	Î 206	Ï 207
D	Ð 208	Ñ 209	Ò 210	Ó 211	Ô 212	Õ 213	Ö 214	× 215	Ø 216	Ù 217	Ú 218	Û 219	Ü 220	Ý 221	Þ 222	ß 223
E	à 224	á 225	â 226	ã 227	ä 228	å 229	æ 230	ç 231	è 232	é 233	ê 234	ë 235	ì 236	í 237	î 238	ï 239
F	ð 240	ñ 241	ò 242	ó 243	ô 244	õ 245	ö 246	÷ 247	ø 248	ù 249	ú 250	û 251	ü 252	ý 253	þ 254	ÿ 255

CHR: h
DEC: 104
HEX: 0x68
BIN: 0110 1000

EEPROM Editor

The EEPROM Editor is used for manipulating MCU's EEPROM memory. You can launch it from the drop-down menu **Tools › EEPROM Editor**. When Use this EEPROM definition is checked compiler will generate Intel hex file project_name.ihex that contains data from EEPROM editor.

When you run mikroElektronika programmer software from mikroPascal PRO for AVR IDE - `project_name.hex` file will be loaded automatically while `ihex` file must be loaded manually.



mikroElektronika EEPROM Tool

EEPROM Editor

☐ Use Eeprom in project

Device: Custom

EEPROM Size: 512 Bytes

EEPROM Fill: Value: 0x FF

EEPROM File

EEPROM Data

Addr	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Char
0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ

EEPROM Edit:

Input Format:
○ Dec
● Hex
○ Float
○ String

EEPROM Address:
Start Address: 0x FFFF
☑ AutoInc

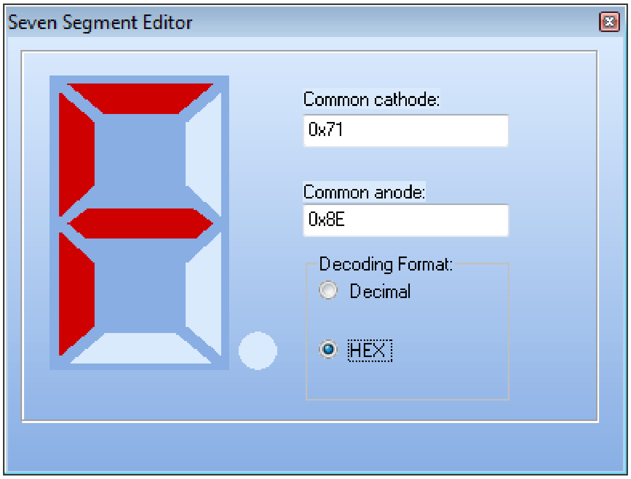
Edit Value:
Hex No: 0x

Size:
○ Byte ○ Word ● DWord

:

7 Segment Display Decoder

The 7 Segment 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 get the requested value in the edit boxes. You can launch it from the drop-down menu **Tools › 7 Segment Decoder** or by clicking the Seven Segment Icon from Tools toolbar.



Seven Segment Editor

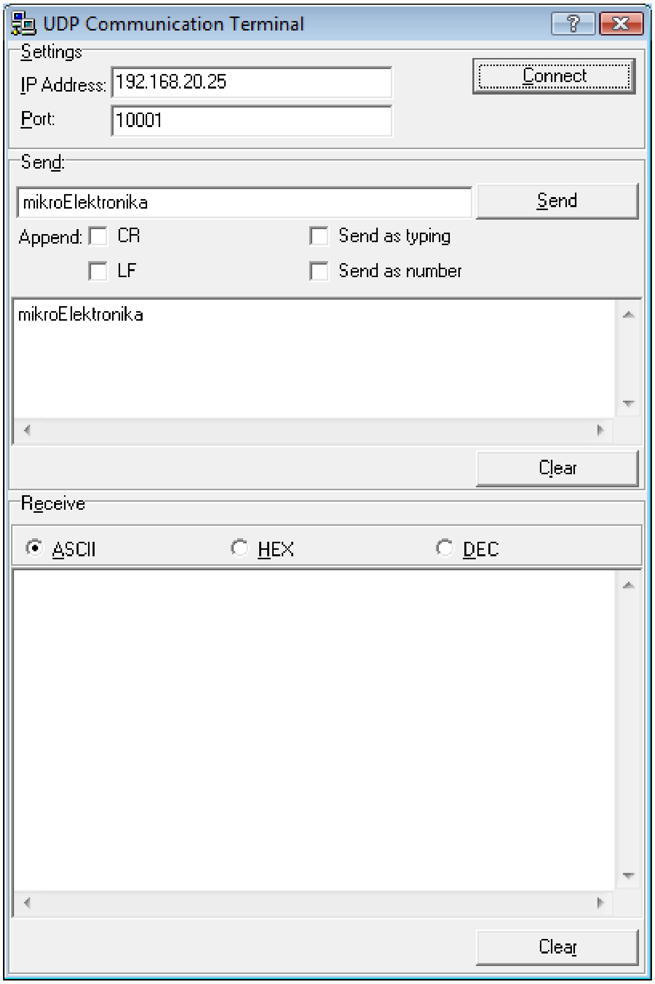
Common cathode: 0x71

Common anode: 0x8E

Decoding Format:
○ Decimal
● HEX

UDP Terminal

The mikroPascal PRO for AVR includes the UDP Terminal. You can launch it from the drop-down menu **Tools › UDP Terminal**.



IP Address: 192.168.20.25

Port: 10001

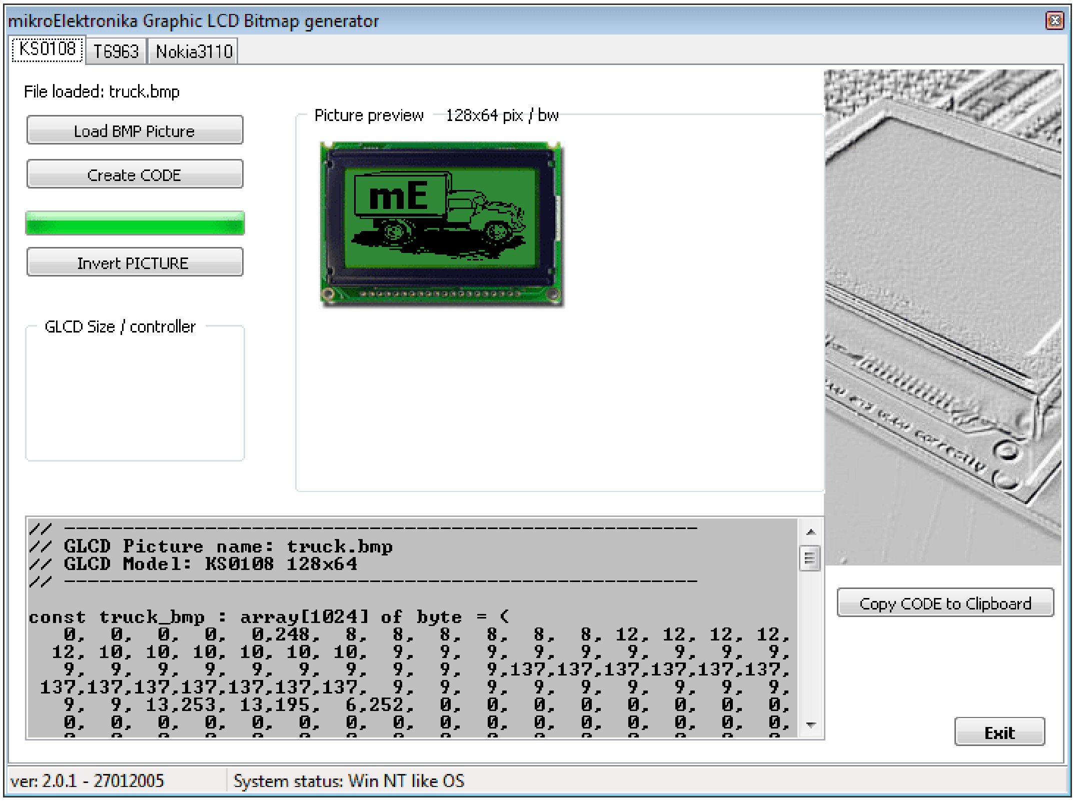
mikroElektronika

mikroElektronika

● ASCII ○ HEX ○ DEC

Graphic Lcd Bitmap Editor

The mikroPascal PRO for AVR includes the Graphic Lcd Bitmap Editor. Output is the mikroPascal PRO for AVR compatible code. You can launch it from the drop-down menu **Tools › Glcd Bitmap Editor**.



File loaded: truck.bmp

128x64 pix / bw

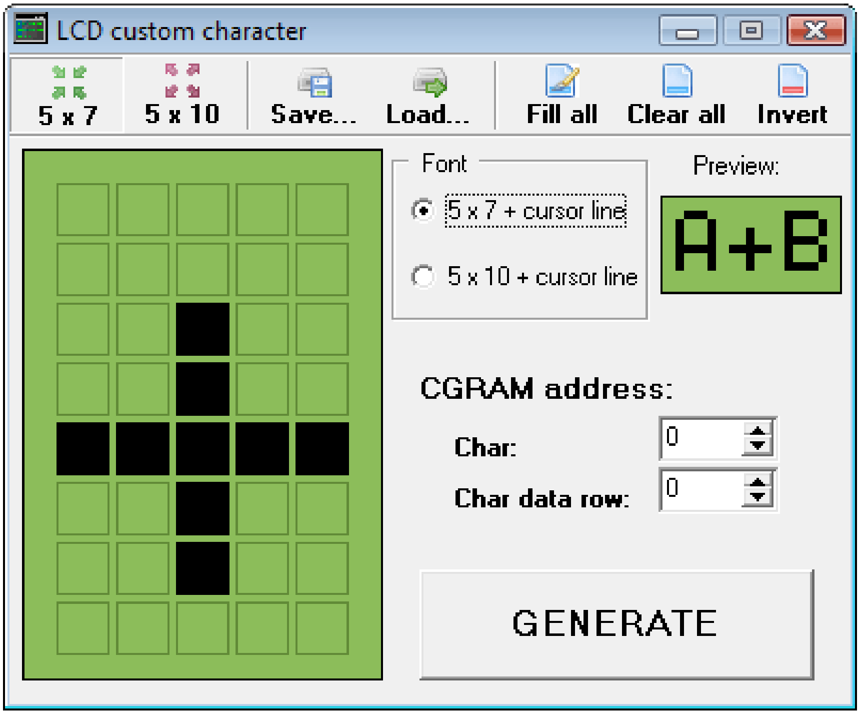
```
// ------------------------------------------------------
// GLCD Picture name: truck.bmp
// GLCD Model: KS0108 128x64
// ------------------------------------------------------

const truck_bmp : array[1024] of byte = (
   0,  0,  0,  0,  0,248,  8,  8,  8,  8,  8,  8, 12, 12, 12, 12,
  12, 10, 10, 10, 10, 10, 10,  9,  9,  9,  9,  9,  9,  9,  9,  9,
   9,  9,  9,  9,  9,  9,  9,  9,  9,  9,137,137,137,137,137,137,
 137,137,137,137,137,137,137,  9,  9,  9,  9,  9,  9,  9,  9,  9,
   9,  9, 13,253, 13,195,  6,252,  0,  0,  0,  0,  0,  0,  0,  0,
   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
```

ver: 2.0.1 - 27012005

System status: Win NT like OS

Lcd Custom Character

mikroPascal PRO for AVR includes the Lcd Custom Character. Output is mikroPascal PRO for AVR compatible code. You can launch it from the drop-down menu **Tools › Lcd Custom Character**.



Font: ● 5 x 7 + cursor line ○ 5 x 10 + cursor line

Preview: A+B

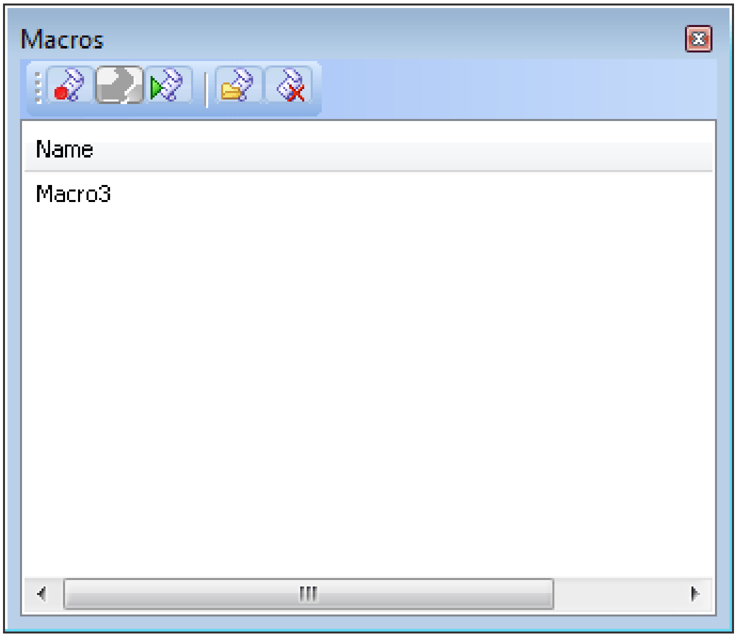
CGRAM address:

Char: 0

Char data row: 0

MACRO EDITOR

A macro is a series of keystrokes that have been 'recorded' in the order performed. A macro allows you to 'record' a series of keystrokes and then 'playback', or repeat, the recorded keystrokes.



Name
Macro3

The Macro offers the following commands:

Icon	Description
●	Starts 'recording' keystrokes for later playback.
■	Stops capturing keystrokesthat was started when the Start Recordig command was selected.
▶	Allows a macro that has been recorded to be replayed.
📂	New macro.
✗	Delete macro.

Related topics: Advanced Code Editor, Code Templates

OPTIONS

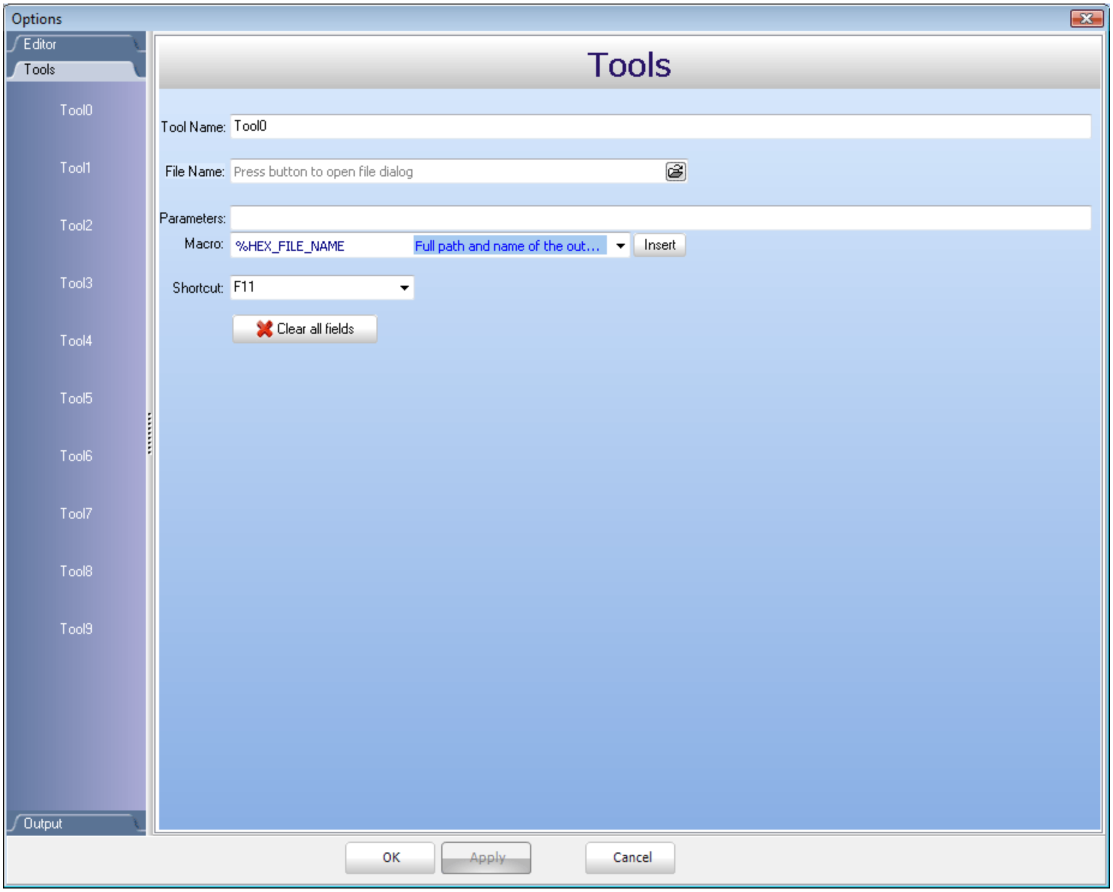
Options menu consists of three tabs: Code Editor, Tools and Output settings

Code editor

The Code Editor is advanced text editor fashioned to satisfy needs of professionals.

Tools

The mikroPascal PRO for AVR includes the Tools tab, which enables the use of shortcuts to external programs, like Calculator or Notepad.
You can set up to 10 different shortcuts, by editing Tool0 - Tool9.



Tool Name: Tool0

File Name: Press button to open file dialog

Parameters:

Macro: %HEX_FILE_NAME Full path and name of the out...

Shortcut: F11

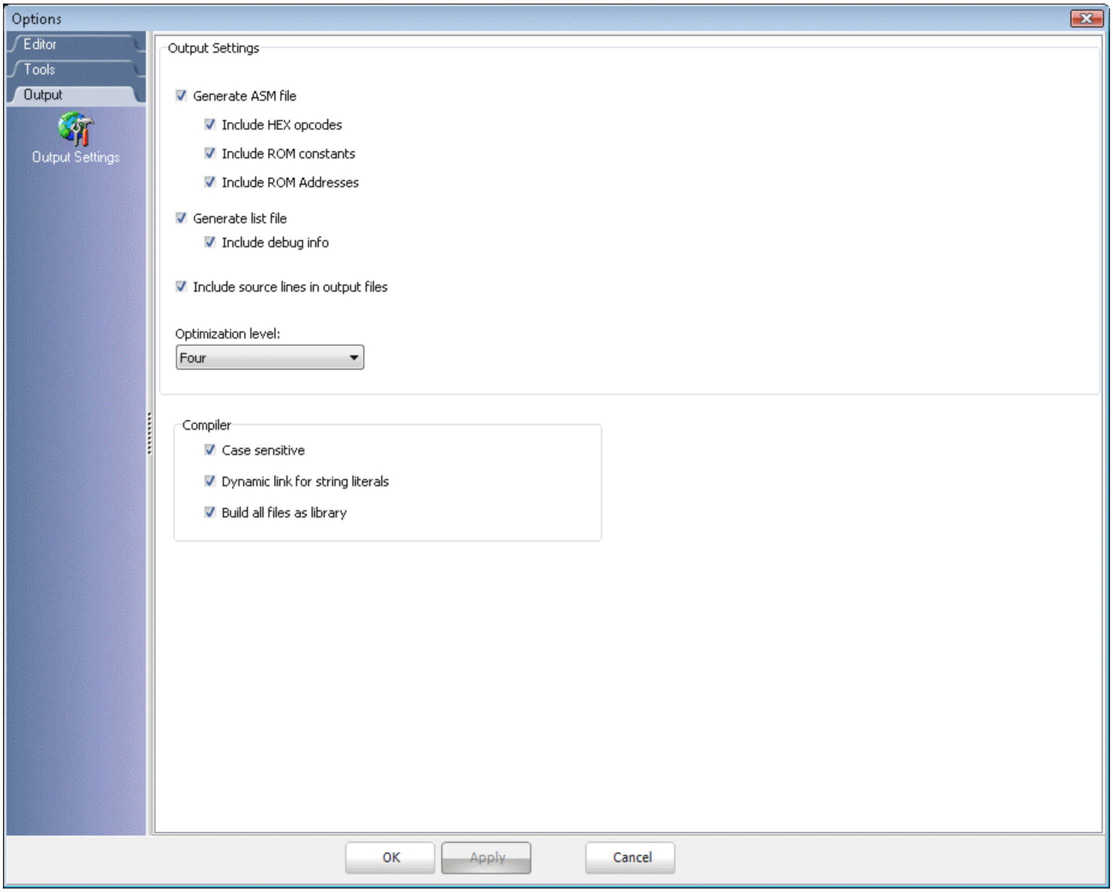
Output settings

By modifying Output Settings, user can configure the content of the output files. You can enable or disable, for example, generation of ASM and List file.

Also, user can choose optimization level, and compiler specific settings, which include case sensitivity, dynamic link for string literals setting (described in mikroPascal PRO for AVR specifics).

Build all files as library enables user to use compiled library (`*.mcl`) on any AVR MCU (when this box is checked), or for a selected AVR MCU (when this box is left unchecked).

For more information on creating new libraries, see Creating New Library.



Output Settings

☑ Generate ASM file

☑ Include HEX opcodes

☑ Include ROM constants

☑ Include ROM Addresses

☑ Generate list file

☑ Include debug info

☑ Include source lines in output files

Optimization level: Four

Compiler

☑ Case sensitive

☑ Dynamic link for string literals

☑ Build all files as library

REGULAR EXPRESSIONS

Introduction

Regular Expressions are a widely-used method of specifying patterns of text to search for. Special metacharacters allow you to specify, for instance, that a particular string you are looking for, occurs at the beginning, or end of a line, or contains `n` recurrences of a certain character.

Simple matches

Any single character matches itself, unless it is a metacharacter with a special meaning described below. A series of characters matches that series of characters in the target string, so the pattern `"short"` would match `"short"` in the target string. You can cause characters that normally function as metacharacters or escape sequences to be interpreted by preceding them with a backslash `"\"`.
For instance, metacharacter `"^"` matches beginning of string, but `"\^"` matches character `"^"`, and `"\\"` matches `"\"`, etc.

Examples :

`unsigned` matches string `'unsigned'`
`\^unsigned` matches string `'^unsigned'`

Escape sequences

Characters may be specified using a escape sequences: `"\n"` matches a newline, `"\t"` a tab, etc. More generally, `\xnn`, where `nn` is a string of hexadecimal digits, matches the character whose ASCII value is `nn`.
If you need wide (Unicode) character code, you can use `'\x{nnnn}'`, where `'nnnn'` - one or more hexadecimal digits.

`\xnn` - char with hex code `nn`
`\x{nnnn)` - char with hex code `nnnn` (one byte for plain text and two bytes for Unicode)
`\t` - tab (HT/TAB), same as `\x09`
`\n` - newline (NL), same as `\x0a`
`\r` - car.return (CR), same as `\x0d`
`\f` - form feed (FF), same as `\x0c`
`\a` - alarm (bell) (BEL), same as `\x07`
`\e` - escape (ESC) , same as `\x1b`

Examples:

`unsigned\x20int` matches `'unsigned int'` (note space in the middle)
`\tunsigned` matches `'unsigned'` (predecessed by tab)

Character classes

You can specify a character class, by enclosing a list of characters in `[]`, which will match any of the characters from the list. If the first character after the `"["` is `"^"`, the class matches any character not in the list.

Examples:

`count[aeiou]r` finds strings `'countar'`, `'counter'`, etc. but not `'countbr'`, `'countcr'`, etc.
`count[^aeiou]r` finds strings `'countbr'`, `'countcr'`, etc. but not `'countar'`, `'counter'`, etc.

Within a list, the `"-"` character is used to specify a range, so that `a-z` represents all characters between `"a"` and `"z"`, inclusive.

If you want `"-"` itself to be a member of a class, put it at the start or end of the list, or precede it with a backslash.
If you want `']'`, you may place it at the start of list or precede it with a backslash.

Examples:

`[-az]` matches `'a'`, `'z'` and `'-'`
`[az-]` matches `'a'`, `'z'` and `'-'`
`[a\-z]` matches `'a'`, `'z'` and `'-'`
`[a-z]` matches all twenty six small characters from `'a'` to `'z'`
`[\n-\x0D]` matches any of `#10,#11,#12,#13`.
`[\d-t]` matches any digit, `'-'` or `'t'`.
`[]-a]` matches any char from `']'..'a'`.

Metacharacters

Metacharacters are special characters which are the essence of regular expressions. There are different types of metacharacters, described below.

Metacharacters - Line separators

`^` - start of line
`$` - end of line
`\A` - start of text
`\Z` - end of text
`.` - any character in line

Examples:

^PORTA - matches string ' PORTA ' only if it's at the beginning of line
PORTA$ - matches string ' PORTA ' only if it's at the end of line
^PORTA$ - matches string ' PORTA ' only if it's the only string in line
PORT.r - matches strings like 'PORTA', 'PORTB', 'PORT1' and so on

The "^" metacharacter by default is only guaranteed to match beginning of the input string/text, and the "$" metacharacter only at the end. Embedded line separators will not be matched by ^" or "$".
You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any line separator within the string, and "$" will match before any line separator. Regular expressions works with line separators as recommended at
http://www.unicode.org/unicode/reports/tr18/

Metacharacters - Predefined classes

\w - an alphanumeric character (including "_")
\W - a nonalphanumeric character
\d - a numeric character
\D - a non-numeric character
\s - any space (same as [\t\n\r\f])
\S - a non space
You may use \w, \d and \s within custom character classes.

Example:

routi\de - matches strings like 'routi1e', 'routi6e' and so on, but not 'routine', 'routime' and so on.

Metacharacters - Word boundaries

A word boundary ("\b") is a spot between two characters that has an alphanumeric character ("\ w") on one side, and a nonalphanumeric character ("\W") on the other side (in either order), counting the imaginary characters off the beginning and end of the string as matching a "\W".

\b - match a word boundary)
\B - match a non-(word boundary)

Metacharacters - Iterators

Any item of a regular expression may be followed by another type of metacharacters - iterators. Using this metacharacters,you can specify number of occurences of previous character, metacharacter or subexpression.

`*` - zero or more ("greedy"), similar to {0,}
`+` - one or more ("greedy"), similar to {1,}
`?` - zero or one ("greedy"), similar to {0,1}
`{n}` - exactly n times ("greedy")
`{n,}` - at least n times ("greedy")
`{n,m}` - at least n but not more than m times ("greedy")
`*?` - zero or more ("non-greedy"), similar to {0,}?
`+?` - one or more ("non-greedy"), similar to {1,}?
`??` - zero or one ("non-greedy"), similar to {0,1}?
`{n}?` - exactly n times ("non-greedy")
`{n,}?` - at least n times ("non-greedy")
`{n,m}?` - at least n but not more than m times ("non-greedy")

So, digits in curly brackets of the form, `{n,m}` , specify the minimum number of times to match the item `n` and the maximum `m`. The form `{n}` is equivalent to `{n,n}` and matches exactly `n` times. The form `{n,}` matches `n` or more times. There is no limit to the size of `n` or `m`, but large numbers will chew up more memory and slow down execution.

If a curly bracket occurs in any other context, it is treated as a regular character.

Examples:

`count.*r ß`- matches strings like `'counter'`, `'countelkjdflkj9r'` and `'countr'`
`count.+r` - matches strings like `'counter'`, `'countelkjdflkj9r'` but not `'countr'`
`count.?r` - matches strings like `'counter'`, `'countar'` and `'countr'` but not `'countelkj9r'`
`counte{2}r` - matches string `'counteer'`
`counte{2,}r` - matches strings like `'counteer'`, `'counteeer'`, `'counteeer'` etc.
`counte{2,3}r` - matches strings like `'counteer'`, or `'counteeer'` but not `'counteeeer'`
A little explanation about "greediness". "Greedy" takes as many as possible, "non-greedy" takes as few as possible.
For example, `'b+'` and `'b*'` applied to string `'abbbbc'` return `'bbbb'`, `'b+?'` returns `'b'`, `'b*?'` returns empty string, `'b{2,3}?'` returns `'bb'`, `'b{2,3}'` returns `'bbb'`.

Metacharacters - Alternatives

You can specify a series of alternatives for a pattern using `"|"` to separate them, so that `bit|bat|bot` will match any of `"bit", "bat"`, or `"bot"` in the target string as would `"b(i|a|o)t)"`. The first alternative includes everything from the last pattern delimiter `("(", "["`, or the beginning of the pattern) up to the first `"|"`, and the last alternative contains everything from the last `"|"` to the next pattern delimiter. For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end.

Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching `rou|rout` against `"routine"`, only the `"rou"` part will match, as that is the first alternative tried, and it successfully matches the target string (this might not seem important, but it is important when you are capturing matched text using parentheses.) Also remember that `"|"` is interpreted as a literal within square brackets, so if you write `[bit|bat|bot]`, you're really only matching `[biao|]`.

Examples:

`rou(tine|te)` - matches strings `'routine'` or `'route'`.

Metacharacters - Subexpressions

The bracketing construct `( ... )` may also be used for define regular subexpressions. Subexpressions are numbered based on the left to right order of their opening parenthesis. First subexpression has number `'1'`

Examples:

`(int){8,10}` matches strings which contain 8, 9 or 10 instances of the `'int'`
`routi([0-9]|a+)e` matches `'routi0e'`, `'routi1e'` , `'routine'`, `'routinne'`, `'routinnne'` etc.

Metacharacters - Backreferences

Metacharacters `\1` through `\9` are interpreted as backreferences. `\` matches previously matched subexpression `#`.

Examples:

`(.)\1+` matches `'aaaa'` and `'cc'`.
`(.+)\1+` matches `'abab'` and `'123123'`
`(['"]?)(\d+)\1` matches `"13"` (in double quotes), or `'4'` (in single quotes) or `77` (without quotes) etc

MIKROPASCAL PRO FOR AVR COMMAND LINE OPTIONS

Usage: `mPAvr.exe [ -<opts> [ -<opts>]] [ <infile> [ -<opts>]] [ -<opts>]]`
Infile can be of `*.mpas and *.mcl type`.

The following parameters and some more (see manual) are valid:

`-P` : MCU for which compilation will be done.
`-FO` : Set oscillator [in MHz].
`-SP` : Add directory to the search path list.
`-N` : Output files generated to file path specified by filename.
`-B` : Save compiled binary files (`*.mcl`) to 'directory'.
`-O` : Miscellaneous output options.
`-DBG` : Generate debug info.
`-L` : Check and rebuild new libraries.
`-DL` : Build all files as libraries.
`-Y` : Dynamic link for string literals.

Example:

```
mPAvr.exe -MSF -DBG -pATMEGA16 -O11111114 -fo8 -N"C:\Lcd\Lcd.mppav" -SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\Defs\" -SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\Uses\LTE64KW\" -SP"C:\Lcd\" "Lcd.mpas" "__Lib_Math.mcl" "__Lib_MathDouble.mcl" "__Lib_System.mcl" "__Lib_Delays.mcl" "__Lib_LcdConsts.mcl" "__Lib_Lcd.mcl"
```

Parameters used in the example:

`-MSF` : Short Message Format; used for internal purposes by IDE.
`-DBG` : Generate debug info.
`-pATMEGA16` : MCU ATMEGA16 selected.
`-O11111114` : Miscellaneous output options.
`-fo8` : Set oscillator frequency [in MHz].
`-N"C:\Lcd\Lcd.mppav" -SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\defs\"` : Output files generated to file path specified by filename.
`-SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\Defs\"` : Add directory to the search path list.
`-SP"C:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\Uses\LTE64KW\"` : Add directory to the search path list.
`-SP"C:\Lcd\"` : Add directory to the search path list.
`"Lcd.mpas" "__Lib_Math.mcl" "__Lib_MathDouble.mcl" "__Lib_System.mcl" "__Lib_Delays.mcl" "__Lib_LcdConsts.mcl" "__Lib_Lcd.mcl"` : Specify input files.

PROJECTS

The mikroPascal PRO for AVR organizes applications into projects, consisting of a single project file (extension `.mcpav`) and one or more source files (extension). mikroPascal PRO for AVR IDE allows you to manage multiple projects (see Project Manager). Source files can be compiled only if they are part of a project.

The project file contains the following information:

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

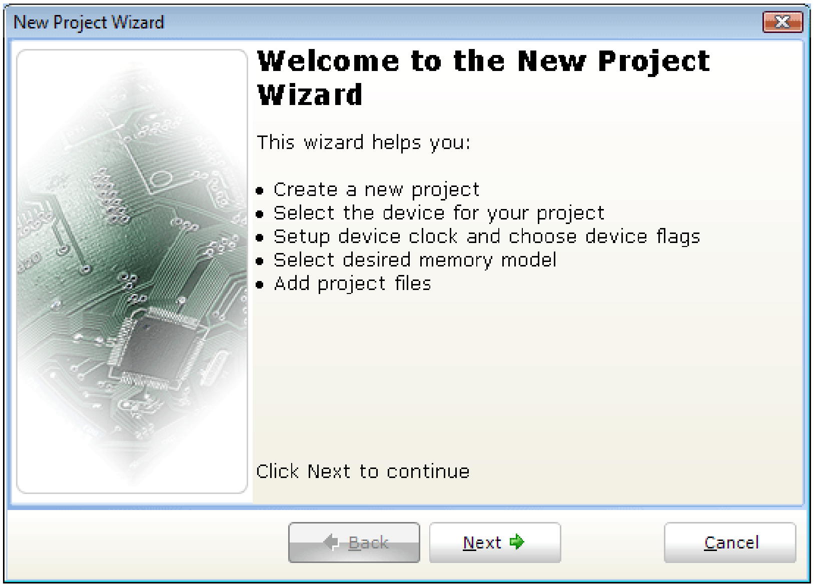
Note that the project does not include files in the same way as preprocessor does, see Add/Remove Files from Project.

New Project

The easiest way to create a project is by means of the New Project Wizard, drop-down menu **Project › New Project** or by clicking the New Project Icon from Project Toolbar.

New Project Wizard Steps

Start creating your New project, by clicking Next button:



New Project Wizard

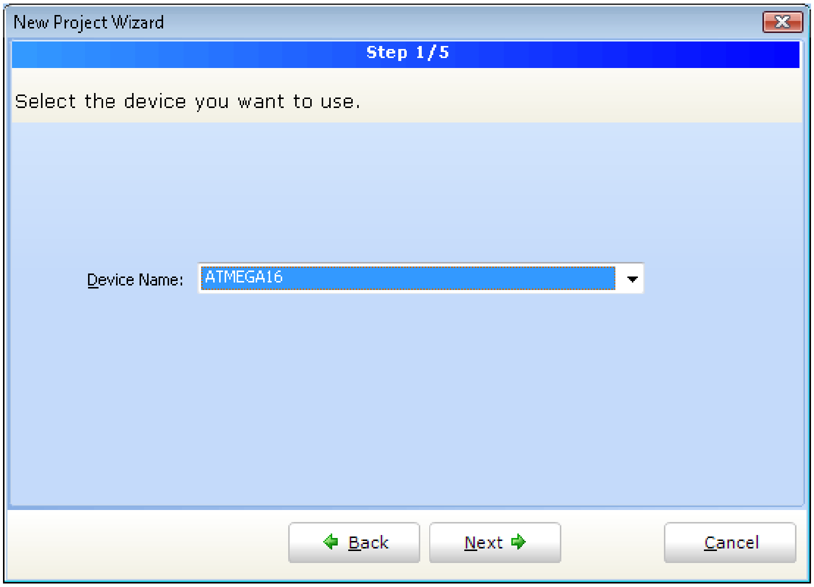
Welcome to the New Project Wizard

This wizard helps you:

- Create a new project
- Select the device for your project
- Setup device clock and choose device flags
- Select desired memory model
- Add project files

Click Next to continue

Step One - Select the device from the device drop-down list.



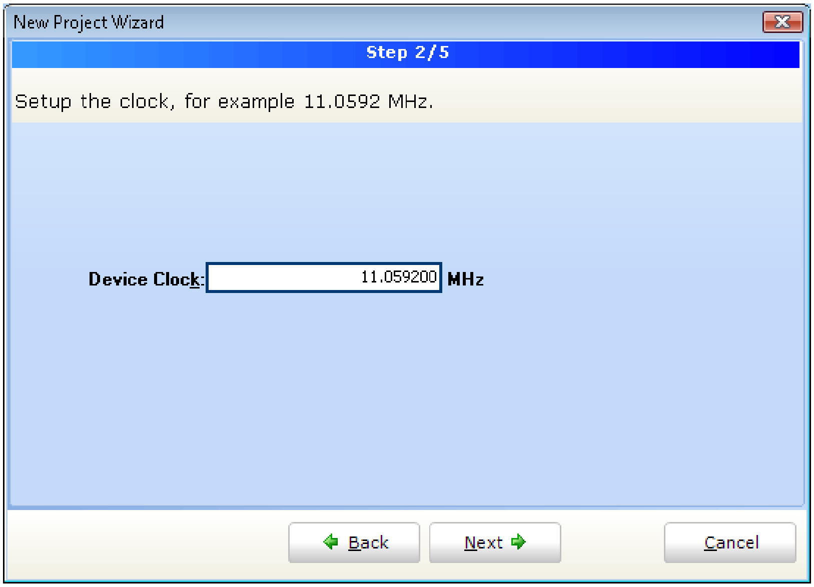
New Project Wizard

Step 1/5

Select the device you want to use.

Device Name: ATMEGA16

Step Two - Enter the oscillator frequency value.



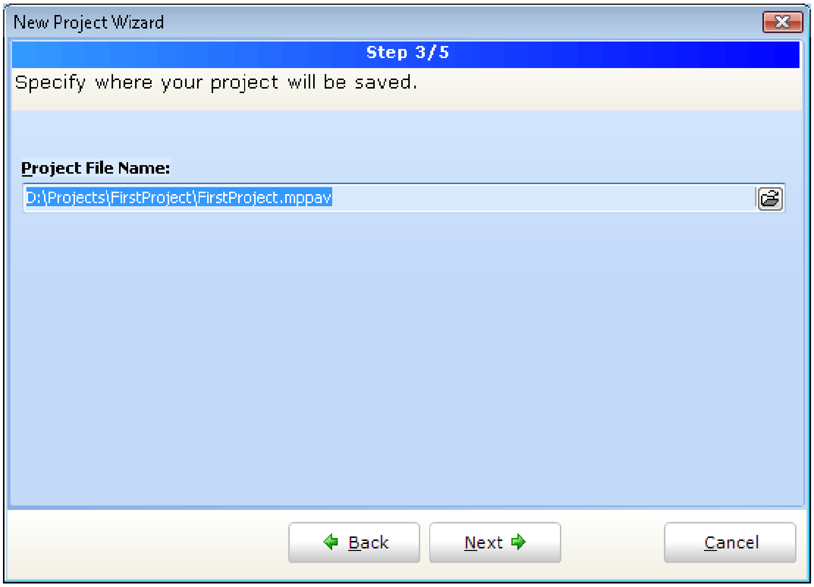
New Project Wizard

Step 2/5

Setup the clock, for example 11.0592 MHz.

Device Clock: 11.059200 MHz

Step Three - Specify the location where your project will be saved.



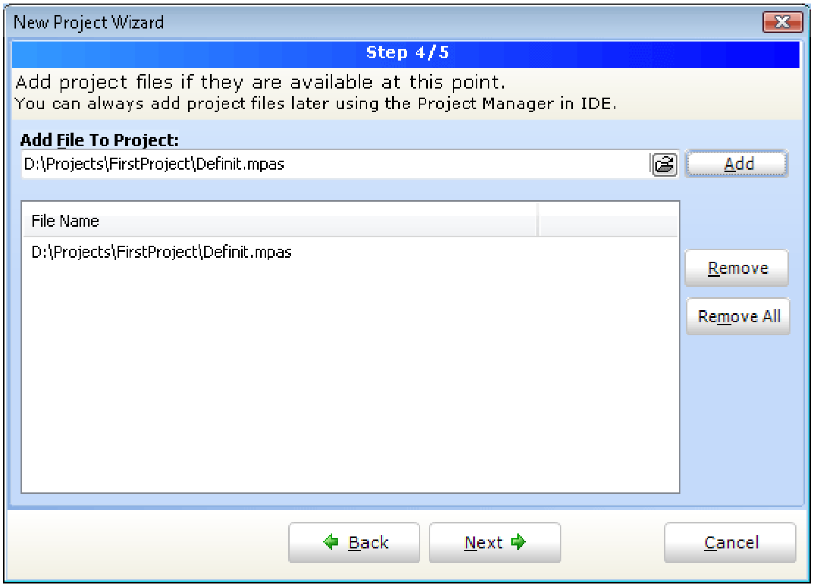
New Project Wizard

Step 3/5

Specify where your project will be saved.

Project File Name: D:\Projects\FirstProject\FirstProject.mppav

Step Four - Add project file to the project if they are avaiable at this point. You can always add project files later using Project Manager.



New Project Wizard

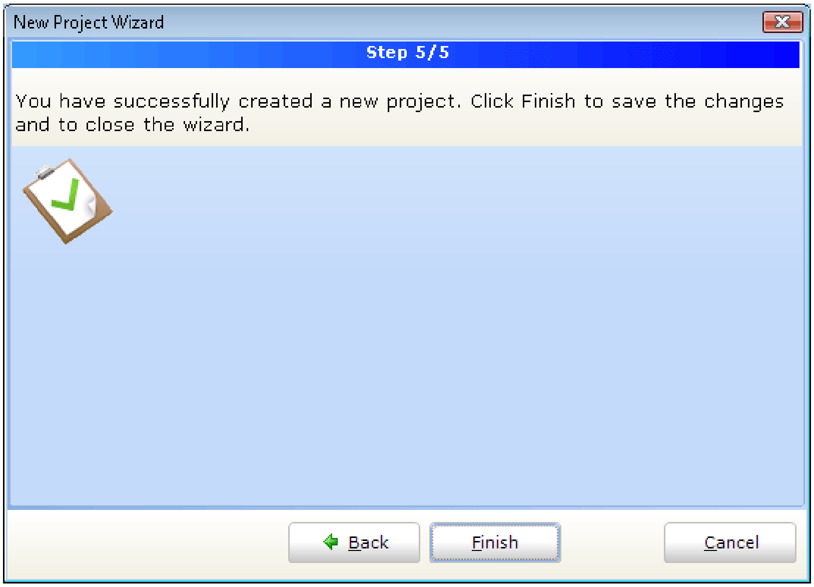
Step 4/5

Add project files if they are available at this point.
You can always add project files later using the Project Manager in IDE.

Add File To Project: D:\Projects\FirstProject\Definit.mpas

File Name
D:\Projects\FirstProject\Definit.mpas

Step Five - Click Finish button to create your New Project:



New Project Wizard

Step 5/5

You have successfully created a new project. Click Finish to save the changes and to close the wizard.

Related topics: Project Manager, Project Settings Customizing Projects
Edit Project

CUSTOMIZING PROJECTS

Edit Project

You can change basic project settings in the Project Settings window. You can change chip and oscillator frequency. Any change in the Project Setting Window affects currently active project only, so in case more than one project is open, you have to ensure that exactly the desired project is set as active one in the Project Manager.

Managing Project Group

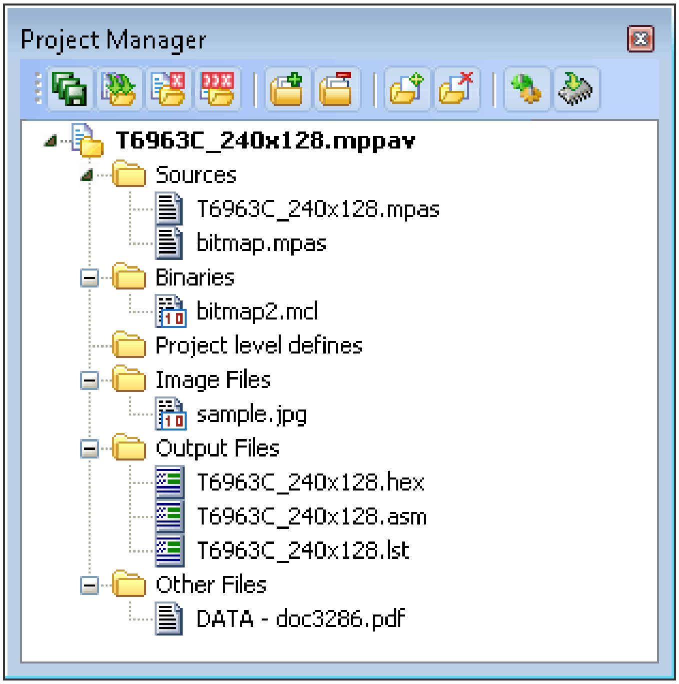
mikroPascal PRO for AVR IDE provides covenient option which enables several projects to be open simultaneously. If you have several projects being connected in some way, you can create a project group.

The project group may be saved by clicking the Save Project Group Icon from the Project Manager window. The project group may be reopend by clicking the Open Project Group Icon .All relevant data about the project group is stored in the project group file (extension `.mpg`)

Add/Remove Files from Project

The project can contain the following file types:

- `.mpas` source files
- `.mcl` binary files
- `.pld` project level defines files
- image files
- `.hex, .asm` and `.lst` files, see output files. These files can not be added or removed from project.
- other files



```
T6963C_240x128.mppav
    Sources
        T6963C_240x128.mpas
        bitmap.mpas
    Binaries
        bitmap2.mcl
    Project level defines
    Image Files
        sample.jpg
    Output Files
        T6963C_240x128.hex
        T6963C_240x128.asm
        T6963C_240x128.lst
    Other Files
        DATA - doc3286.pdf
```

The list of relevant source files is stored in the project file (extension `.mppav`).

To add source file to the project, click the Add File to Project Icon . Each added source file must be self-contained, i.e. it must have all necessary definitions after preprocessing.

To remove file(s) from the project, click the Remove File from Project Icon .

See File Inclusion for more information.

Project Level Defines

Project Level Defines (`.pld`) files can also be added to project. Project level define files enable you to have defines that are visible in all source files in the project. One project may contain several `pld` files. A file must contain one definition per line, for example:

```
ANALOG
DEBUG
TEST
```

There are some predefined project level defines. See predefined project level defines

Related topics: Project Manager, Project Settings

SOURCE FILES

Source files containing Pascal code should have the extension .mpas. The list of source files relevant to the application is stored in project file with extension `.mppav`, along with other project information. You can compile source files only if they are part of the project.

Managing Source Files

Creating new source file

To create a new source file, do the following:

1. Select **File › New Unit** from the drop-down menu, or press Ctrl+N, or click the New File Icon 🗋 from the File Toolbar.
2. A new tab will be opened. This is a new source file. Select **File › Save** from the drop-down menu, or press Ctrl+S, or click the Save File Icon 💾 from the File Toolbar and name it as you want.

If you use the New Project Wizard, an empty source file, named after the project with extension `.mpas`, will be created automatically. The mikroPascal PRO for AVR does not require you to have a source file named the same as the project, it's just a matter of convenience.

Opening an existing file

Select **File › Open** from the drop-down menu, or press Ctrl+O, or click the Open File Icon 📂 from the File Toolbar. In Open Dialog browse to the location of the file that you want to open, select it and click the Open button.
The selected file is displayed in its own tab. If the selected file is already open, its current Editor tab will become active.

Printing an open file

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

Saving file

1. Make sure that the window containing the file that you want to save is the active window.
2. Select **File › Save** from the drop-down menu, or press Ctrl+S, or click the Save File Icon 💾 from the File Toolbar.

Saving file under a different name

1. Make sure that the window containing the file that you want to save is the active window.
2. Select **File › Save As** from the drop-down menu. The New File Name dialog will be displayed.
3. In the dialog, browse to the folder where you want to save the file.
4. In the File Name field, modify the name of the file you want to save.
5. Click the Save button.

Closing file

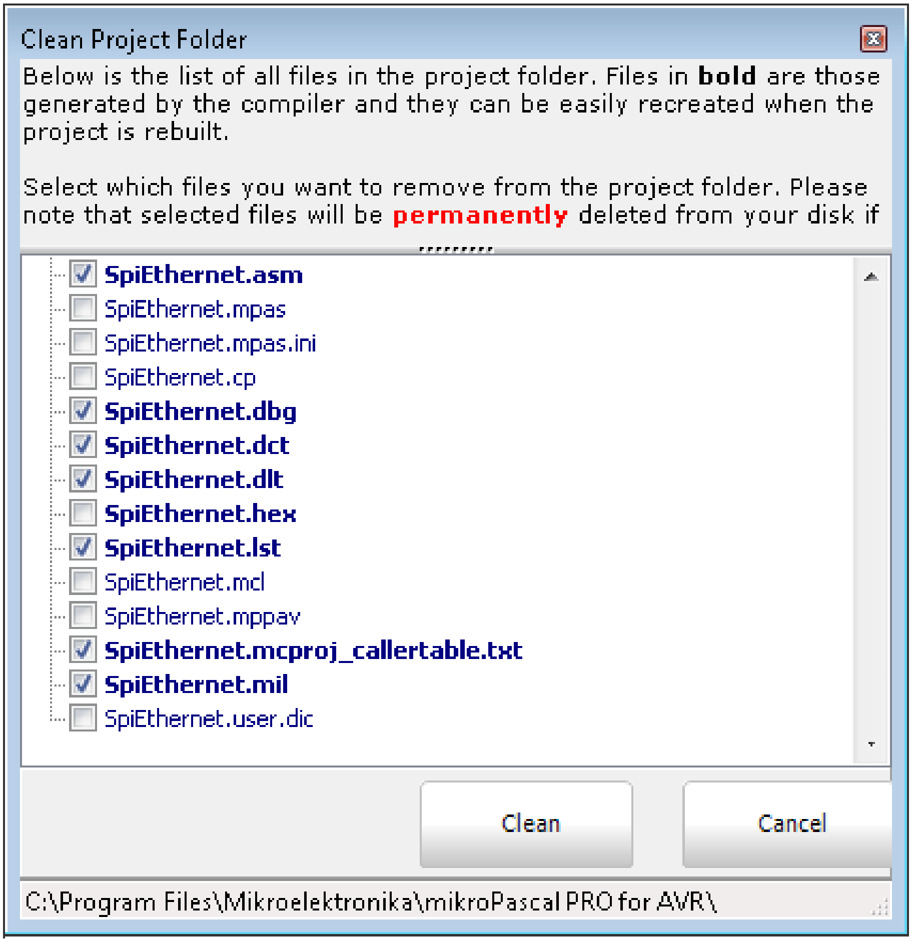
1. Make sure that the tab containing the file that you want to close is the active tab.
2. Select **File › Close** from the drop-down menu, or right click the tab of the file that you want to close and select **Close** option from the context menu.
4. If the file has been changed since it was last saved, you will be prompted to save your changes.

Related topics:File Menu, File Toolbar, Project Manager, Project Settings,

CLEAN PROJECT FOLDER

Clean Project Folder

This menu gives you option to choose which files from your current project you want to delete.
Files marked in bold can be easily recreated by building a project. Other files should be marked for deletion only with a great care, because IDE cannot recover them.



Clean Project Folder

Below is the list of all files in the project folder. Files in **bold** are those generated by the compiler and they can be easily recreated when the project is rebuilt.

Select which files you want to remove from the project folder. Please note that selected files will be **permanently** deleted from your disk if

- ☑ **SpiEthernet.asm**
- □ SpiEthernet.mpas
- □ SpiEthernet.mpas.ini
- □ SpiEthernet.cp
- ☑ **SpiEthernet.dbg**
- ☑ **SpiEthernet.dct**
- ☑ **SpiEthernet.dlt**
- □ **SpiEthernet.hex**
- ☑ **SpiEthernet.lst**
- □ SpiEthernet.mcl
- □ SpiEthernet.mppav
- ☑ **SpiEthernet.mcproj_callertable.txt**
- ☑ **SpiEthernet.mil**
- □ SpiEthernet.user.dic

C:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\

Compilation

When you have created the project and written the source code, it's time to compile it. Select **Project › Build** from the drop-down menu, or click the Build Icon from the Project Toolbar. If more more than one project is open you can compile all open projects by selecting **Project › Build All** from the drop-down menu, or click the Build All Icon from the Project Toolbar.

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

Output Files

Upon successful compilation, the mikroPascal PRO for AVR will generate output files in the project folder (folder which contains the project file .mppav). Output files are summarized in the table below:

Format	Description	File Type
Intel HEX	Intel style hex records. Use this file to program AVR MCU.	`.hex`
Binary	mikro Compiled Library. Binary distribution of application that can be included in other projects.	`.mcl`
List File	Overview of AVR memory allotment: instruction addresses, registers, routines and labels.	`.lst`
Assembler File	Human readable assembly with symbolic names, extracted from the List File.	`.asm`

Assembly View

After compiling the program in the mikroPascal PRO for AVR, you can click the View Assembly icon or select **Project › View Assembly** from the drop-down menu to review the generated assembly code (`.asm` file) in a new tab window. Assembly is human-readable with symbolic names.

Related topics:Project Menu, Project Toolbar, Error Window, Project Manager, Project Settings

Compiler Error Messages:

- `"%s"` is not valid identifier.
- Unknown type `"%s"`.
- Identifier `"%s"` was not declared.
- Syntax error: Expected `"%s"` but `"%s"` found.
- Argument is out of range `"%s"`.
- Syntax error in additive expression.
- File `"%s"` not found.
- Invalid command `"%s"`.
- Not enough parameters.
- Too many parameters.
- Too many characters.
- Actual and formal parameters must be identical.
- Invalid ASM instruction: `"%s"`.
- Identifier `"%s"` has been already declared in `"%s"`.
- Syntax error in multiplicative expression.
- Definition file for `"%s"` is corrupted.
- ORG directive is currently supported for interrupts only.
- Not enough ROM.
- Not enough RAM.
- External procedure `"%s"` used in `"%s"` was not found.
- Internal error: `"%s"`.
- Unit cannot recursively use itself.
- `"%s"` cannot be used out of loop.
- Supplied and formal parameters do not match (`"%s"` to `"%s"`).
- Constant cannot be assigned to.
- Constant array must be declared as global.
- Incompatible types (`"%s"` to `"%s"`).
- Too many characters (`"%s"`).
- Soft_Uart cannot be initialized with selected baud rate/device clock.
- Main label cannot be used in modules.
- Break/Continue cannot be used out of loop.
- Preprocessor Error: `"%s"`.
- Expression is too complicated.
- Duplicated label `"%s"`.
- Complex type cannot be declared here.
- Record is empty.
- Unknown type `"%s"`.
- File not found `"%s"`.
- Constant argument cannot be passed by reference.
- Pointer argument cannot be passed by reference.
- Operator `"%s"` not applicable to these operands `"%s"`.
- Exit cannot be called from the main block.
- Complex type parameter must be passed by reference.

- Error occured while compiling "%s".
- Recursive types are not allowed.
- Adding strings is not allowed, use "strcat" procedure instead.
- Cannot declare pointer to array, use pointer to structure which has array field.
- Return value of the function "%s" is not defined.
- Assignment to for loop variable is not allowed.
- "%s" is allowed only in the main program.
- Start address of "%s" has already been defined.
- Simple constant cannot have fixed address.
- Invalid date/time format.
- Invalid operator "%s".
- File "%s" is not accessible.
- Forward routine "%s" is missing implementation.
- ";" is not allowed before "else".
- Not enough elements: expected "%s", but "%s" elements found.
- Too many elements: expected "%s" elements.
- "external" is allowed for global declarations only.
- Destination size ("%s") does not match source size ("%s").
- Routine prototype is different from previous declaration.
- Division by zero.
- Uart module cannot be initialized with selected baud rate/device clock.
- % cannot be of "%s" type.

Warning Messages:

- Implicit typecast of integral value to pointer.
- Library "%s" was not found in search path.
- Interrupt context saving has been turned off.
- Variable "%s" is not initialized.
- Return value of the function "%s" is not defined.
- Identifier "%s" overrides declaration in unit "%s".
- Generated baud rate is "%s" bps (error = "%s" percent).
- Result size may exceed destination array size.
- Infinite loop.
- Implicit typecast performed from "%s" to "%s".
- Source size ("%s") does not match destination size ("%s").
- Array padded with zeros ("%s") in order to match declared size ("%s").
- Suspicious pointer conversion.

Hint Messages:

- Constant "%s" has been declared, but not used.
- Variable "%s" has been declared, but not used.
- Unit "%s" has been recompiled.
- Variable "%s" has been eliminated by optimizer.
- Compiling unit "%s".

SOFTWARE SIMULATOR OVERVIEW

The Source-level Software Simulator is an integral component of the mikroPascal PRO for AVR environment. It is designed to simulate operations of the AVR MCUs and assist the users in debugging Pascal code written for these devices.

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

Note: The Software Simulator simulates the program flow and execution of instruction lines, but it cannot fully emulate AVR device behavior, i.e. it doesn't update timers, interrupt flags, etc.

Watch Window

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

The Watch Window displays variables and registers of the MCU, along with their addresses and values.

There are two ways of adding variable/register to the watch list:

- by its real name (variable's name in "Pascal" code). Just select desired variable/register from Select variable from list drop-down menu and click the Add Button Add .
- by its name ID (assembly variable name). Simply type name ID of the variable/register you want to display into ***Search the variable by assemby name*** box and click the Add Button Add .

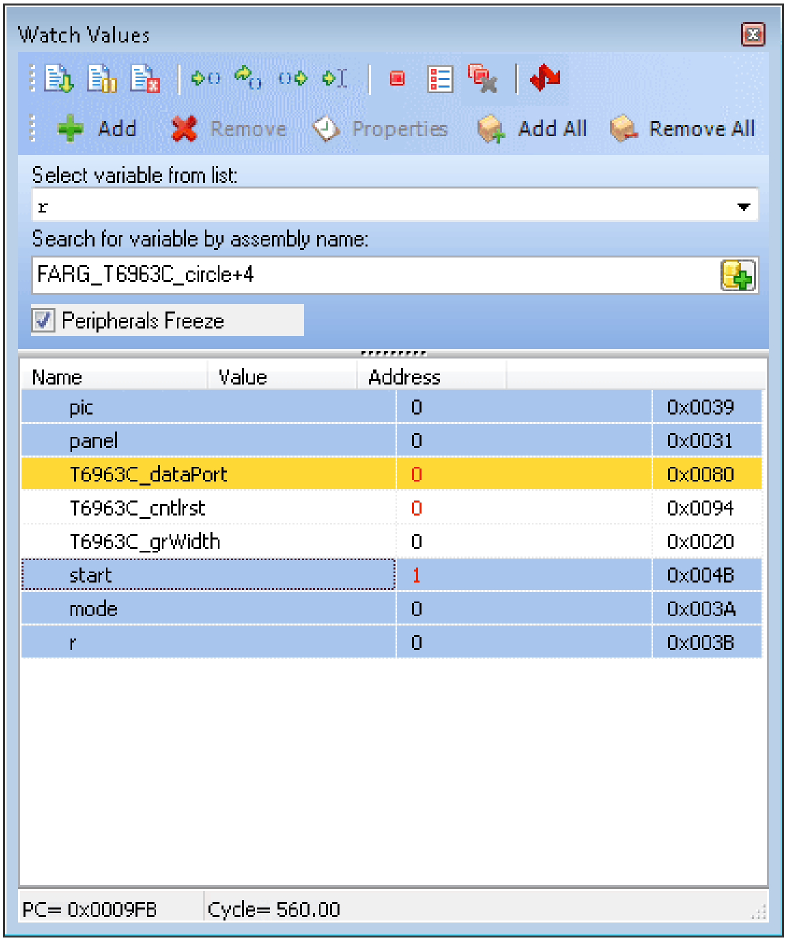
Variables can also be removed from the Watch window, just select the variable that you want to remove and then click the Remove Button Remove .

Add All Button Add All adds all variables.

Remove All Button Remove All removes all variables.

You can also expand/collapse complex variables, i.e. struct type variables, strings...

Values are updated as you go through the simulation. Recently changed items are colored red.



Select variable from list: r

Search for variable by assembly name: FARG_T6963C_circle+4

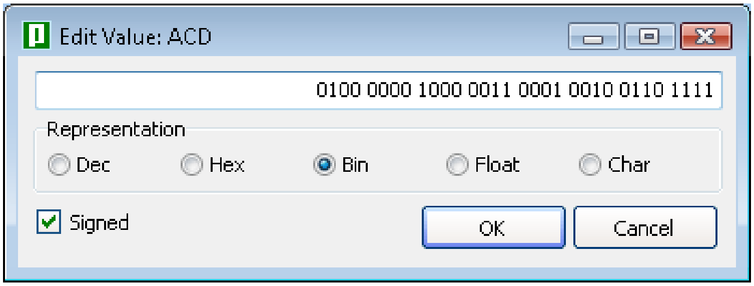
☑ Peripherals Freeze

Name	Value	Address
pic	0	0x0039
panel	0	0x0031
T6963C_dataPort	0	0x0080
T6963C_cntlrst	0	0x0094
T6963C_grWidth	0	0x0020
start	1	0x004B
mode	0	0x003A
r	0	0x003B

PC= 0x0009FB   Cycle= 560.00

Double clicking a variable or clicking the Properties Button Properties opens the Edit Value window in which you can assign a new value to the selected variable/register. Also, you can choose the format of variable/register representation between decimal, hexadecimal, binary, float or character. All representations except float are unsigned by default. For signed representation click the check box next to the **Signed** label.

An item's value can be also changed by double clicking item's value field and typing the new value directly.



Edit Value: ACD

0100 0000 1000 0011 0001 0010 0110 1111

Representation: ○ Dec ○ Hex ● Bin ○ Float ○ Char

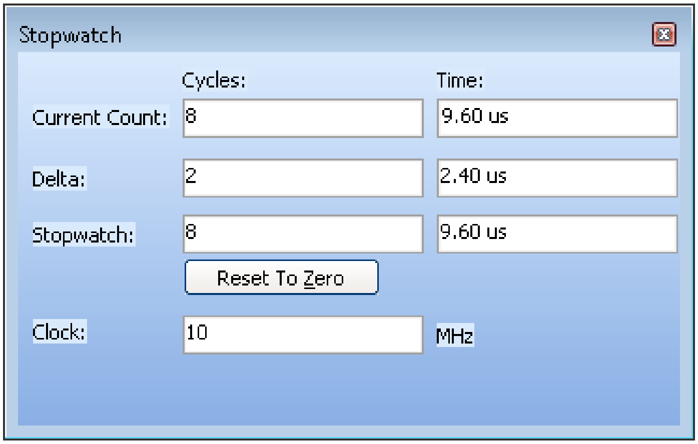
☑ Signed

Stopwatch Window

The Software Simulator Stopwatch Window is available from the drop-down menu, **View › Debug Windows › Stopwatch.**

The Stopwatch Window displays a current count of cycles/time since the last Software Simulator action. Stopwatch measures the execution time (number of cycles) from the moment Software Simulator has started and can be reset at any time. Delta represents the number of cycles between the lines where Software Simulator action has started and ended.

Note: The user can change the clock in the Stopwatch Window, which will recalculate values for the latest specified frequency. Changing the clock in the Stopwatch Window does not affect actual project settings – it only provides a simulation.



Stopwatch

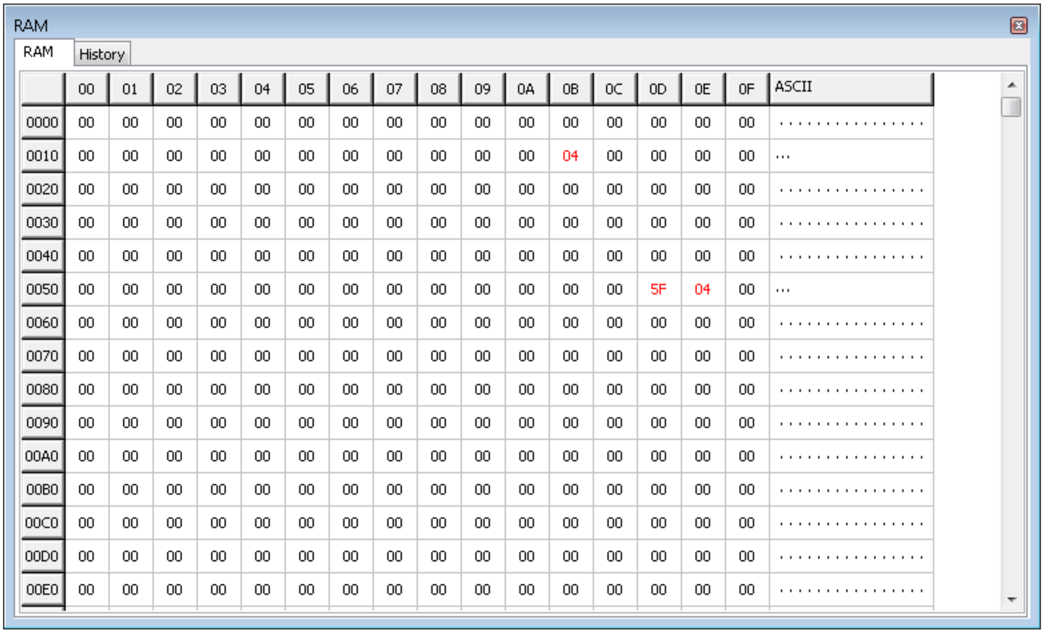
	Cycles:	Time:
Current Count:	8	9.60 us
Delta:	2	2.40 us
Stopwatch:	8	9.60 us

Clock: 10 MHz

RAM Window

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

The RAM Window displays a map of MCU’s RAM, with recently changed items colored red. You can change value of any field by double-clicking it.



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0010	00	00	00	00	00	00	00	00	00	00	00	04	00	00	00	00	...
0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0050	00	00	00	00	00	00	00	00	00	00	00	00	00	5F	04	00	...
0060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Software Simulator Options

Name	Description	Function Key	Toolbar Icon
Start Debugger	Start Software Simulator.	[F9]	⬇
Run/Pause Debugger	Run or pause Software Simulator.	[F6]	⏸
Stop Debugger	Stop Software Simulator.	[Ctrl+F2]	✖
Toggle Breakpoints	Toggle breakpoint at the current cursor position. To view all breakpoints, select Run > View Breakpoints from the drop–down menu. Double clicking an item in the Breakpoints Window List locates the breakpoint.	[F5]	🟥
Run to cursor	Execute all instructions between the current instruction and cursor position.	[F4]	➜I
Step Into	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.	[F7]	➜{}
Step Over	Execute the current Pascal (single or multi–cycle) instruction, then halt.	[F8]	➜{}
Step Out	Execute all remaining instructions in the current routine, return and then halt.	[Ctrl+F8]	{}➜

Related topics: Run Menu, Debug Toolbar

CREATING NEW LIBRARY

mikroBasic PRO for AVR allows you to create your own libraries. In order to create a library in mikroBasic PRO for AVR follow the steps bellow:

1. Create a new Pascal source file, see Managing Source Files
2. Save the file in one of the subfolders of the compiler's Uses folder (LTE64kW or GT64kW, see note on the end of the page):
   `DriveName:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\Uses\LTE64kW\__Lib_Example.mpas`
3. Write a code for your library and save it.
4. Add `__Lib_Example` file in some project, see Project Manager. Recompile the project.
   If you wish to use this library for all MCUs, then you should go to **Tools › Options › Output settings**, and check Build all files as library box.
   This will build libraries in a common form which will work with all MCUs. If this box is not checked, then library will be build for selected MCU.
   Bear in mind that compiler will report an error if a library built for specific MCU is used for another one.
5. Compiled file `__Lib_Example.mcl should appear in ...\mikroBasic PRO for AVR\Uses\LTE64kW\` folder.
6. Open the definition file for the MCU that you want to use. This file is placed in the compiler's Defs folder:
   `DriveName:\Program Files\Mikroelektronika\mikroPascal PRO for AVR\Defs\` and it is named `MCU_NAME.mlk`, for example `ATMEGA16.mlk`
7. Add the the following segment of code to `<LIBRARIES>` node of the definition file (definition file is in XML format):

```
<LIB>
  <ALIAS>Example_Library</ALIAS>
  <FILE>__Lib_Example</FILE>
  <TYPE>REGULAR</TYPE>
  </LIB>
```

8. Add Library to mlk file for each MCU that you want to use with your library.
9. Click Refresh button in Library Manager
10. `Example_Library` should appear in the Library manager window.

Multiple Library Versions

Library Alias represents unique name that is linked to corresponding Library `.mcl` file. For example UART library for ATMEGA16 is different from UART library for ATMEGA128 MCU. Therefore, two different UART Library versions were made, see `mlk` files for these two MCUs. Note that these two libraries have the same Library Alias (UART) in both mlk files. This approach enables you to have identical representation of UART library for both MCUs in Library Manager.

Note: In the Uses folder, there should be two subfolders, LTE64kW and GT64kW, depending on the Flash memory size of the desired MCU. See AVR Specifics for a detailed information regarding this subject.

Related topics: Library Manager, Project Manager, Managing Source Files

CHAPTER 3

mikropascal PRO for AVR Specifics

The following topics cover the specifics of mikroPascal PRO for AVR compiler:

- Pascal Standard Issues
- Predefined Globals and Constants
- Accessing Individual Bits
- Interrupts
- AVR Pointers
- Linker Directives
- Built-in Routines
- Code Optimization

PASCAL STANDARD ISSUES

Divergence from the Pascal Standard

- Function recursion is not supported because of no easily-usable stack and limited memory AVR Specific

Pascal Language Extensions

mikroPascal PRO for AVR has additional set of keywords that do not belong to the standard Pascal language keywords:

- `code`
- `data`
- `io`
- `rx`
- `sfr`
- `register`
- `at`
- `sbit`
- `bit`

Related topics: Keywords, AVR Specific

PREDEFINED GLOBALS AND CONSTANTS

To facilitate programming of AVR compliant MCUs, the mikroPascal PRO for AVR implements a number of predefined globals and constants.

All AVR **SFR registers** are implicitly declared as global variables of volatile word. These identifiers have an external linkage, and are visible in the entire project. When creating a project, the mikroPascal PRO for AVR will include an appropriate (`*.mpas`) file from defs folder, containing declarations of available **SFR registers** and constants.

Math constants

In addition, several commonly used math constants are predefined in mikroPascal PRO for AVR:

```
PI           =   3.1415926
PI_HALF      =   1.5707963
TWO_PI       =   6.2831853
E            =   2.7182818
```

For a complete set of predefined globals and constants, look for “Defs” in the mikroPascal PRO for AVR installation folder, or probe the Code Assistant for specific letters (Ctrl+Space in the Code Editor).

Predefined project level defines

These defines are based on a value that you have entered/edited in the current project, and it is equal to the name of selected device for the project.
If ATmega16 is selected device, then ATmega16 token will be defined as 1, so it can be used for conditional compilation:

```
{$IFDEF ATmega16}
...
{$ENDIF}
```

Related topics: Project level defines

ACCESSING INDIVIDUAL BITS

The mikroPascal PRO for AVR allows you to access individual bits of 8-bit variables. It also supports sbit and bit data types

Accessing Individual Bits Of Variables

To access the individual bits, simply use the direct member selector (.) with a variable, followed by one of identifiers `B0, B1, … , B7, or 0, 1, … 7`, with `7` being the most significant bit :

```
// Clear bit 0 on PORTA
PORTA.B0 := 0;

// Clear bit 5 on PORTB
PORTB.5 := 0;
```

There is no need of any special declarations. This kind of selective access is an intrinsic feature of mikroPascal PRO for AVR and can be used anywhere in the code. Identifiers `B0-B7` are not case sensitive and have a specific namespace. You may override them with your own members `B0-B7` within any given structure.

See Predefined Globals and Constants for more information on register/bit names.

sbit type

The mikroPascal PRO for AVR compiler has sbit data type which provides access to bit-addressable SFRs. You can access them in several ways:

```
var LEDA : sbit at PORTA.B0;
var name : sbit at sfr-name.B<bit-position>;

var LEDB : sbit at PORTB.0;
var name : sbit at sfr-name.<bit-position>;
```

bit type

The mikroPascal PRO for AVR compiler provides a bit data type that may be used for variable declarations. It can not be used for argument lists, and function-return values.

```
var bf : bit;          // bit variable
```

There are no pointers to bit variables:

```
~~var ptr : ^bit;~~       // invalid
```

An array of type bit is not valid:

```
~~var arr[5] : bit;~~     // invalid
```

Note :

- Bit variables can not be initialized.
- Bit variables can not be members of records.
- Bit variables do not have addresses, therefore unary operator @ (address of) is not applicable to these variables.

Related topics: Predefined globals and constants

INTERRUPTS

AVR derivates acknowledges an interrupt request by executing a hardware generated CALL to the appropriate servicing routine ISRs. ISRs are organized in IVT. ISR is defined as a standard function but with the org directive afterwards which connects the function with specific interrupt vector. For example org 0x000B is IVT address of Timer/Counter 2 Overflow interrupt source of the ATMEGA16.
For more information on interrupts and IVT refer to the specific data sheet.

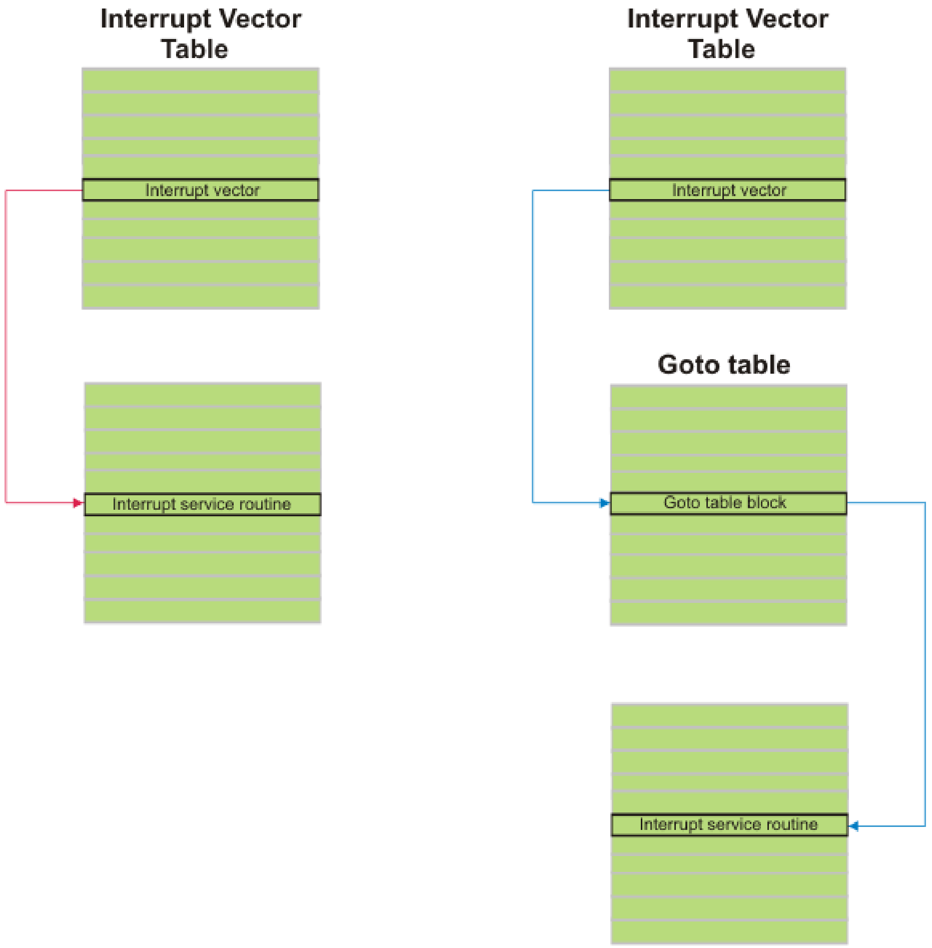
Function Calls from Interrupt

Calling functions from within the interrupt routine is allowed. The compiler takes care about the registers being used, both in "interrupt" and in "main" thread, and performs "smart" context-switching between them two, saving only the registers that have been used in both threads. It is not recommended to use function call from interrupt. In case of doing that take care of stack depth.

```
// Interrupt routine
procedure Interrupt(); org 0x16;
begin
    RS485Master_Receive(dat);
end;
```

Most of the MCUs can access interrupt service routines directly, but some can not reach interrupt service routines if they are allocated on addresses greater than 2K from the IVT. In this case, compiler automatically creates Goto table, in order to jump to such interrupt service routines.

These principles can be explained on the picture below :



Related topics: Pascal standard issues

LINKER DIRECTIVES

mikroPascal PRO for AVR uses internal algorithm to distribute objects within memory. If you need to have a variable or a routine at the specific predefined address, use the linker directives absolute and org.

Note: You must specify an even address when using the linker directives.

Directive absolute

Directive `absolute` specifies the starting address in RAM for a variable. If the variable spans more than 1 word (16-bit), the higher words will be stored at the consecutive locations.

Directive absolute is appended to the declaration of a variable:

```
var x : word; absolute $32;
// Variable x will occupy 1 word (16 bits) at address $32

    y : longint; absolute $34;
// Variable y will occupy 2 words at addresses $34 and $36
```

Be careful when using the absolute directive because you may overlap two variables by accident. For example:

```
var  i : word; absolute $42;
// Variable i will occupy 1 word at address $42;

     jj : longint; absolute $40;
// Variable will occupy 2 words at $40 and $42; thus,
// changing i changes jj at the same time and vice versa
```

Note: You must specify an even address when using the `absolute` directive.

Directive org

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

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

org directive can be used with main routine too. For example:

```
program Led_Blinking;

procedure some_proc();
begin
 ...
end;


org 0x800;              // main procedure starts at 0x800
begin
  DDRB := 0xFF;

  while TRUE do
    begin
      PORTB := 0x00;
      Delay_ms(500);
      PORTB := 0xFF;
      Delay_ms(500);
    end;
end.
```

Note: You must specify an even address when using the org directive.

BUILT-IN ROUTINES

The mikroPascal PRO for AVR compiler provides a set of useful built-in utility functions.

The `Delay_us` and `Delay_ms` routines are implemented as "inline"; i.e. code is generated in the place of a call, so the call doesn't count against the nested call limit.

The `Vdelay_ms, Delay_Cyc` and `Get_Fosc_kHz` are actual Pascal routines. Their sources can be found in `Delays.mpas` file located in the `uses` folder of the compiler.

- Lo
- Hi
- Higher
- Highest
- Inc
- Dec
- Delay_us
- Delay_ms
- Vdelay_ms
- Delay_Cyc
- Clock_Khz
- Clock_Mhz
- SetFuncCall

Lo

Prototype	`function Lo(number: longint): byte;`
Returns	Lowest 8 bits (byte)of number, bits 7..0.
Description	Function returns the lowest byte of `number`. Function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. 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.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x1AC30F4;`
	`tmp := Lo(d);   // Equals 0xF4`

Hi

Prototype	`function Hi(number: longint): byte;`
Returns	Returns next to the lowest byte of `number`, bits 8..15.
Description	Function returns next to the lowest byte of `number`. Function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. 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.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x1AC30F4;`
	`tmp := Hi(d);   // Equals 0x30`

Higher

Prototype	`function Higher(number: longint): byte;`
Returns	Returns next to the highest byte of `number`, bits 16..23.
Description	Function returns next to the highest byte of `number`. Function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. 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.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x1AC30F4;`
	`tmp := Higher(d);   // Equals 0xAC`

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. 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.
Requires	Arguments must be variable of scalar type (i.e. Arithmetic Types and Pointers).

Example	`d := 0x1AC30F4;`
	`tmp := Highest(d);   // Equals 0x01`

Inc

Prototype	`procedure Inc(var par : longint);`
Returns	Nothing.
Description	Increases parameter par by 1.
Requires	Nothing.

Example	`p := 4;`
	`Inc(p);   // p is now 5`

Dec

Prototype	`procedure Dec(var par : longint);`
Returns	Nothing.
Description	Decreases parameter par by 1.
Requires	Nothing.

Example	`p := 4;`
	`Dec(p);   // p is now 3`

Delay_us

Prototype	`procedure Delay_us(time_in_us: const longword);`
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. 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.
Requires	Nothing.
Example	`Delay_us(1000);   // One millisecond pause`

Delay_ms

Prototype	`procedure Delay_ms(time_in_ms: const longword);`
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. 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.
Requires	Nothing.
Example	`Delay_ms(1000);   // One second pause`

Vdelay_ms

Prototype	`procedure Vdelay_ms(time_in_ms: word);`
Returns	Nothing.
Description	Creates a software delay in duration of time_in_ms milliseconds (a variable). Generated delay is not as precise as the delay created by Delay_ms. Note that Vdelay_ms is library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Requires	Nothing.

Example	`pause := 1000;`
	`// ...`
	`Vdelay_ms(pause);   // ~ one second pause`

Delay_Cyc

Prototype	`procedure Delay_Cyc(Cycles_div_by_10: byte);`
Returns	Nothing.
Description	Creates a delay based on MCU clock. Delay lasts for 10 times the input parameter in MCU cycles. Note that Delay_Cyc is library function rather than a built-in routine; it is presented in this topic for the sake of convenience. There are limitations for Cycles_div_by_10 value. Value Cycles_div_by_10 must be between 2 and 257.
Requires	Nothing.
Example	`Delay_Cyc(10);   // Hundred MCU cycles pause`

Clock_KHz

Prototype	`function Clock_KHz(): word;`
Returns	Device clock in KHz, rounded to the nearest integer.
Description	Function returns device clock in KHz, rounded to the nearest integer. 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.
Requires	Nothing.
Example	`clk := Clock_kHz();`

Clock_MHz

Prototype	`function Clock_MHz(): byte;`
Returns	Device clock in MHz, rounded to the nearest integer.
Description	Function returns device clock in MHz, rounded to the nearest integer. 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.
Requires	Nothing.
Example	`clk := Clock_MHz();`

SetFuncCall

Prototype	`procedure SetFuncCall(FuncName: string);`
Returns	Nothing.
Description	Function informs the linker about a specific routine being called. SetFuncCall has to be called in a routine which accesses another routine via a pointer. Function prepares the caller tree, and informs linker about the procedure usage, making it possible to link the called routine.
Requires	Nothing.

Example	`procedure first(p, q: byte);`
	`begin`
	`...`
	`  SetFuncCall(second); // let linker know that we will call the routine 'second'`
	`...`
	`end`

CODE OPTIMIZATION

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

Constant folding

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

Constant propagation

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

Copy propagation

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

Value numbering

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

"Dead code" ellimination

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

Stack allocation

Temporary registers ("Stacks") are being used more rationally, allowing VERY complex expressions to be evaluated with a 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 more attention is payed to implement specific solutions for the code "building bricks" that further reduce output code size.

CHAPTER 4

AVR Specifics

Types Efficiency

First of all, you should know that AVR ALU, which performs arithmetic operations, is optimized for working with bytes. Although mikroPascal PRO is capable of handling very complex data types, AVR may choke on them, especially if you are working on some of the older models. This can dramatically increase the time needed for performing even simple operations. Universal advice is to use the smallest possible type in every situation. It applies to all programming in general, and doubly so with microcontrollers. Types efficiency is determined by the part of RAM memory that is used to store a variable/constant.

Nested Calls Limitations

There are no Nested Calls Limitations, except by RAM size. A Nested call represents a function call to another function within the function body. With each function call, the stack increases for the size of the returned address. Number of nested calls is equel to the capacity of RAM which is left out after allocation of all variables.

Important notes:

- There are many different types of derivates, so it is necessary to be familiar with characteristics and special features of the microcontroller in you are using.
- Some of the AVR MCUs have hardware multiplier. Due to this, be sure to pay atten tion when porting code from one MCU to another, because compiled code can vary by its size.
- Not all microcontrollers share the same instruction set. It is advisable to carefully read the instruction set of the desired MCU, before you start writing your code. Compiler automatically takes care of appropiate instruction set, and if unapropriate asm instruction is used in in-line assembly, compiler will report an error.
- Program counter size is MCU dependent. Thus, there are two sets of libraries :

- MCUs with program counter size larger than 16 bits (flash memory size larger than 128kb)
- MCUs with program counter size less or equal 16 bits (flash memory size smaller than 128kb)
- Assembly SPM instruction and its derivates must reside in Boot Loader section of program memory.
- Part of flash memory can be dedicated to Boot Loader code. For details, refer to AVR memory organization.

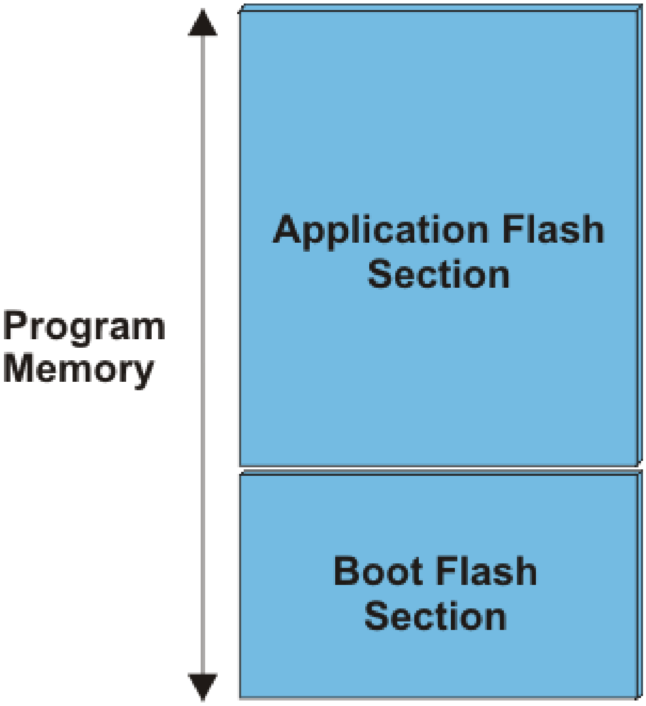
Related topics: mikroPascal PRO for AVR specifics, AVR memory organization

AVR Memory Organization

The AVR microcontroller's memory is divided into Program Memory and Data Memory. Program Memory (ROM) is used for permanent saving program being executed, while Data Memory (RAM) is used for temporarily storing and keeping intermediate results and variables.

Program Memory (ROM)

Program Memory (ROM) is used for permanent saving program (CODE) being executed, and it is divided into two sections, Boot Program section and the Application Program section. The size of these sections is configured by the BOOTSZ fuse. These two sections can have different level of protection since they have different sets of Lock bits.
Depending on the settings made in compiler, program memory may also used to store a constant variables. The AVR executes programs stored in program memory only. `code` memory type specifier is used to refer to program memory.



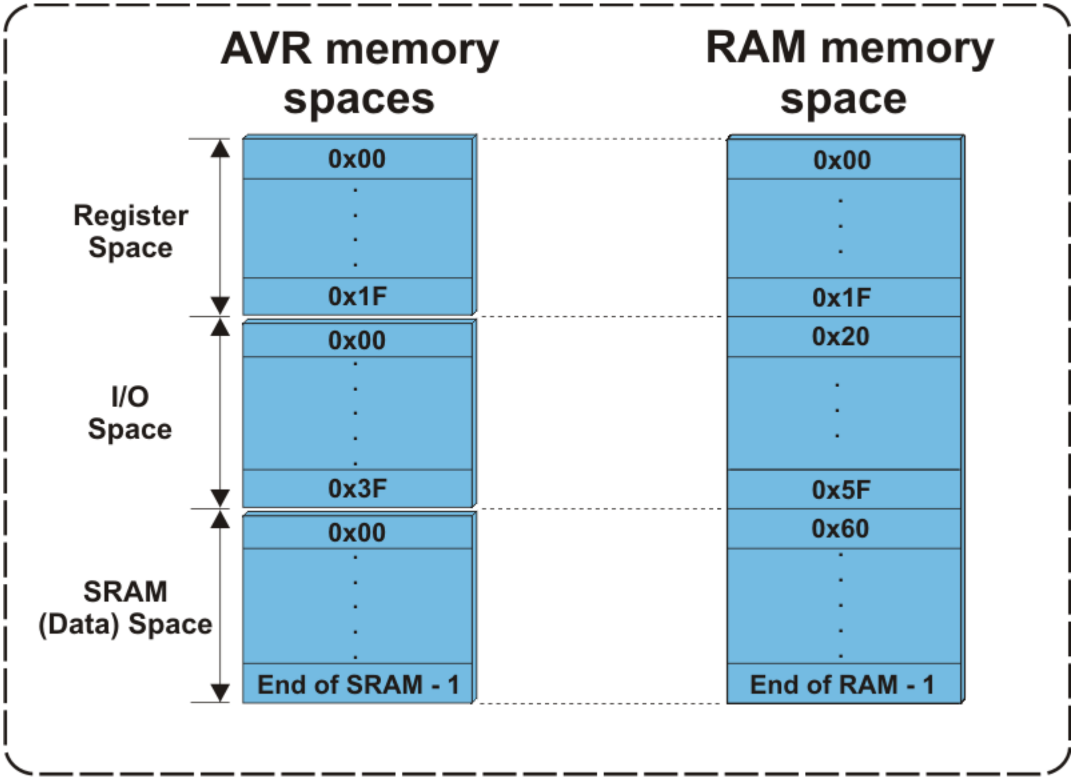
Data Memory

Data memory consists of :

- Rx space
- I/O Memory
- Extended I/O Memory (MCU dependent)
- Internal SRAM

Rx space consists of 32 general purpose working 8-bit registers (R0-R31). These registers have the shortest (fastest) access time, which allows single-cycle Arithmetic Logic Unit (ALU) operation.
I/O Memory space contains addresses for CPU peripheral function, such as Control registers, SPI, and other I/O functions.
Due to the complexity, some AVR microcontrollers with more peripherals have Extended I/O memory, which occupies part of the internal SRAM. Extended I/O memory is MCU dependent.
Storing data in I/O and Extended I/O memory is handled by the compiler only. Users can not use this memory space for storing their data.
Internal SRAM (Data Memory) is used for temporarily storing and keeping intermediate results and variables (static link and dynamic link).

There are four memory type specifiers that can be used to refer to the data memory: `rx`, `data`, `io`, `sfr` and `register`.



Related topics: Accessing individual bits, SFRs, Memory type specifiers

MEMORY TYPE SPECIFIERS

The mikroPascal PRO for AVR supports usage of all memory areas. Each variable may be explicitly assigned to a specific memory space by including a memory type specifier in the declaration, or implicitly assigned.

The following memory type specifiers can be used:

- code
- data
- rx
- io
- sfr

Memory type specifiers can be included in variable declaration.
For example:

```
var data_buffer : char; data;        // puts data_buffer in data ram
const txt = 'Enter parameter'; code; // puts text in program memory
```

code

Description	The `code` memory type may be used for allocating constants in program memory.

Example	`// puts txt in program memory`
	`const txt = 'Enter parameter'; code;`

data

Description	This memory specifier is used when storing variable to the internal data SRAM.

Example	`// puts data_buffer in data ram`
	`var data_buffer : char; data;`

rx

Description	This memory specifier allows variable to be stored in the Rx space (Register file). Note: In most of the cases, there will be enough space left for the user variables in the Rx space. However, since compiler uses Rx space for storing temporary variables, it might happen that user variables will be stored in the internal data SRAM, when writing complex programs.

Example	`// puts y in Rx space`
	`var y : char; rx;`

io

Description	This memory specifier allows user to access the I/O Memory space.

Example	`// put io_buff in io memory space`
	`var io_buff : byte; io;`

sfr

Description	This memory specifier in combination with (rx, io, data) allows user to access special function registers. It also instructs compiler to maintain same identifier in Pascal and assembly.

Example	`var io_buff : byte; io; sfr;  // put io_buff in I/O memory space`
	`var y : char; rx; sfr;             // puts y in Rx space`
	
	`var temp : byte; data; sfr; and var temp : byte; sfr; are equivalent, and put temp in Extended I/O Space.`

register

Description	If no other memory specifier is used (rx, io, sfr, code or data), the register specifer places variable in Rx space, and instructs compiler to maintain same identifier in C and assembly.
Example	`var y : char; register;`

Note: If none of the memory specifiers are used when declaring a variable, data specifier will be set as default by the compiler.

Related topics: AVR Memory Organization, Accessing individual bits, SFRs, Constants, Functions

CHAPTER 5

mikroPascal PRO for AVR Language Reference

The mikroPascal PRO for AVR Language Reference describes the syntax,semantics and implementation of mikroPascal PRO for AVR Language reference.

The aim of this referenceguide is to provide a more understandable description of the mikroPascal PRO for AVR language references to the user.

MIKROPASCAL PRO FOR AVR LANGUAGE REFERENCE

LEXICAL ELEMENTS OVERVIEW

The following topics provide a formal definition of the mikroPascal PRO for AVR lexical elements. They describe different categories of word-like units (tokens) recognized by mikroPascal PRO for AVR.

In the tokenizing phase of compilation, the source code file is parsed (i.e. broken down) into tokens and whitespace. The tokens in mikroPascal PRO for AVR are derived from a series of operations performed on your programs by the compiler.

Whitespace

Whitespace is a collective name given to spaces (blanks), horizontal and vertical tabs, newline characters and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, two sequences

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

and

```
var
i : char;

    j : word;
```

are lexically equivalent and parse identically to give nine tokens:

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

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 a part of the string). For example,
Whitespace in Strings

```
some_string := 'mikro foo';
```

parses into four tokens, including a single string literal token:

```
some_string
:=
'mikro foo'
;
```

Comments

Comments are pieces of a 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 which are enclosed with braces or (* and *):

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

(* Comment can be
   written in this way too. *)
```

or single-line comments:

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

Nested comments

mikroPascal PRO for AVR doesn't allow nested comments. The attempt to nest a comment like this

```
{ i { identifier } : word; }
```

fails, because the scope of the first open brace "{ " ends at the first closed brace "} ". This gives us

```
: word; }
```

which would generate a syntax error.

Tokens

Token is the smallest element of the Pascal program that compiler can recognize. 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 PRO for AVR recognizes the following kinds of tokens:

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

Token Extraction Example

Here is an example of token extraction. Take a look at the following example code sequence:

```
end_flag := 0;
```

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

The compiler would parse it as the following four tokens:

```
end_flag      // variable identifier
:=            // assignment operator
0             // literal
;             // statement terminator
```

Note that `:=` parses as one token (the longest token possible), not as token `:` followed by token `=`.

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 format used in the source code.

Integer Literals

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

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

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

The percent-sign prefix (`%`) indicates a binary numeral (for example, `%01010000`).

Here are some examples:

```
11          // decimal literal
$11         // hex literal, equals decimal 17
0x11        // hex literal, equals decimal 17
%11         // binary literal, equals decimal 3
```

The allowed range of values is imposed by the largest data type in mikroPascal PRO for AVR – `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 decimal fraction (but not both).

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

mikroPascal PRO for AVR limits floating-point constants to range ±1.17549435082 * 10-38 .. ±6.80564774407 * 1038.

Here are some examples:

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

Character Literals

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

Character literal can be assigned to variables of the 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 PRO for AVR.

String Literals

String literal is a sequence of characters from the extended ASCII character set, written in one line and 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 a number of characters it consists of. String is stored internally as the given sequence of characters plus a final null character. This null character is introduced to terminate the string, it does not count against the string’s total length.

String literal with nothing in between the apostrophes (null string) is stored as a single null character.

You can assign string literal to a string variable or to an array of char.

Here are several string literals:

```
'Hello world!'              // message, 12 chars long
'Temperature is stable'     // message, 21 chars long
'  '                        // two spaces, 2 chars long
'C'                         // letter, 1 char long
''                          // null string, 0 chars long
```

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

Keywords

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

Beside standard Pascal keywords, all relevant SFRs are defined as global variables and represent reserved words that cannot be redefined (for example: W0, TMR1, T1CON, 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 Pascal:

- absolute
- abstract
- and
- array
- as
- asm
- assembler
- at
- automated
- bdata
- begin
- bit
- case
- cdecl
- class
- code
- compact
- const
- constructor
- contains
- data
- default
- deprecated
- destructor
- dispid
- dispinterface
- div
- do
- downto
- dynamic
- end
- except
- export
- exports
- external
- far
- file
- final
- finalization
- finally
- for
- forward
- goto
- helper
- idata
- if
- ilevel
- implementation
- implements
- in
- index
- inherited
- initialization
- inline
- interface
- is
- label
- library
- message
- mod
- name
- near
- nil
- nodefault
- not
- object
- of
- on
- operator
- org
- out
- overload
- override
- package
- packed
- pascal
- pdata
- platform
- private
- procedure
- program
- property
- protected
- public
- published
- raise
- read
- readonly
- record
- register
- reintroduce
- repeat
- requires
- safecall
- sbit
- sealed
- set
- shl
- shr
- small
- stdcall
- stored
- string
- threadvar
- to
- try
- type
- unit
- until
- uses
- var
- virtual
- volatile
- while
- with
- write
- writeonly
- xdata
- xor

Also, mikroPascal PRO for AVR includes a number of predefined identifiers used in libraries. You can replace them by your own definitions, if you plan to develop your own libraries. For more information, see mikroPascal PRO for AVR 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 (don't get confused about the meaning of object in object-oriented programming).

Identifiers can contain the letters a to z and A to Z, underscore character "_", and digits from 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 are an equivalent identifier.

Uniqueness and Scope

Although identifier names are arbitrary (according to the stated rules), if the same name is used for more than one identifier within the same scope then error arises. Duplicated names are illegal within same scope. For more information, refer to Scope and Visibility.

Identifier Examples

Here are some valid identifiers:

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

and here are some invalid identifiers:

```
7temp        // NO -- cannot begin with a numeral
%higher      // NO -- cannot contain special characters
xor          // NO -- cannot match reserved word
j23.07.04    // NO -- cannot contain special characters (dot)
```

PUNCTUATORS

The mikroPascal punctuators (also known as separators) are:

- [ ] – Brackets
- ( ) – Parentheses
- , – Comma
- ; – Semicolon
- : – Colon
- . – 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 arguments
function func2(n : word);   //
```

Function declaration with parametersFor more information, refer to Operators Precedence and Associativity, Expressions and Functions and Procedures.

Comma

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

Semicolon (;) is a statement terminator. Every statement in Pascal must be terminated with a semicolon. The exceptions are: the last (outer most) end statement in the program which is terminated with a dot and the last statement before end which doesn't need to be terminated with a semicolon.

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;
```

For more information, refer to Labels.

Dot

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

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

For more information, refer to Records.

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

PROGRAM ORGANIZATION

Pascal imposes quite 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 Scope and Visibility.

Organization of Main Unit

Basically, the 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, the compiler may not be able to comprehend the program correctly.

When writing code, follow the model presented below. The main unit should look like this:

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

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

{ constants declarations }
const ...

{ types declarations }
type ...

{ variables declarations }
var  Name[ , Name2...] : [^]type; [absolute 0x123;] [external;]
[volatile;] [register;] [sfr;]

{ labels declarations }
label ...

{ procedures declarations }
procedure procedure_name(parameter_list);
  { local declarations }
  begin
    ...
  end;

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

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

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

Organization of Other Units

Units other than main start with the keyword unit. Implementation section starts with the keyword implementation. Follow the model presented below:

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

//*******************************************************
//* Interface (globals):
//*******************************************************

{ constants declarations }
const ...

{ types declarations }
type ...

{ variables declarations }
var Name[, Name2...] : [^]type; [absolute 0x123;] [external;]
[volatile;] [register;] [sfr;]

{ procedures prototypes }
procedure procedure_name([var] [const] ParamName : [^]type; [var]
[const] ParamName2, ParamName3 : [^]type);

{ functions prototypes }
function function_name([var] [const] ParamName : [^]type; [var]
[const] ParamName2, ParamName3 : [^]type) : [^]type;

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

implementation

{ constants declarations }
const ...

{ types declarations }
type ...
```

```
{ variables declarations }
var Name[ , Name2...] : [ ^] type; [ absolute 0x123;] [ external;]
[ volatile;] [ register;] [ sfr;]

{ labels declarations }
label ...

{  procedures declarations }
procedure procedure_name([ var] [ const] ParamName : [ ^] type; [ var]
[ const] ParamName2, ParamName3 : [ ^] type); [ ilevel 0x123;] [ over-
load;] [ forward;]
  {  local declarations }
  begin
    ...
  end;

{ functions declarations }
function function_name([ var] [ const] ParamName : [ ^] type; [ var]
[ const] ParamName2, ParamName3 : [ ^] type) : [ ^] type; [ ilevel 0x123;]
[ overload;] [ forward;]
  { local declarations }
  begin
    ...
  end;

end.
```

Note: constants, types and variables used in the `implementation` section are inaccessible to other units. This feature is not applied to the procedures and functions in the current version, but it will be added to the future ones.

Note: Functions and procedures must have the same declarations in the interface and implementation section. Otherwise, compiler will report an error.

SCOPE AND VISIBILITY

Scope

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

Place of declaration	Scope
Identifier is declared in the declaration of a program, function, or procedure	Scope extends from the point where it is declared to the end of the current block, including all blocks enclosed within that scope. Identifiers in the outermost scope (file scope) of the main unit are referred to as globals, while other identifiers are locals.
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.
Identifier is declared in the implementation section of a unit, but not within the block of 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 to the identifier's associated object can be made.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier, i.e. 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 PRO for AVR, each project consists of a single project file and one or more unit files. Project file, with extension .mppav contains information about the project, while unit files, with extension .mpas, contain the actual source code.

Units allow you to:

- break large programs into encapsulated parts 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. In order to build a project, the compiler needs either a source file or a compiled unit file (.mcl file) for each unit.

Uses Clause

mikroPascal PRO for AVR includes units by means of the 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;
```

For the given unit name, the compiler will check for the presence of `.mcl` and `.mpas` files, in order specified by the search paths.

- If both `.mpas` and `.mcl` files are found, the compiler will check their dates and include the newer one in the project. If the `.mpas` file is newer than `.mcl`, a new library will be written over the old one;
- If only `.mpas` file is found, the 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, the compiler will include it as it is found;
- If none found, the compiler will issue a “File not found” warning.

Main Unit

Every project in mikroPascal PRO for AVR requires a single main unit file. The main unit file 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 the Project Wizard, the Code Editor will display a new main unit. It contains the bare-bones of the Pascal program:

```
program MyProject;

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

Nothing should precede the keyword program except comments. After the program name, you can optionally place the uses clause.

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

Other Units

Units 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, labels and types) 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 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 a variable.

Variables are declared in the declaration part of the file or routine — each variable needs to be declared before being 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. Syntax for variable declaration is:

```
var identifier_list : type;
```

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

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

Variables and AVR

Every declared variable consumes part of RAM. Data type of variable determines not only allowed range of values, but also the space variable occupies in RAM. Bear in mind that operations using different types of variables take different time to be completed. mikroPascal PRO for AVR recycles local variable memory space – local variables declared in different functions and procedures share the same memory space, if possible.

There is no need to declare SFRs explicitly, as mikroPascal PRO for AVR automatically declares relevant registers as global variables of `volatile word` see SFR for details.

Constants

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

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

```
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. `type` is optional and in the absence of type, the compiler assumes the “smallest” of all types that can accommodate `value`.

Note: You cannot omit type when 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 a label and colon like this:

```
label_identifier : statement
```

Before marking a statement, you must declare a 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;
```

Note: label should be followed by end of line (CR) otherwise compiler will report an error:

```
label loop;
...
loop: Beep; // compiler will report an error
loop: // compiler will report an error
```

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. When executed, a function returns a value while procedure does not.

mikroPascal PRO for AVR does not support inline routines.

Functions

A function is declared like this:

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

`function_name` represents a function's name and can be any valid identifier. `return_type` is a type of return value and can be any simple type. Within parentheses, `parameter_list` is a formal parameter list very similar to variable declaration. In Pascal, parameters are always passed to a function by the value — to pass an argument by 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.

Calling a 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 a function call, all formal parameters are created as local objects initialized by values of actual arguments. Upon return from a function, a temporary object is created in the place of the call and it is initialized by the value of the function result. This means that function call as an operand in complex expression is treated as the function result.

In standard Pascal, a `function_name` is automatically created local variable that can be used for returning a value of a function. mikroPascal PRO for AVR also 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. If the return value of a function is not defined the compiler will report an error.

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

Example

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 3^{12} for example:

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

Procedures

Procedure is declared like this:

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

procedure_name represents a procedure's name and can be any valid identifier. Within parentheses, parameter_list is a formal parameter list very similar to variable declaration. In Pascal, parameters are always passed to a procedure by the value — to pass an argument by 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.

Calling a 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 the values of actual arguments.

Procedure call is a self-contained statement.

Example

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;
```

A function can return a complex type. Follow the example bellow to learn how to declare and use a function which returns a complex type.

Example:

This example shows how to declare a function which returns a complex type.

```
program Example;

type TCircle = record  // Record
    CenterX, CenterY: word;
    Radius: byte;
end;

var MyCircle: TCircle; // Global variable

function DefineCircle(x, y: word; r: byte): TCircle; // DefineCircle
function returns a Record

begin
  result.CenterX := x;
  result.CenterY := y;
  result.Radius  := r;
end;

begin
  MyCircle := DefineCircle(100, 200, 30);                        //
Get a Record via function call
   MyCircle.CenterX := DefineCircle(100, 200, 30).CenterX + 20; //
Access a Record field via function call
  //                    |------------------------|  |-----|
  //                       |                           |
  //                     Function returns TCircle     Access to one
field of TCircle
end.
```

Forward declaration

A function can be declared without having it followed by it's implementation, by having it followed by the forward procedure. The effective implementation of that function must follow later in the unit. The function can be used after a forward declaration as if it had been implemented already. The following is an example of a forward declaration:

```
program Volume;

var Volume : word;

function First(a, b : word) : word; forward;

function Second(c : word) : word;
var tmp : word;
begin
  tmp := First(2, 3);
  result := tmp * c;
end;

function First(a, b : word) : word;
begin
  result := a * b;
end;

begin
  Volume := Second(4);
end.
```

TYPES

Pascal is 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 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 PRO for AVR 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
- records

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. Basic memory unit in mikroPascal PRO for AVR has 16 bits.

Here is an overview of simple types in mikroPascal PRO for AVR:

Type	Size	Range
`byte, char`	8–bit	0 .. 255
`short`	8–bit	-127 .. 128
`word`	16–bit	0 .. 65535
`integer`	16–bit	-32768 .. 32767
`dword`	32–bit	0 .. 4294967295
`longint`	32–bit	-2147483648 .. 2147483647
`real`	32–bit	±1.17549435082 * 10-38 .. ±6.80564774407 * 1038
`bit`	1–bit	0 or 1
`sbit`	1–bit	0 or 1

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 Declaration

Array types are denoted by constructions in the following form:

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

Each of the elements of an array is numbered from `index_start` through `index_end`. The 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 number 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. The opposite is possible, when the trailing “excess” elements are assigned zeroes.

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

Multi-dimensional Arrays

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

```
m : array[5] of array[10] of byte;    // 2-dimensional array of size
5x10
```

A variable m is an array of 5 elements, which in turn are arrays of 10 byte each. Thus, we have a matrix of 5x10 elements where the first element is `m[0][0]` and last one is `m[4][9]`. The first element of the 4th row would be `m[3][0]`.

STRINGS

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

```
string_name : string[ length]
```

The specifier length is a number of characters the string consists of. String is stored internally as the given sequence of characters plus a final null character which is introduced to terminate the string. It does not count against the 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 shorter or of equal length than the one on the right side. 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 will cause an unpredictable behavior.

String Concatenating

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

Here is an example:

```
var msg      : string[20];
    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"
          res_txt     ;       // Result of ADC


  //...
```

Note: In current version plus operator for concatenating strings will accept at most two operands.

Note

mikroPascal PRO for AVR 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 carat prefix (^) before type. For example, in order to create a pointer to an integer, write:

```
^integer;
```

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

```
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.

Pointers to program memory space are declared using the keyword const:

```
program const_ptr;

// constant array will be stored in program memory
const b_array: array[5] of byte = (1,2,3,4,5);

const ptr: ^byte;      // ptr is pointer to program memory space

begin
  ptr   := @b_array;  // ptr now points to b_array[0]
  P0  := ptr^;
  ptr   := ptr + 3;    // ptr now points to b_array[3]
  P0  := ptr^;
end.
```

Function Pointers

Function pointers are allowed in mikroPascal PRO for AVR. The example shows how to define and use a function pointer:

Example:

Example demonstrates the usage of function pointers. It is shown how to declare a procedural type, a pointer to function and finally how to call a function via pointer.

```
 program Example;

 type TMyFunctionType = function (param1, param2: byte; param3: word)
: word;  // First, define the procedural type
             var              MyPtr:             ^TMyFunctionType;
// This is a pointer to previously defined type
     Sample: word;

 function  Func1(p1,  p2:  byte;  p3:  word):  word;//  Now,  define  few
functions which will be pointed to. Make sure that parameters match
the type definition
 begin
     result := p1 and p2 or p3;                 // return something
 end;

 function Func2(abc: byte; def: byte; ghi: word): word;    // Another
function of the same kind. Make sure that parameters match the type
definition
 begin
   result := abc * def + ghi;                   // return something
 end;

 function Func3(first, yellow: byte; monday: word): word// Yet anoth-
er function. Make sure that parameters match the type definition
 begin
   result := monday - yellow - first;           // return something
 end;

 //  main program:
 begin
    MyPtr  := @Func1;                    ,// MyPtr now points to Func1
    Sample := MyPtr^(1, 2, 3);            // Perform function call via
pointer, call Func1, the return value is 3
    MyPtr  := @Func2;                     // MyPtr now points to Func2
    Sample := MyPtr^(1, 2, 3);            // Perform function call via
pointer, call Func2, the return value is 5
    MyPtr  := @Func3;                     // MyPtr now points to Func3
    Sample := MyPtr^(1, 2, 3);            // Perform function call via
pointer, call Func3, the return value is 0
 end.
```

@ Operator

The `@` operator returns the address of a variable or routine, i.e. `@` constructs a pointer to its operand. The following rules are applied 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 (the result is of `longint`).

Records

A record (analogous to a structure in some languages) represents a heterogeneous set of elements. Each element is called a field. The declaration of the 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 have to worry about naming conflicts between field identifiers and other variables.

Note: In mikroPascal PRO for AVR, 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 declare 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 have declared 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 variable of one type to a variable of another type is typecasting. mikroPascal PRO for AVR supports both implicit and explicit conversions for built-in types.

Implicit 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 type to more complex type taking the following steps:

```
byte/char     ⇢  word
short         ⇢  integer
short         ⇢  longint
integer       ⇢  longint
integer       ⇢  real
```

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). For example:

```
var a : byte; b : word;
...
a := $FF;
b := a;  // a is promoted to word, b becomes $00FF
```

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, i.e. if the result fits in destination range.

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

Explicit Conversion

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

```
word(b) := a;    // Compiler will report an error
```

Conversions Examples

Here is an example of conversion:

```
var a, b, c : byte;
          d : word;
...
a := 241;
b := 128;

c   := a + b;              // equals 113
c   := word(a + b);        // equals 113
d   := a + b;              // equals 369
```

OPERATORS

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

There are four types of operators in mikroPascal PRO for AVR:

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

Operators Precedence and Associativity

There are 4 precedence categories in mikroPascal PRO for AVR. 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 + -`	←
3	2	`* / div mod and shl shr`	→
2	2	`+ - or xor`	→
1	2	`= <> < > <= >=`	→

Arithmetic Operators

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results. Since the char operators are technically bytes, they can be also used as unsigned operands in arithmetic operations.

All arithmetic operators associate from left to right.

Operator	Operation	Operands	Result
`+`	addition	`byte, short, word, integer, longint, dword, real`	`byte, short,word, integer, longint, dword, real`
`-`	subtraction	`byte, short, word, integer, longint, dword, real`	`byte, short,word, integer, longint, dword, real`
`*`	multiplication	`byte, short, word, integer, longint, dword, real`	`word, integer, longint, dword, real`
`/`	division, floating-point	`byte, short, word, integer, longint, dword, real`	`real`
`div`	division, rounds down to nearest integer	`byte, short, word, integer, longint,dword`	`byte, short, word, integer, longint, dword`
`mod`	modulus, returns the remainder of integer division (cannot be used with floatin points)	`byte, short, word, integer, longint, dword`	`byte, short, word, integer, longint, dword`

Division by Zero

If 0 (zero) is used explicitly as the second operand (`i.e. x div 0`), the 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), the result will be the maximum integer (i.e `255`, if the result is byte type; `65536`, if the result is `word` type, etc.).

Unary Arithmetic Operators

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 data.

For example:

```
b := -a;
```

Relational Operators

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

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

All relational operators associate from left to right.

Relational Operators in Expressions

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

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

Bitwise Operators

Use bitwise operators to modify 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.

Bitwise Operators Overview

Operator	Operation
`and`	bitwise AND; compares pairs of bits and generates a 1 result if both bits are 1, otherwise it returns 0
`or`	bitwise (inclusive) OR; compares pairs of bits and generates a 1 result if either or both bits are 1, otherwise it returns 0
`xor`	bitwise exclusive OR (XOR); compares pairs of bits and generates a 1 result if the bits are complementary, otherwise it returns 0
`not`	bitwise complement (unary); inverts each bit
`shl`	bitwise shift left; moves the bits to the left, discards the far left bit and assigns 0 to the right most bit.
`shr`	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends

Logical Operations on Bit Level

and	**0**	**1**
0	0	0
1	0	1

or	**0**	**1**
0	0	1
1	1	1

xor	**0**	**1**
0	0	1
1	1	0

not	0	1
	1	0

Bitwise operators and, or, and xor perform logical operations on the appropriate pairs of bits of their operands. not operator 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 a number is converted from less complex to more complex data type, the upper bytes are filled with zeroes. If a number is converted from more complex to less complex data type, the data is simply truncated (the 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 zeroes if sign bit is 0 (number is positive). If number is converted from more complex data type to less complex, data is simply truncated (upper bytes are lost).

For example:

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

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

Boolean Operators

Although mikroPascal PRO for AVR 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 (XOR)
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, constants, variables and function calls. More complex expressions can be created from primary expressions by using operators. 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 which depend on the operators in use, presence of parentheses and 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 PRO for AVR.

STATEMENTS

Statements define algorithmic actions within a program. Each statement needs to be terminated with 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 most simple statements are assignments, procedure calls and jump statements. These can be combined to form loops, branches and other structured statements.

Refer to:

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

Assignment Statements

Assignment statements have the form:

```
variable := expression;
```

The statement evaluates expression and assigns its value to `variable`. All the rules of implicit conversion are applied. `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. Also note that, although similar, the construction is not related to the declaration of constants.

Compound Statements (Blocks)

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 is allowed to be used when Pascal syntax requires a single statement. Blocks can be nested up to the limits of memory.

For example, the `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;
    i := i + 1;
  end;
```

Conditional Statements

Conditional or selection statements select one of alternative courses of action by testing certain values. There are two types of selection statements:

- if
- case

If Statement

Use if to implement a conditional statement. The syntax of if statement has the form:

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

If expression evaluates to true then statement1 executes. If expression is false then statement2 executes. The expression must convert to a boolean type; 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

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

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

The compiler treats the construction in this way:

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

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

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

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. The syntax of the `case` statement is:

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

`selector` is an expression which should evaluate as integral value. `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. Afterwards the case statement compares it against all available values. If the match is found, the `statement` following the match evaluates, and the case statement terminates. In case there are multiple matches, the first matching statement will be executed. If none of values matches selector, then `default_statement` in the else clause (if there is some) is executed.

Here's a simple example of the case statement:

```
case operator of
  '*' : result := n1 * n2;
  '/' : 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
  0:         opmode := 0;
  1,2,3,4:  opmode := 1;
  5,6,7:    opmode := 2;
end;
```

In mikroPascal PRO for AVR, values in the case statement can be variables too:

```
case byte_variable of

  byte_var1: opmode := 0;  // this will be compiled correctly

  byte_var2:
             opmode := 1;  // avoid this case, compiler will parse
                    // a variable followed by colon sign as label

  byte_var3: //         adding a comment solves the parsing problem
             opmode := 2;
end;
```

Nested Case statement

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

ITERATION STATEMENTS

Iteration statements let you loop a set of statements. There are three forms of iteration statements in mikroPascal PRO for AVR:

- for
- while
- repeat

You can use the statements `break` and continue to control the flow of a loop statement. 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. The syntax of the for statement is:

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

`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 `initial_value` and will increment (or decrement) until it reaches `final_value`. With each iteration, statement will be executed.

`initial_value` and `final_value` should be expressions compatible with `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 the for statement:

```
s := 0;
for i := 0 to n-1 do
  s := s + a[ i] * b[ i] ;
```

Endless Loop

The for statement results in an endless loop if final_value equals or exceeds the range of the 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. The syntax of the while statement is:

```
while expression do statement
```

`statement` is executed repeatedly as long as `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;
```

Probably the easiest way to create an endless loop is to use the statement:

```
while TRUE do ...;
```

Repeat Statement

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

```
repeat statement until expression
```

`statement` is executed repeatedly as long as 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 four such statements in mikroPascal PRO for AVR:

- break
- continue
- exit
- goto

Break and Continue Statements

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:

```
Lcd_Out(1,1,'Insert CF card');

// Wait for CF card to be plugged; refresh every second
while TRUE do
begin
  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 keyword for
- `continue` statement in while loop moves program counter to the line with loop con dition (top of the loop),
- `continue` statement in repeat loop moves program counter to the line with loop condition (bottom of the loop).

```
// continue jumps here
for i := ... do
  begin
    ...
    continue;
    ...
  end;

// continue jumps here
while condition do
  begin
    ...
    continue;
    ...
  end;
  begin
    ...
    continue;
    ...
  // continue jumps here
until condition;
```

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.

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
```

asm Statement

mikroPascal PRO for AVR allows embedding assembly in the source code by means of the 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;
```

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, the linker will issue an error. This is not applied to predefined globals such as P0.

For example, the following code will not be compiled because the linker won't be able to recognize the variable myvar:

```
program test;
var myvar : word;
begin

  asm
    MOV      #10, W0
    MOV      W0, _myvar
  end;
end.
```

Adding the following line (or similar one) above the asm block would let linker know that variable is used:

```
myvar := 20;
```

DIRECTIVES

Directives are words of special significance which provide additional functionality regarding compilation and output.

The following directives are available for use:

- Compiler directives for conditional compilation,
- Linker directives for object distribution in memory.

Compiler Directives

mikroPascal PRO for AVR treats comments beginning with a `“$”` immediately following an opening brace as a compiler directive; for example, `{ $ELSE}` . The compiler directives are not case sensitive.

You can use a 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 have begun.

Directives $DEFINE and $UNDEFINE

Use directive `$DEFINE` to define a conditional compiler constant (`“flag”`). You can use any identifier for a flag, with no limitations. No conflicts with program identifiers are possible because the 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 the `$IFDEF` directive. `$IFDEF` tests whether a flag is currently defined or not, i.e. whether a previous `$DEFINE` directive has been processed for that flag and is still in force.

Directive `$IFDEF` is terminated with the `$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. `$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}
```

Include Directive $I

The $I parameter directive instructs mikroPascal PRO for AVR to include the named text file in the compilation. In effect, the file is inserted in the compiled text right after the {$I filename} directive. If filename does not specify a directory path, then, in addition to searching for the file in the same directory as the current unit, mikroPascal PRO for AVR will search for file in order specified by the search paths.

To specify a filename that includes a space, surround the file name with quotation marks:{$I "My file"}.

There is one restriction to the use of include files: An include file can't be specified in the middle of a statement part. In fact, all statements between the begin and end of a statement part must exist in the same source file.

Predefined Flags

The compiler sets directives upon completion of project settings, so the user doesn't need to define certain flags.
Here is an example:

```
{$IFDEF ATMEGA16}    // If ATmega16 MCU is selected
{$IFDEF ATMEGA128}   // IF ATmega128 MCU is selected
```

In some future releases of the compiler, the JTAG flag will be added also.

See also predefined project level defines.

Linker Directives

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

Note: You must specify an even address when using the linker directives.

Directive absolute

Directive `absolute` specifies the starting address in RAM for a variable. If the variable spans more than 1 word (16-bit), the higher words will be stored at the consecutive locations.

Directive absolute is appended to the declaration of a variable:

```
var x : word; absolute $32;
// Variable x will occupy 1 word (16 bits) at address $32

    y : longint; absolute $34;
// Variable y will occupy 2 words at addresses $34 and $36
```

Be careful when using the absolute directive because you may overlap two variables by accident. For example:

```
var  i : word; absolute $42;
// Variable i will occupy 1 word at address $42;

     jj : longint; absolute $40;
// Variable will occupy 2 words at $40 and $42; thus,
// changing i changes jj at the same time and vice versa
```

Note: You must specify an even address when using the absolute directive.

Directive org

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

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

org directive can be used with main routine too. For example:

```
program Led_Blinking;

procedure some_proc();
begin
  ...
end;


org 0x800;              // main procedure starts at 0x800
begin
  DDRB := 0xFF;

  while TRUE do
    begin
      PORTB := 0x00;
      Delay_ms(500);
      PORTB := 0xFF;
      Delay_ms(500);
    end;
end.
```

Note: You must specify an even address when using the org directive.

CHAPTER 6

mikroPascal PRO for AVR Libraries

mikroPascal PRO for AVR provides a set of libraries which simplify the initialization and use of AVR compliant MCUs and their modules:

Use Library manager to include mikroPascal PRO for AVR Libraries in you project.

Hardware AVR-specific Libraries

- ADC Library
- CANSPI Library
- Compact Flash Library
- EEPROM Library
- Flash Memory Library
- Graphic Lcd Library
- Keypad Library
- Lcd Library
- Manchester Code Library
- Multi Media Card library
- OneWire Library
- Port Expander Library
- PS/2 Library
- PWM Library
- PWM 16 bit Library
- RS-485 Library
- Software I2C Library
- Software SPI Library
- Software UART Library
- Sound Library
- SPI Library
- SPI Ethernet Library
- SPI Graphic Lcd Library
- SPI Lcd Library
- SPI Lcd8 Library
- SPI T6963C Graphic Lcd Library
- T6963C Graphic Lcd Library
- TWI Library
- UART Library

Miscellaneous Libraries

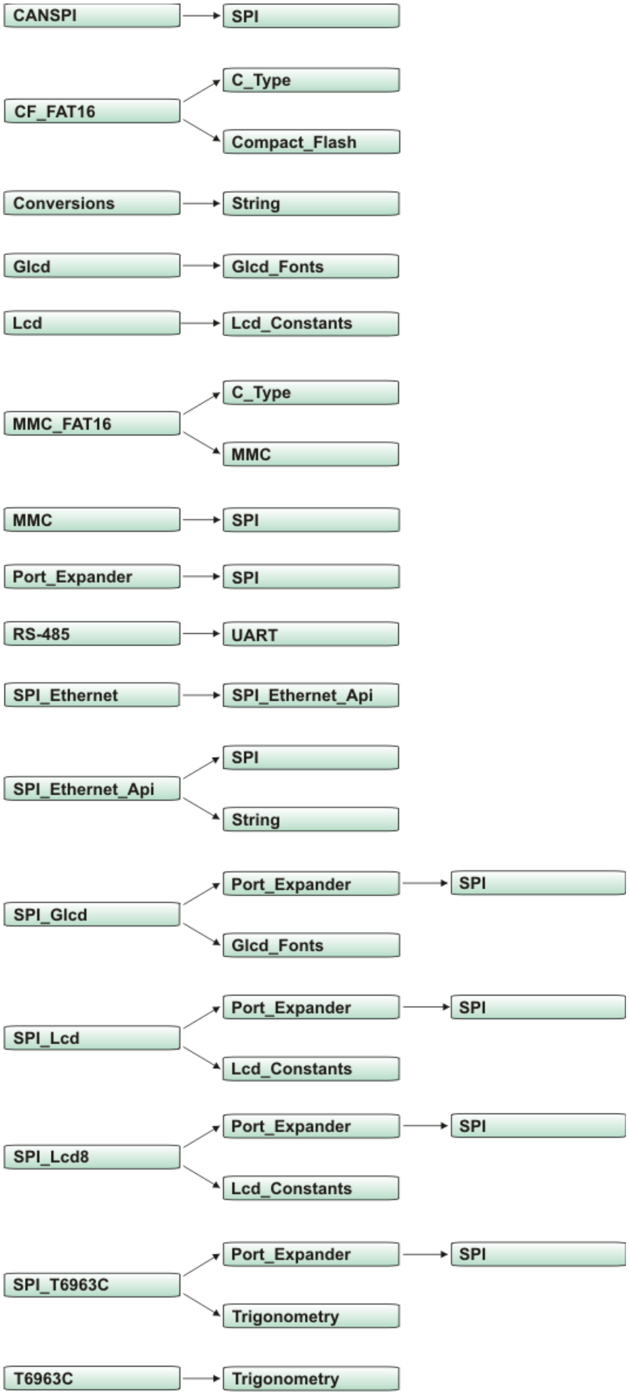
- Button Library
- Conversions Library
- Math Library
- String Library
- Time Library
- Trigonometry Library

See also Built-in Routines.

LIBRARY DEPENDENCIES

Certain libraries use (depend on) function and/or variables, constants defined in other libraries.
Image below shows clear representation about these dependencies.

For example, SPI_Glcd uses Glcd_Fonts and Port_Expander library which uses SPI library.
This means that if you check SPI_Glcd library in Library manager, all libraries on which it depends will be checked too.



Related topics: Library manager, AVR Libraries

ADC LIBRARY

ADC (Analog to Digital Converter) module is available with a number of AVR micros. Library function `ADC_Read` is included to provide you comfortable work with the module in single-ended mode.

ADC_Read

Prototype	`function ADC_Read( channel : byte ) : word;`
Returns	10-bit or 12-bit (MCU dependent) unsigned value from the specified `channel`.
Description	Initializes AVR 's internal ADC module to work with XTAL frequency prescaled by 128. Clock determines the time period necessary for performing A/D conversion. Parameter `channel` represents the channel from which the analog value is to be acquired. Refer to the appropriate datasheet for channel-to-pin mapping.
Requires	Nothing.

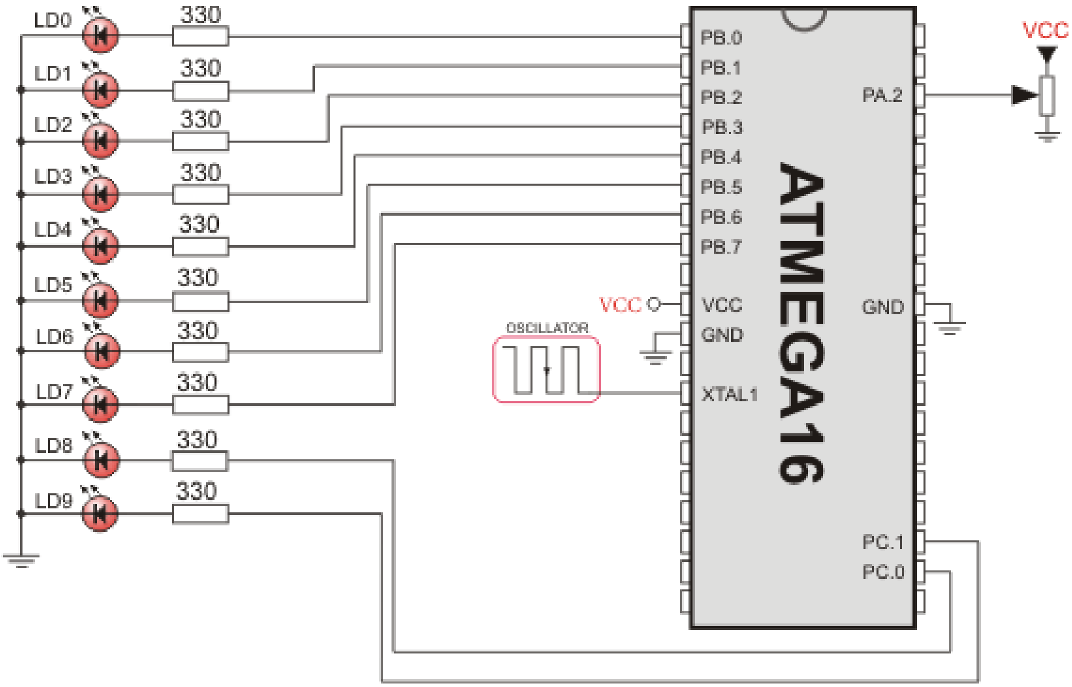
Example	`var tmp : word;`
	`...`
	`tmp := ADC_Read(2);   // Read analog value from channel 2`
	`Library Example`

Library Example

This example code reads analog value from channel 2 and displays it on PORTB and PORTC.

```
program ADC_on_LEDs;
var adc_rd : word;

begin

  DDRB := 0xFF;                 // Set PORTB as output
  DDRC := 0xFF;                 // Set PORTC as output

  while TRUE do
    begin
      adc_rd := ADC_Read(2);    // get ADC value from 2nd channel
      PORTB := adc_rd;          // display adc_rd[ 7..0]
      PORTC := Hi(adc_rd);      // display adc_rd[ 9..8]
    end;
end.
```

HW Connection



ADC HW connection

CANSPI LIBRARY

The SPI module is available with a number of the AVR compliant MCUs. The mikroPascal PRO for AVR provides a library (driver) for working with mikroElektronika's CANSPI Add-on boards (with MCP2515 or MCP2510) via SPI interface.

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

Data transfer rates depend on distance. For example, 1 Mbit/s can be achieved at network lengths below 40m while 250 Kbit/s can be achieved at network lengths below 250m. The greater distance the lower maximum bitrate that can be achieved. The lowest bitrate defined by the standard is 200Kbit/s. Cables used are shielded twisted pairs.

CAN supports two message formats:

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

Note:

- Consult the CAN standard about CAN bus termination resistance.
- An effective CANSPI communication speed depends on SPI and certainly is slow er than “real” CAN.
- CANSPI module refers to mikroElektronika's CANSPI Add-on board connected to SPI module of MCU.
- Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

External dependencies of CANSPI Library

The following variables must be defined in all projects using Sound Library:	Description:	Example :
`var CanSpi_CS : sbit; sfr; external;`	Chip Select line.	`var CanSpi_CS : sbit at PORTB.B0;`
`var CanSpi_Rst : sbit; sfr; external;`	Reset line.	`var CanSpi_Rst : sbit at PORTB.B2;`
`var CanSpi_CS_Bit_Direction : sbit; sfr; external;`	Direction of the Chip Select pin.	`var CanSpi_CS_Bit_Direction : sbit at DDRB.B0;`
`var CanSpi_Rst_Bit_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var CanSpi_Rst_Bit_Direction : sbit at DDRB.B2;`

Library Routines

- CANSPISetOperationMode
- CANSPIGetOperationMode
- CANSPIInitialize
- CANSPISetBaudRate
- CANSPISetMask
- CANSPISetFilter
- CANSPIread
- CANSPIWrite

The following routines are for an internal use by the library only:

- RegsToCANSPIID
- CANSPIIDToRegs

Be sure to check CANSPI constants necessary for using some of the functions.

CANSPISetOperationMode

Prototype	`procedure CANSPISetOperationMode(mode: byte; WAIT: byte);`
Returns	Nothing.
Description	Sets the CANSPI module to requested mode. Parameters : - `mode`: CANSPI module operation mode. Valid values: `CANSPI_OP_MODE` constants (see CANSPI constants). - `WAIT`: CANSPI mode switching verification request. If `WAIT = 0`, the call is non blocking. The function does not verify if the CANSPI module is switched to requested mode or not. Caller must use `CANSPIGetOperationMode` to verify correct opertion mode before performing mode specific operation. If `WAIT != 0`, the call is blocking – the function won't "return" until the requested mode is set.
Requires	The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set the CANSPI module into configuration mode (wait inside CANSPISetOperationMode until this mode is set)`
	`CANSPISetOperationMode(CANSPI_MODE_CONFIG, 0xFF);`

CANSPIGetOperationMode

Prototype	`function CANSPIGetOperationMode(): byte;`
Returns	Current operation mode.
Description	The function returns current operation mode of the CANSPI module. Check `CANSPI_OP_MODE` constants (see CANSPI constants) or device datasheet for operation mode codes.
Requires	The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// check whether the CANSPI module is in Normal mode and if it is do something.`
	`if (CANSPIGetOperationMode() = CANSPI_MODE_NORMAL) then`
	`begin`
	`   ...`
	`end;`

CANSPIInitialize

Prototype	`procedure CANSPIInitialize(SJW: byte; BRP: byte; PHSEG1: byte; PHSEG2: byte; PROPSEG: byte; CAN_CONFIG_FLAGS: byte);`
Returns	Nothing.
Description	Initializes the CANSPI module. Stand-Alone CAN controller in the CANSPI module is set to: - Disable CAN capture - Continue CAN operation in Idle mode - Do not abort pending transmissions - Fcan clock: 4*Tcy (Fosc) - Baud rate is set according to given parameters - CAN mode: Normal - Filter and mask registers IDs are set to zero - Filter and mask message frame type is set according to `CAN_CONFIG_FLAGS` value `SAM, SEG2PHTS, WAKFIL` and `DBEN` bits are set according to`CAN_CONFIG_FLAGS` value. Parameters: - `SJW` as defined in CAN controller's datasheet - `BRP` as defined in CAN controller's datasheet - `PHSEG1` as defined in CAN controller's datasheet - `PHSEG2` as defined in CAN controller's datasheet - `PROPSEG` as defined in CAN controller's datasheet - `CAN_CONFIG_FLAGS` is formed from predefined constants (see CANSPI constants)

Requires

Global variables :

- CanSpi_CS: Chip Select line
- CanSpi_Rst: Reset line
- CanSpi_CS_Bit_Direction: Direction of the Chip Select pin
- CanSpi_Rst_Bit_Direction: Direction of the Reset pin

must be defined before using this function.

The CANSPI routines are supported only by MCUs with the SPI module.

The SPI module needs to be initialized. See the SPI1_Init and SPI1_Init_Advanced routines.

MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example

```
// CANSPI module connections
var CanSpi_CS     : sbit at  PORTB.B0;
    CanSpi_CS_Direction : sbit at DDRB.B0;
    CanSpi_Rst : sbit at PORTB.B2;
    CanSpi_Rst_Direction : sbit at DDRB.B2;
// End CANSPI module connections

var Can_Init_Flags: byte;
  ...
  Can_Init_Flags := CAN_CONFIG_SAMPLE_THRICE and  // form value to
be used
                     CAN_CONFIG_PHSEG2_PRG_ON and  // with
CANSPIInitialize
                     CAN_CONFIG_XTD_MSG       and
                     CAN_CONFIG_DBL_BUFFER_ON and
                     CAN_CONFIG_VALID_XTD_MSG;
  ...
  Spi_Rd_Ptr := @SPI1_Read;     // Pass pointer to SPI Read func-
tion of used SPI module
  SPI1_Init();                            // initialize SPI module
  CANSPIInitialize(1,3,3,3,1,Can_Init_Flags);  // initialize
external CANSPI module
```

CANSPISetBaudRate

Prototype	`procedure CANSPISetBaudRate(SJW: byte; BRP: byte; PHSEG1: byte; PHSEG2: byte; PROPSEG: byte; CAN_CONFIG_FLAGS: byte);`
Returns	Nothing.
Description	Sets the CANSPI module baud rate. Due to complexity of the CAN protocol, you can not simply force a bps value. Instead, use this function when the CANSPI module is in Config mode. `SAM, SEG2PHTS` and WAKFIL bits are set according to `CAN_CONFIG_FLAGS` value. Refer to datasheet for details. Parameters: - SJW as defined in CAN controller's datasheet - BRP as defined in CAN controller's datasheet - PHSEG1 as defined in CAN controller's datasheet - PHSEG2 as defined in CAN controller's datasheet - PROPSEG as defined in CAN controller's datasheet - CAN_CONFIG_FLAGS is formed from predefined constants (see CANSPI constants)
Requires	The CANSPI module must be in Config mode, otherwise the function will be ignored. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set required baud rate and sampling rules`
	`var can_config_flags: byte;`
	`...`
	`CANSPISetOperationMode(CANSPI_MODE_CONFIG,0xFF);            // set CONFIGURATION mode (CANSPI module mast be in config mode for baud rate settings)`
	`can_config_flags := CANSPI_CONFIG_SAMPLE_THRICE and`
	`                     CANSPI_CONFIG_PHSEG2_PRG_ON and`
	`                     CANSPI_CONFIG_STD_MSG        and`
	`                     CANSPI_CONFIG_DBL_BUFFER_ON and`
	`                     CANSPI_CONFIG_VALID_XTD_MSG and`
	`                     CANSPI_CONFIG_LINE_FILTER_OFF;`
	`CANSPISetBaudRate(1, 1, 3, 3, 1, can_config_flags);`

CANSPISetMask

Prototype	`procedure CANSPISetMask(CAN_MASK: byte; val: longint; CAN_CONFIG_FLAGS: byte);`
Returns	Nothing.

Description	Configures mask for advanced filtering of messages. The parameter value is bit-adjusted to the appropriate mask registers.
	Parameters:
	- `CAN_MASK`: CANSPI module mask number. Valid values: `CANSPI_MASK` constants (see CANSPI constants)
	- `val`: mask register value
	- `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values:
	`CANSPI_CONFIG_ALL_VALID_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE and CANSPI_CONFIG_STD_MSG,`
	`CANSPI_CONFIG_MATCH_MSG_TYPE and CANSPI_CONFIG_XTD_MSG.`
	(see CANSPI constants)

Requires	The CANSPI module must be in Config mode, otherwise the function will be ignored. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set the appropriate filter mask and message type value`
	`CANSPISetOperationMode(CANSPI_MODE_CONFIG,0xFF);                // set CONFIGURATION mode (CANSPI module must be in config mode for mask settings)`
	
	`// Set all B1 mask bits to 1 (all filtered bits are relevant):`
	`// Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`// Complement will do the trick and fill it up with ones.`
	`CANSPISetMask(CANSPI_MASK_B1, -1, CANSPI_CONFIG_MATCH_MSG_TYPE and CANSPI_CONFIG_XTD_MSG);`

CANSPISetFilter

Prototype	`procedure CANSPISetFilter(CAN_FILTER: byte; val: longint; CAN_CONFIG_FLAGS: byte);`
Returns	Nothing.
Description	Configures message filter. The parameter value is bit-adjusted to the appropriate filter registers. Parameters: - `CAN_FILTER`: CANSPI module filter number. Valid values: CANSPI_FILTER constants (see CANSPI constants) - `val`: filter register value - `CAN_CONFIG_FLAGS`: selects type of message to filter. Valid values: `CANSPI_CONFIG_ALL_VALID_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE and CANSPI_CONFIG_STD_MSG`, `CANSPI_CONFIG_MATCH_MSG_TYPE and CANSPI_CONFIG_XTD_MSG`. (see CANSPI constants)
Requires	The CANSPI module must be in Config mode, otherwise the function will be ignored. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// set the appropriate filter value and message type`
	`CANSPISetOperationMode(CANSPI_MODE_CONFIG,0xFF);`
	`// set CONFIGURATION mode (CANSPI module must be in config mode for filter settings)`
	
	`// Set id of filter B1_F1 to 3:`
	`CANSPISetFilter(CANSPI_FILTER_B1_F1, 3, CANSPI_CONFIG_XTD_MSG);`

CANSPIRead

Prototype	`function CANSPIRead(var id: longint; var rd_data: array[ 8] of byte; data_len: byte; var CAN_RX_MSG_FLAGS: byte): byte;`
Returns	- `0` if nothing is received - `0xFF` if one of the Receive Buffers is full (message received)
Description	If at least one full Receive Buffer is found, it will be processed in the following way: - Message ID is retrieved and stored to location provided by the id parameter - Message data is retrieved and stored to a buffer provided by the `rd_data` parameter - Message length is retrieved and stored to location provided by the `data_len` parameter - Message flags are retrieved and stored to location provided by the `CAN_RX_MSG_FLAGS` parameter Parameters: - `id`: message identifier storage address - `rd_data`: data buffer (an array of bytes up to 8 bytes in length) - `data_len`: data length storage address. - `CAN_RX_MSG_FLAGS`: message flags storage address
Requires	The CANSPI module must be in a mode in which receiving is possible. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// check the CANSPI module for received messages. If any was received do something.`
	`var msg_rcvd, rx_flags, data_len: byte;`
	`  rd_data: array[ 8] of byte;`
	`  msg_id: longint;`
	`...`
	`CANSPISetOperationMode(CANSPI_MODE_NORMAL,0xFF);`
	`// set NORMAL mode (CANSPI module must be in mode in which receive is possible)`
	`...`
	`rx_flags := 0;`
	`// clear message flags`
	`if (msg_rcvd = CANSPIRead(msg_id, rd_data, data_len, rx_flags)`
	`begin`
	`    ...`
	`end;`

CANSPIWrite

Prototype	`function CANSPIWrite(id: longint; var wr_data: array[ 8] of byte; data_len: byte; CAN_TX_MSG_FLAGS: byte): byte;`
Returns	- `0` if all Transmit Buffers are busy - `0xFF` if at least one Transmit Buffer is available
Description	If at least one empty Transmit Buffer is found, the function sends message in the queue for transmission. Parameters: - `id`:CAN message identifier. Valid values: 11 or 29 bit values, depending on message type (standard or extended) - `wr_data`: data to be sent (an array of bytes up to 8 bytes in length) - `data_len`: data length. Valid values: 1 to 8 - `CAN_RX_MSG_FLAGS`: message flags
Requires	The CANSPI module must be in mode in which transmission is possible. See CANSPISetOperationMode. The CANSPI routines are supported only by MCUs with the SPI module. MCU has to be properly connected to mikroElektronika's CANSPI Extra Board or similar hardware. See connection example at the bottom of this page.

Example	`// send message extended CAN message with the appropriate ID and data`
	`var tx_flags: byte;`
	`  rd_data: array[ 8] of byte;`
	`  msg_id: longint;`
	`...`
	`CANSPISetOperationMode(CAN_MODE_NORMAL, 0xFF);`
	`// set NORMAL mode (CANSPI must be in mode in which transmission is possible)`
	
	`tx_flags := CANSPI_TX_PRIORITY_0 ands CANSPI_TX_XTD_FRAME;`
	`// set message flags`
	`CANSPIWrite(msg_id, rd_data, 2, tx_flags);`

CANSPI Constants

There is a number of constants predefined in the CANSPI library. You need to be familiar with them in order to be able to use the library effectively. Check the example at the end of the chapter.

CANSPI_OP_MODE

The `CANSPI_OP_MODE` constants define CANSPI operation mode. Function CANSPISetOperationMode expects one of these as it's argument:

```
const
      CANSPI_MODE_BITS   = 0xE0;   // Use this to access opmode  bits
       CANSPI_MODE_NORMAL = 0x00;
       CANSPI_MODE_SLEEP  = 0x20;
       CANSPI_MODE_LOOP   = 0x40;
       CANSPI_MODE_LISTEN = 0x60;
       CANSPI_MODE_CONFIG = 0x80;
```

CANSPI_CONFIG_FLAGS

The CANSPI_CONFIG_FLAGS constants define flags related to the CANSPI module configuration. The functions CANSPIInitialize, CANSPISetBaudRate, CANSPISetMask and CANSPISetFilter expect one of these (or a bitwise combination) as their argument:

```
const
    CANSPI_CONFIG_DEFAULT          = 0xFF;    // 11111111

    CANSPI_CONFIG_PHSEG2_PRG_BIT  = 0x01;
    CANSPI_CONFIG_PHSEG2_PRG_ON   = 0xFF;     // XXXXXXX1
    CANSPI_CONFIG_PHSEG2_PRG_OFF  = 0xFE;     // XXXXXXX0

    CANSPI_CONFIG_LINE_FILTER_BIT = 0x02;
    CANSPI_CONFIG_LINE_FILTER_ON  = 0xFF;     // XXXXXX1X
    CANSPI_CONFIG_LINE_FILTER_OFF = 0xFD;     // XXXXXX0X

    CANSPI_CONFIG_SAMPLE_BIT       = 0x04;
    CANSPI_CONFIG_SAMPLE_ONCE      = 0xFF;    // XXXXX1XX
    CANSPI_CONFIG_SAMPLE_THRICE   = 0xFB;     // XXXXX0XX

    CANSPI_CONFIG_MSG_TYPE_BIT     = 0x08;
    CANSPI_CONFIG_STD_MSG          = 0xFF;    // XXXX1XXX
    CANSPI_CONFIG_XTD_MSG          = 0xF7;    // XXXX0XXX

    CANSPI_CONFIG_DBL_BUFFER_BIT  = 0x10;
    CANSPI_CONFIG_DBL_BUFFER_ON   = 0xFF;     // XXX1XXXX
    CANSPI_CONFIG_DBL_BUFFER_OFF  = 0xEF;     // XXX0XXXX

    CANSPI_CONFIG_MSG_BITS         = 0x60;
    CANSPI_CONFIG_ALL_MSG          = 0xFFz    // X11XXXXX
    CANSPI_CONFIG_VALID_XTD_MSG   = 0xDF;     // X10XXXXX
    CANSPI_CONFIG_VALID_STD_MSG   = 0xBF;     // X01XXXXX
    CANSPI_CONFIG_ALL_VALID_MSG   = 0x9F;     // X00XXXXX
```

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

```
init := CANSPI_CONFIG_SAMPLE_THRICE     and
        CANSPI_CONFIG_PHSEG2_PRG_ON     and
        CANSPI_CONFIG_STD_MSG            and
        CANSPI_CONFIG_DBL_BUFFER_ON     and
        CANSPI_CONFIG_VALID_XTD_MSG     and
        CANSPI_CONFIG_LINE_FILTER_OFF;
...
CANSPIInitialize(1, 1, 3, 3, 1, init);   // initialize CANSPI
```

CANSPI_TX_MSG_FLAGS

CANSPI_TX_MSG_FLAGS are flags related to transmission of a CAN message:

```
const
    CANSPI_TX_PRIORITY_BITS = 0x03;
    CANSPI_TX_PRIORITY_0     = 0xFC;    // XXXXXX00
    CANSPI_TX_PRIORITY_1     = 0xFD;    // XXXXXX01
    CANSPI_TX_PRIORITY_2     = 0xFE;    // XXXXXX10
    CANSPI_TX_PRIORITY_3     = 0xFF;    // XXXXXX11

    CANSPI_TX_FRAME_BIT      = 0x08;
    CANSPI_TX_STD_FRAME      = 0xFF;    // XXXXX1XX
    CANSPI_TX_XTD_FRAME      = 0xF7;    // XXXXX0XX

    CANSPI_TX_RTR_BIT         = 0x40;
    CANSPI_TX_NO_RTR_FRAME   = 0xFF;    // X1XXXXXX
    CANSPI_TX_RTR_FRAME      = 0xBF;    // X0XXXXXX
```

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

```
// form value to be used as sending message flag:
send_config := CANSPI_TX_PRIORITY_0    and
               CANSPI_TX_XTD_FRAME      and
               CANSPI_TX_NO_RTR_FRAME;
...
CANSPIWrite(id, data, 1, send_config);
```

CANSPI_RX_MSG_FLAGS

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

```
const
    CANSPI_RX_FILTER_BITS = 0x07;   // Use this to access filter bits
     CANSPI_RX_FILTER_1     = 0x00;
```

```
CANSPI_RX_FILTER_2     = 0x01;
CANSPI_RX_FILTER_3     = 0x02;
CANSPI_RX_FILTER_4     = 0x03;
CANSPI_RX_FILTER_5     = 0x04;
CANSPI_RX_FILTER_6     = 0x05;

CANSPI_RX_OVERFLOW      = 0x08;   // Set if Overflowed else cleared
CANSPI_RX_INVALID_MSG  = 0x10;// Set if invalid else cleared
CANSPI_RX_XTD_FRAME     = 0x20;   // Set if XTD message else cleared
CANSPI_RX_RTR_FRAME     = 0x40; // Set if RTR message else cleared
CANSPI_RX_DBL_BUFFERED = 0x80;   // Set if this message was hardware
double-buffered
```

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

```
if (MsgFlag and CANSPI_RX_OVERFLOW <> 0) then
begin
  ...
  // Receiver overflow has occurred.
  // We have lost our previous message.
end;
```

CANSPI_MASK

The CANSPI_MASK constants define mask codes. Function CANSPISetMask expects one of these as it's argument:

```
const
     CANSPI_MASK_B1 = 0;
     CANSPI_MASK_B2 = 1;
```

CANSPI_FILTER

The CANSPI_FILTER constants define filter codes. Functions CANSPISetFilter expects one of these as it's argument:

```
const
    CANSPI_FILTER_B1_F1 = 0;
    CANSPI_FILTER_B1_F2 = 1;
    CANSPI_FILTER_B2_F1 = 2;
    CANSPI_FILTER_B2_F2 = 3;
    CANSPI_FILTER_B2_F3 = 4;
    CANSPI_FILTER_B2_F4 = 5;
```

Library Example

This is a simple demonstration of CANSPI Library routines usage. First node initiates the communication with the second node by sending some data to its address. The second node responds by sending back the data incremented by 1. First node then does the same and sends incremented data back to second node, etc.

Code for the first CANSPI node:

```
program Can_Spi_1st;

var Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags : byte;    // can
flags
    Rx_Data_Len : byte;              // received data length in bytes
     RxTx_Data    : array[8]  of byte;        // can rx/tx data buffer
     Msg_Rcvd : byte;                         // reception flag
     Tx_ID, Rx_ID : longint;                 // can rx and tx ID

// CANSPI module connections
var CanSpi_CS : sbit at  PORTB.B0;
     CanSpi_CS_Direction : sbit at DDRB.B0;
     CanSpi_Rst : sbit at PORTB.B2;
     CanSpi_Rst_Direction : sbit at DDRB.B2;
// End CANSPI module connections

begin
  ADCSRA.7 := 0;                          // Set AN pins to Digital I/O
  PORTC := 0;
  DDRC := 255;

 Can_Init_Flags := 0;                                          //
  Can_Send_Flags := 0;                                // clear flags
  Can_Rcv_Flags  := 0;                                          //

  Can_Send_Flags := _CANSPI_TX_PRIORITY_0 and                 // form
value to be used
                    _CANSPI_TX_XTD_FRAME and //   with CANSPIWrite
                    _CANSPI_TX_NO_RTR_FRAME;

  Can_Init_Flags := _CANSPI_CONFIG_SAMPLE_THRICE and         // form
value to be used
                 _CANSPI_CONFIG_PHSEG2_PRG_ON and// with CANSPIInit
                    _CANSPI_CONFIG_XTD_MSG and
                    _CANSPI_CONFIG_DBL_BUFFER_ON and
                    _CANSPI_CONFIG_VALID_XTD_MSG;

  SPI1_Init();                                                    //
initialize SPI1 module
```

```
CANSPIInitialize(1,3,3,3,1,Can_Init_Flags);                       //
Initialize external CANSPI module
  CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF);               //
set CONFIGURATION mode
         CANSPISetMask(_CANSPI_MASK_B1,-1,_CANSPI_CONFIG_XTD_MSG);
// set all mask1 bits to ones
         CANSPISetMask(_CANSPI_MASK_B2,-1,_CANSPI_CONFIG_XTD_MSG);
// set all mask2 bits to ones
     CANSPISetFilter(_CANSPI_FILTER_B2_F4,3,_CANSPI_CONFIG_XTD_MSG);
// set id of filter B1_F1 to 3

  CANSPISetOperationMode(_CANSPI_MODE_NORMAL,0xFF)// set NORMAL mode

  RxTx_Data[0]  := 9;                     // set initial data to be sent

  Tx_ID := 12111;                                    // set transmit ID

  CANSPIWrite(Tx_ID, RxTx_Data, 1, Can_Send_Flags);               //
send initial message
  while (TRUE) do
begin                                                 // endless loop
          Msg_Rcvd := CANSPIRead(Rx_ID , RxTx_Data , Rx_Data_Len,
Can_Rcv_Flags);   // receive message
                    if ((Rx_ID = 3) and Msg_Rcvd) then
// if message received check id
         begin
                                        PORTC := RxTx_Data[0];
// id correct, output data at PORTC
                                              Inc(RxTx_Data[0])    ;
// increment received data
           Delay_ms(10);
                 CANSPIWrite(Tx_ID, RxTx_Data, 1, Can_Send_Flags);
// send incremented data back
         end;
    end;
end.
```

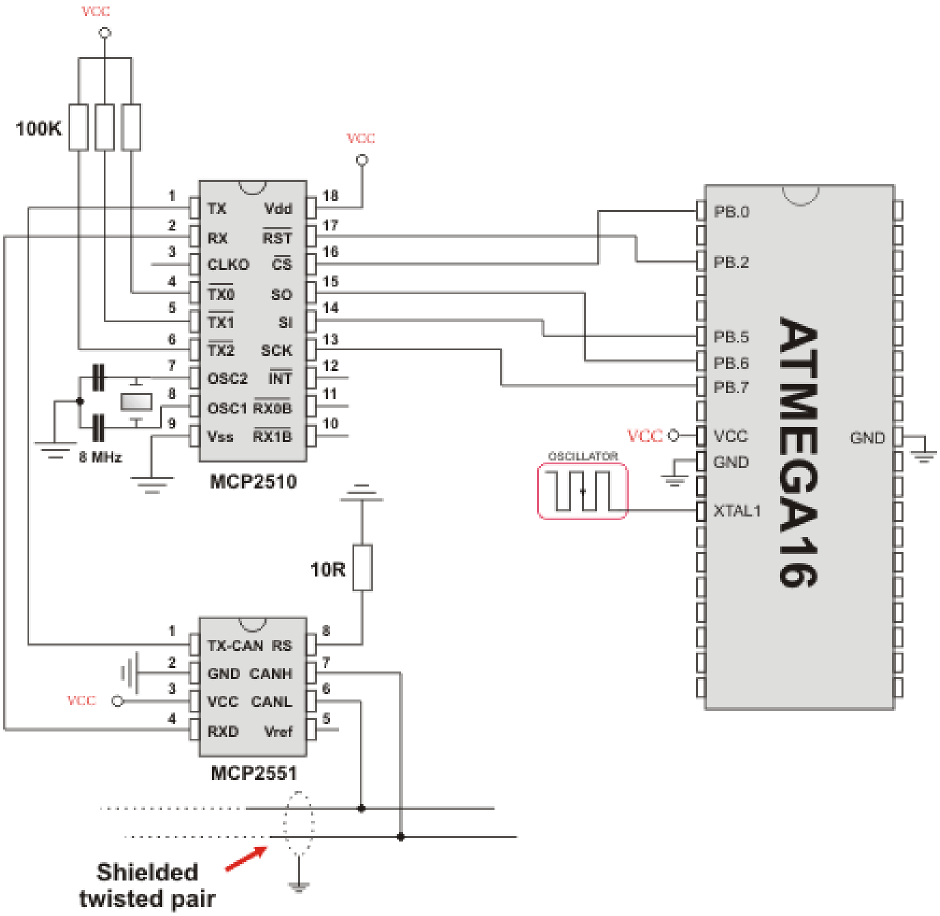
Code for the second CANSPI node:

```
program Can_Spi_2nd;

var Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags : byte;     // can
flags
    Rx_Data_Len : byte;                                           //
received data length in bytes
    RxTx_Data    : array[8] of byte;        // CAN rx/tx data buffer
    Msg_Rcvd : byte;                                // reception flag
    Tx_ID, Rx_ID : longint;                       // can rx and tx ID

// CANSPI module connections
```

```
var CanSpi_CS : sbit at PORTB.B0;
    CanSpi_CS_Direction : sbit at DDRB.B0;
    CanSpi_Rst : sbit at PORTB.B2;
    CanSpi_Rst_Direction : sbit at DDRB.B2;
// End CANSPI module connections

begin

  PORTC := 0;                                  // clear PORTC
  DDRC := 255;                                 // set PORTC as output

  Can_Init_Flags := 0;                                             //
  Can_Send_Flags := 0;                        // clear flags
  Can_Rcv_Flags  := 0;                                             //

  Can_Send_Flags := _CANSPI_TX_PRIORITY_0 and                      //
form value to be used
                   _CANSPI_TX_XTD_FRAME and     //  with CANSPIWrite
                      _CANSPI_TX_NO_RTR_FRAME;

  Can_Init_Flags := _CANSPI_CONFIG_SAMPLE_THRICE and               //
Form value to be used
                  _CANSPI_CONFIG_PHSEG2_PRG_ON and//  with CANSPIInit
                      _CANSPI_CONFIG_XTD_MSG and
                      _CANSPI_CONFIG_DBL_BUFFER_ON and
                      _CANSPI_CONFIG_VALID_XTD_MSG and
                      _CANSPI_CONFIG_LINE_FILTER_OFF;

                                                       SPI1_Init();
// initialize SPI1 module
                      Spi_Rd_Ptr           :=            @SPI1_Read;
// Pass pointer to SPI Read function of used SPI module
                       CANSPIInitialize(1,3,3,3,1,Can_Init_Flags);
// initialize external CANSPI module
                 CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF);
// set CONFIGURATION mode
         CANSPISetMask(_CANSPI_MASK_B1,-1,_CANSPI_CONFIG_XTD_MSG);
// set all mask1 bits to ones
         CANSPISetMask(_CANSPI_MASK_B2,-1,_CANSPI_CONFIG_XTD_MSG);
// set all mask2 bits to ones

CANSPISetFilter(_CANSPI_FILTER_B2_F3,12111,_CANSPI_CONFIG_XTD_MSG);
// set id of filter B1_F1 to 3
                 CANSPISetOperationMode(_CANSPI_MODE_NORMAL,0xFF);
// set NORMAL mode
                             Tx_ID               :=               3;
// set tx ID

while (TRUE) do
```

```
begin
        Msg_Rcvd := CANSPIRead(Rx_ID , RxTx_Data , Rx_Data_Len,
Can_Rcv_Flags);  // receive message
               if  ((Rx_ID  =  12111)  and  Msg_Rcvd)  then
// if message received check id
       begin
                                PORTC  :=  RxTx_Data[0];
// id correct, output data at PORTC
                                     Inc(RxTx_Data[0])   ;
// increment received data
              CANSPIWrite(Tx_ID, RxTx_Data, 1, Can_Send_Flags);
// send incremented data back
       end;
    end;
end.
```

HW Connection



Example of interfacing CAN transceiver MCP2510 with MCU via SPI interface

COMPACT FLASH LIBRARY

The Compact Flash Library provides routines for accessing data on Compact Flash card (abbr. CF further in text). CF cards are widely used memory elements, commonly used with digital cameras. Great capacity and excellent access time of only a few microseconds make them very attractive for the microcontroller applications.

In CF card, data is divided into sectors. One sector usually comprises 512 bytes. Routines for file handling, the Cf_Fat routines, are not performed directly but successively through 512B buffer.

Note: Routines for file handling can be used only with FAT16 file system.

Note: Library functions create and read files from the root directory only.

Note: 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 the FAT1 table gets corrupted.

Note: If MMC/SD card has Master Boot Record (MBR), the library will work with the first available primary (logical) partition that has non-zero size. If MMC/SD card has Volume Boot Record (i.e. there is only one logical partition and no MBRs), the library works with entire card as a single partition. For more information on MBR, physical and logical drives, primary/secondary partitions and partition tables, please consult other resources, e.g. Wikipedia and similar.

Note: Before writing operation, make sure not to overwrite boot or FAT sector as it could make your card on PC or digital camera unreadable. Drive mapping tools, such as Winhex, can be of great assistance.

External dependencies of Compact Flash Library

The following variables must be defined in all projects using Compact Flash Library:	Description:	Example :
`var CF_Data_Port : byte; sfr; external;`	Compact Flash Data Port.	`var CF_Data_Port : byte at PORTD;`
`var CF_Data_Port_Direction : byte; sfr; external;`	Direction of the Compact Flash Data Port.	`var CF_Data_Port_Direction : byte at DDRD;`
`var CF_RDY : sbit; sfr; external;`	Ready signal line.	`var CF_RDY : sbit at PINB.B7;`
`var CF_WE : sbit; sfr; external;`	Write Enable signal line.	`var CF_WE : sbit at PORTB.B6;`
`var CF_OE : sbit; sfr; external;`	Output Enable signal line.	`var CF_OE : sbit at PORTB.B5;`
`var CF_CD1 : sbit; sfr; external;`	Chip Detect signal line.	`var CF_CD1 : sbit at PINB.B4;`
`var CF_CE1 : sbit; sfr; external;`	Chip Enable signal line.	`var CF_CE1 : sbit at PORTB.B3;`
`var CF_A2 : sbit; sfr; external;`	Address pin 2.	`var CF_A2 : sbit at PORTB.B2;`
`var CF_A1 : sbit; sfr; external;`	Address pin 1.	`var CF_A1 : sbit at PORTB.B1;`
`var CF_A0 : sbit; sfr; external;`	Address pin 0.	`var CF_A0 : sbit at PORTB.B0;`
`var CF_RDY_direction : sbit; sfr; external;`	Direction of the Ready pin.	`var CF_RDY_direction : sbit at DDRB.B7;`
`var CF_WE_direction : sbit; sfr; external;`	Direction of the Write Enable pin.	`var CF_WE_direction : sbit at DDRB.B6;`
`var CF_OE_direction : sbit; sfr; external;`	Direction of the Output Enable pin.	`var CF_OE_direction : sbit at DDRB.B5;`
`var CF_CD1_direction : sbit; sfr; external;`	Direction of the Chip Detect pin.	`var CF_CD1_direction : sbit at DDRB.B4;`
`var CF_CE1_direction : sbit; sfr; external;`	Direction of the Chip Enable pin.	`var CF_CE1_direction : sbit at DDRB.B3;`
`var CF_A2_direction : sbit; sfr; external;`	Direction of the Address 2 pin.	`var CF_A2_direction : sbit at DDRB.B2;`
`var CF_A1_direction : sbit; sfr; external;`	Direction of the Address 1 pin.	`var CF_A1_direction : sbit at DDRB.B1;`
`var CF_A0_direction : sbit; sfr; external;`	Direction of the Address 0 pin.	`var CF_A0_direction : sbit at DDRB.B0;`

Library Routines

- Cf_Init
- Cf_Detect
- Cf_Enable
- Cf_Disable
- Cf_Read_Init
- Cf_Read_Byte
- Cf_Write_Init
- Cf_Write_Byte
- Cf_Read_Sector
- Cf_Write_Sector

Routines for file handling:

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

Cf_Init

Prototype	`procedure Cf_Init() ;`
Returns	Nothing.
Description	Initializes ports appropriately for communication with CF card.
Requires	Global variables : - `CF_Data_Port :` Compact Flash data port - `CF_RDY :` Ready signal line - `CF_WE :` Write enable signal line - `CF_OE :` Output enable signal line - `CF_CD1 :` Chip detect signal line - `CF_CE1 :` Enable signal line - `CF_A2 :` Address pin 2

Requires	- CF_A1 : Address pin 1 - CF_A0 : Address pin 0 - CF_Data_Port_direction : Direction of the Compact Flash data direction port - CF_RDY_direction : Direction of the Ready pin - CF_WE_direction : Direction of the Write enable pin - CF_OE_direction : Direction of the Output enable pin - CF_CD1_direction : Direction of the Chip detect pin - CF_CE1_direction : Direction of the Chip enable pin - CF_A2_direction : Direction of the Address 2 pin - CF_A1_direction : Direction of the Address 1 pin - CF_A0_direction : Direction of the Address 0 pin must be defined before using this function.

Example	// set compact flash pinout
	var CF_Data_Port : byte **at** PORTD;
	var Cf_Data_Port_Direction : byte **at** DDRD;
	
	var CF_RDY : **sbit at** PINB.B7;
	var CF_WE : **sbit at** PORTB.B6;
	var CF_OE : **sbit at** PORTB.B5;
	var CF_CD1 : **sbit at** PINB.B4;
	var CF_CE1 : **sbit at** PORTB.B3;
	var CF_A2 : **sbit at** PORTB.B2;
	var CF_A1 : **sbit at** PORTB.B1;
	var CF_A0 : **sbit at** PORTB.B0;
	
	var CF_RDY_direction : **sbit at** DDRB.B7;
	var CF_WE_direction : **sbit at** DDRB.B6;
	var CF_OE_direction : **sbit at** DDRB.B5;
	var CF_CD1_direction : **sbit at** DDRB.B4;
	var CF_CE1_direction : **sbit at** DDRB.B3;
	var CF_A2_direction : **sbit at** DDRB.B2;
	var CF_A1_direction : **sbit at** DDRB.B1;
	var CF_A0_direction : **sbit at** DDRB.B0;
	// end of cf pinout
	
	// Init CF
	begin
	  Cf_Init();
	end;

Cf_Detect

Prototype	`function CF_Detect() : byte ;`
Returns	- `1` - if CF card was detected - `0` - otherwise
Description	Checks for presence of CF card by reading the `chip detect` pin.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// Wait until CF card is inserted:`
	`while (Cf_Detect() = 0) do`
	`  nop;`

Cf_Enable

Prototype	`procedure Cf_Enable();`
Returns	Nothing.
Description	Enables the device. Routine needs to be called only if you have disabled the device by means of the Cf_Disable routine. These two routines in conjunction allow you to free/occupy data line when working with multiple devices.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// enable compact flash`
	`Cf_Enable();`

Cf_Disable

Prototype	`procedure Cf_Disable();`
Returns	Nothing.
Description	Routine disables the device and frees the data lines 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.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// disable compact flash`
	`Cf_Disable();`

Cf_Read_Init

Prototype	`procedure Cf_Read_Init(address : dword; sector_count : byte);`
Returns	Nothing.
Description	Initializes CF card for reading. Parameters : - `address`: the first sector to be prepared for reading operation. - `sector_count`: number of sectors to be prepared for reading operation.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// initialize compact flash for reading from sector 590`
	`Cf_Read_Init(590, 1);`

Cf_Read_Byte

Prototype	`function CF_Read_Byte() : byte;`
Returns	Returns a byte read from Compact Flash sector buffer. **Note:** Higher byte of the `unsigned` return value is cleared.
Description	Reads one byte from Compact Flash sector buffer location currently pointed to by internal read pointers. These pointers will be autoicremented upon reading.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init CF card must be initialized for reading operation. See Cf_Read_Init.

Example	`// Read a byte from compact flash:`
	`var data : byte;`
	`...`
	`data := Cf_Read_Byte();`

Cf_Write_Init

Prototype	`procedure Cf_Write_Init(address : dword; sectcnt : byte);`
Returns	Nothing.
Description	Initializes CF card for writing. Parameters : - `address`: the first sector to be prepared for writing operation. - `sectcnt`: number of sectors to be prepared for writing operation.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// initialize compact flash for writing to sector 590`
	`Cf_Write_Init(590, 1);`

Cf_Write_Byte

Prototype	`procedure Cf_Write_Byte(data_ : byte) ;`
Returns	Nothing.
Description	Writes a byte to Compact Flash sector buffer location currently pointed to by writing pointers. These pointers will be autoicremented upon reading. When sector buffer is full, its content will be transfered to appropriate flash memory sector. Parameters : - `data_`: byte to be written.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init. CF card must be initialized for writing operation. See Cf_Write_Init.

Example	`var data_ : byte;`
	`...`
	`data := 0xAA;`
	`Cf_Write_Byte(data);`

Cf_Read_Sector

Prototype	`procedure Cf_Read_Sector(sector_number : dword; var buffer : array[ 512] of byte);`
Returns	Nothing.
Description	Reads one sector (512 bytes). Read data is stored into buffer provided by the buffer parameter. Parameters : - `sector_number`: sector to be read. - `buffer`: data buffer of at least 512 bytes in length.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// read sector 22`
	`var data : array[ 512] of byte;`
	`...`
	`Cf_Read_Sector(22, data);`

Cf_Write_Sector

Prototype	`procedure Cf_Write_Sector(sector_number : dword; var buffer : array[ 512] of byte) ;`
Returns	Nothing.
Description	Writes 512 bytes of data provided by the buffer parameter to one CF sector. Parameters : - `sector_number`: sector to be written to. - `buffer`: data buffer of 512 bytes in length.
Requires	The corresponding MCU ports must be appropriately initialized for CF card. See Cf_Init.

Example	`// write to sector 22`
	`var data : array[ 512] of byte;`
	`...`
	`Cf_Write_Sector(22, data);`

Cf_Fat_Init

Prototype	`function Cf_Fat_Init(): byte;`
Returns	- `0` - if CF card was detected and successfully initialized - `1` - if FAT16 boot sector was not found - `255` - if card was not detected
Description	Initializes CF card, reads CF FAT16 boot sector and extracts data needed by the library.
Requires	Nothing.

Example	`//init the FAT library`
	`if (Cf_Fat_Init() = 0) then`
	`  begin`
	`  ...`
	`  end`

Cf_Fat_QuickFormat

Prototype	`function Cf_Fat_QuickFormat(var cf_fat_label : string[11]) : byte;`
Returns	- `0` - if CF card was detected, successfully formated and initialized - `1` - if FAT16 format was unseccessful - `255` - if card was not detected
Description	Formats to FAT16 and initializes CF card. Parameters : - `cf_fat_label`: volume label (11 characters in length). If less than 11 characters are provided, the label will be padded with spaces. If an empty string is passed, the volume will not be labeled. **Note:** This routine can be used instead or in conjunction with the Cf_Fat_Init routine. **Note:** If CF card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.
Requires	Nothing.

Example	`// format and initialize the FAT library`
	`if (Cf_Fat_QuickFormat('mikroE') = 0) then`
	`begin`
	`  ...`
	`end;`

Cf_Fat_Assign

<table>
<tr><td><b>Prototype</b></td><td><code>function Cf_Fat_Assign(var filename: array[12] of char; file_cre_attr: byte): byte;</code></td></tr>
<tr><td><b>Returns</b></td><td>- <code>0</code> if file does not exist and no new file is created.
- <code>1</code> if file already exists or file does not exist but a new file is created.</td></tr>
<tr><td><b>Description</b></td><td>Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied to the assigned file.

Parameters :

- <code>filename</code>: name of the file that should be assigned for file operations. The file name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to the proper case automatically, so the user does not have to take care of that.
Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between the file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension.

- <code>file_cre_attr</code>: file creation and attributs flags. Each bit corresponds to the appropriate file attribut:
<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x01</td><td>Read only</td></tr>
<tr><td>1</td><td>0x02</td><td>Hidden</td></tr>
<tr><td>2</td><td>0x04</td><td>System</td></tr>
<tr><td>3</td><td>0x08</td><td>Volum Label</td></tr>
<tr><td>4</td><td>0x10</td><td>Subdirectory</td></tr>
<tr><td>5</td><td>0x20</td><td>Arhive</td></tr>
<tr><td>6</td><td>0x40</td><td>Device(internal use only,never found on disk).</td></tr>
<tr><td>7</td><td>0x80</td><td>File creation flag.If the file does not exist and this flag is set,a new file with specified name will be created.</td></tr>
</table>

<b>Note:</b>Long File Names (LFN) not suppoted</td></tr>
<tr><td><b>Requires</b></td><td>CF card and CF library must be initialized for file operations. See Cf_Fat_Init.</td></tr>
<tr><td><b>Example</b></td><td><code>// create file with archive attribut if it does not already exist</code>
<code>Cf_Fat_Assign('MIKRO007.TXT',0xA0);</code></td></tr>
</table>

Cf_Fat_Reset

Prototype	`procedure Cf_Fat_Reset(var size: dword);`
Returns	Nothing.
Description	Opens currently assigned file for reading. Parameters : - `size`: buffer to store file size to. After file has been open for reading its size is returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`var size : dword;`
	`...`
	`Cf_Fat_Reset(size);`

Cf_Fat_Read

Prototype	`procedure Cf_Fat_Read(var bdata: byte);`
Returns	Nothing.
Description	Reads a byte from currently assigned file opened for reading. Upon function execution file pointers will be set to the next character in the file. Parameters : - `bdata`: buffer to store read byte to. Upon this function execution read byte is - returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign. File must be open for reading. See Cf_Fat_Reset.

Example	`var character : byte;`
	`...`
	`Cf_Fat_Read(character);`

Cf_Fat_Rewrite

Prototype	`procedure Cf_Fat_Rewrite();`
Returns	Nothing.
Description	Opens currently assigned file for writing. If the file is not empty its content will be erased.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. The file must be previously assigned. See Cf_Fat_Assign.

Example	`// open file for writing`
	`Cf_Fat_Rewrite();`

Cf_Fat_Append

Prototype	`procedure Cf_Fat_Append();`
Returns	Nothing.
Description	Opens currently assigned file for appending. Upon this function execution file pointers will be positioned after the last byte in the file, so any subsequent file writing operation will start from there.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`// open file for appending`
	`Cf_Fat_Append();`

Cf_Fat_Delete

Prototype	`procedure Cf_Fat_Delete();`
Returns	Nothing.
Description	Deletes currently assigned file from CF card.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`// delete current file`
	`Cf_Fat_Delete();`

Cf_Fat_Write

Prototype	`procedure Cf_Fat_Write(var fdata: array[512] of byte; data_len: word);`
Returns	Nothing.
Description	Writes requested number of bytes to currently assigned file opened for writing. Parameters : - `fdata`: data to be written. - `data_len`: number of bytes to be written.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.

Example	`var file_contents : array[42] of byte;`
	`...`
	`Cf_Fat_Write(file_contents, 42); // write data to the assigned file`

Cf_Fat_Set_File_Date

Prototype	`procedure Cf_Fat_Set_File_Date(year: word; month: byte; day: byte; hours: byte; mins: byte; seconds: byte);`
Returns	Nothing.
Description	Sets the date/time stamp. Any subsequent file writing operation will write this stamp to currently assigned file's time/date attributs. Parameters : - `year`: year attribute. Valid values: 1980-2107 - `month`: month attribute. Valid values: 1-12 - `day`: day attribute. Valid values: 1-31 - `hours`: hours attribute. Valid values: 0-23 - `mins`: minutes attribute. Valid values: 0-59 - `seconds`: seconds attribute. Valid values: 0-59
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously 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: byte; var day: byte; var hours: byte; var mins: byte);`
Returns	Nothing.
Description	Reads time/date attributes of currently assigned file. Parameters : - `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attribute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`var year : word;`
	`    month, day, hours, mins : byte;`
	`...`
	`Cf_Fat_Get_File_Date(year, month, day, hours, mins);`

Cf_Fat_Get_File_Size

Prototype	`function Cf_Fat_Get_File_Size(): dword;`
Returns	Size of the currently assigned file in bytes.
Description	This function reads size of currently assigned file in bytes.
Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init. File must be previously assigned. See Cf_Fat_Assign.

Example	`var my_file_size : dword;`
	`...`
	`my_file_size := Cf_Fat_Get_File_Size();`

Cf_Fat_Get_Swap_File

Prototype	`function Cf_Fat_Get_Swap_File(sectors_cnt: longint; var filename : string[11]; file_attr : byte): dword;`
Returns	- Number of the start sector for the newly created swap file, if there was enough free space on CF card to create file of required size. - 0 - otherwise.
Description	This function is used to create a swap file of predefined name and size on the CF media. If a file with specified name already exists on the media, search for consecutive sectors will ignore sectors occupied by this file. Therefore, it is recommended to erase such file if it exists before calling this function. If it is not erased and there is still enough space for a new swap file, this function will delete it after allocating new memory space for a new swap file. The purpose of the swap file is to make reading and writing to CF media as fast as possible, by using the Cf_Read_Sector() and Cf_Write_Sector() functions directly, without potentially damaging the FAT system. The swap file can be considered as a "window" on the media where the user can freely write/read data. Its main purpose in the mikroPascal's 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. Parameters: - `sectors_cnt`: number of consecutive sectors that user wants the swap file to - have. - `filename`: name of the file that should be assigned for file operations. The file name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to the proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between the file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension. - `file_attr`: file creation and attributs flags. Each bit corresponds to the appro priate file attribut:

<table>
<tr>
<td><b>Description</b></td>
<td>
<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x01</td><td>Read Only</td></tr>
<tr><td>1</td><td>0x02</td><td>Hidden</td></tr>
<tr><td>2</td><td>0x04</td><td>System</td></tr>
<tr><td>3</td><td>0x08</td><td>Volume Label</td></tr>
<tr><td>4</td><td>0x10</td><td>Subdirectory</td></tr>
<tr><td>5</td><td>0x20</td><td>Archive</td></tr>
<tr><td>6</td><td>0x40</td><td>Device (internal use only, never found on disk)</td></tr>
<tr><td>7</td><td>0x80</td><td>Not used</td></tr>
</table>
<b>Note:</b> Long File Names (LFN) are not supported.
</td>
</tr>
<tr>
<td><b>Requires</b></td>
<td>CF card and CF library must be initialized for file operations. See Cf_Fat_Init.</td>
</tr>
<tr>
<td><b>Example</b></td>
<td>
<pre>
// Try to create a swap file with archive atribute, whose size
will be at least 1000 sectors.
//   If it succeeds, it sends the No. of start sec-
tor over UART
var size : dword;
...
size := Cf_Fat_Get_Swap_File(1000, "mikroE.txt", 0x20);
if (size <> 0) then
begin
  UART1_Write(0xAA);
  UART1_Write(Lo(size));
  UART1_Write(Hi(size));
  UART1_Write(Higher(size));
  UART1_Write(Highest(size));
  UART1_Write(0xAA);
end
...
</pre>
</td>
</tr>
</table>

Library Example

The following example is a simple demonstration of CF(Compact Flash) Library which shows how to use CF card data accessing routines.

```
program CF_Fat16_Test;

var

 // set compact flash pinout
 Cf_Data_Port : byte at PORTD;
 Cf_Data_Port_Direction : byte  at DDRD;

 CF_RDY : sbit at PINB.B7;
 CF_WE  : sbit at PORTB.B6;
 CF_OE  : sbit at PORTB.B5;
 CF_CD1 : sbit at PINB.B4;
 CF_CE1 : sbit at PORTB.B3;
 CF_A2  : sbit at PORTB.B2;
 CF_A1  : sbit at PORTB.B1;
 CF_A0  : sbit at PORTB.B0;

 CF_RDY_direction : sbit at DDRB.B7;
 CF_WE_direction  : sbit at DDRB.B6;
 CF_OE_direction  : sbit at DDRB.B5;
 CF_CD1_direction : sbit at DDRB.B4;
 CF_CE1_direction : sbit at DDRB.B3;
 CF_A2_direction  : sbit at DDRB.B2;
 CF_A1_direction  : sbit at DDRB.B1;
 CF_A0_direction  : sbit at DDRB.B0;
 // end of cf pinout

 FAT_TXT : string[ 20] ;
 file_contents : string[ 50] ;

 filename : string[ 14] ;    // File names

 character : byte;
 loop, loop2 : byte;
 size : longint;

 Buffer : array[ 512]  of byte;

//-------------- Writes string to USART
procedure Write_Str(var ostr: array[ 2]  of byte);
var
  i : byte;
begin
 i := 0;
```

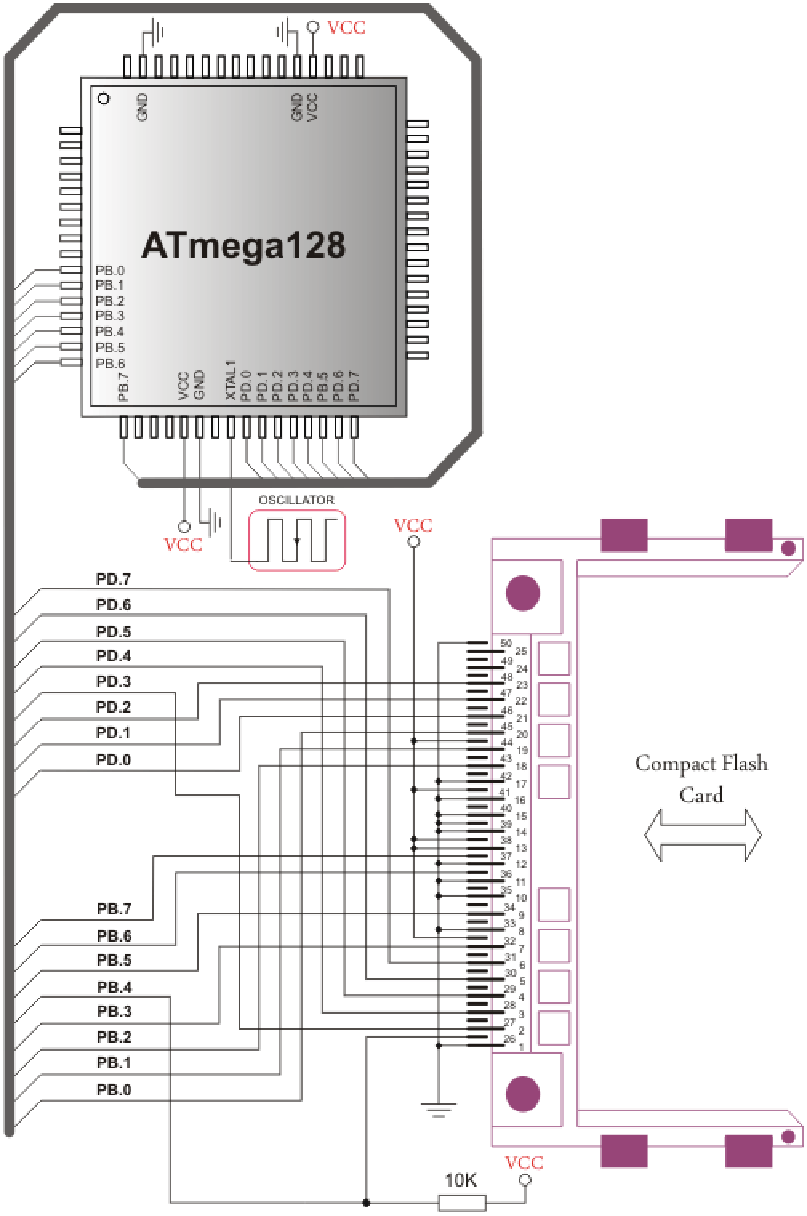
```
while ostr[ i]  <> 0 do begin
    UART1_Write(ostr[ i] );
    Inc(i);
  end;
  UART1_Write($0A);
end;

//-------------- Creates new file and writes some data to it
procedure Create_New_File;
begin
  filename[ 7]  := 'A';
  Cf_Fat_Assign(filename, 0xA0);       // Will not find file and then
create file
  Cf_Fat_Rewrite();          // To clear file and start with new data
  for loop:=1 to 90 do     //  We want 5 files on the MMC card
    begin
      PORTC := loop;
      file_contents[ 0]  := loop div 10 + 48;
      file_contents[ 1]  := loop mod 10 + 48;
    Cf_Fat_Write(file_contents, 38); // write data to the assigned file
      UART1_Write('.');
    end;
end;

//-------------- Creates many new files and writes data to them
procedure Create_Multiple_Files;
begin
  for loop2 := 'B' to 'Z' do
    begin
      UART1_Write(loop2); // this line can slow down the performance
      filename[ 7]  := loop2;                    // set filename
      Cf_Fat_Assign(filename, 0xA0);        // find existing file or
create a new one
     Cf_Fat_Rewrite;        // To clear file and start with new data
      for loop := 1 to 44 do
        begin
          file_contents[ 0]  := loop div 10 + 48;
          file_contents[ 1]  := loop mod 10 + 48;
            Cf_Fat_Write(file_contents,  38);  //  write data to the
assigned file
        end;
    end;
end;

//-------------- Opens an existing file and rewrites it
procedure Open_File_Rewrite;
begin
  filename[ 7]  := 'C';              // Set filename for single-file tests
Cf_Fat_Assign(filename, 0);
Cf_Fat_Rewrite;
```

```
  for loop := 1 to 55 do
   begin
    file_contents[0] := byte(loop div 10 + 48);
    file_contents[1] := byte(loop mod 10 + 48);
    Cf_Fat_Write(file_contents, 38); // write data to the assigned file
   end;
 end;

//-------------- Opens an existing file and appends data to it
//                 (and alters the date/time stamp)
procedure Open_File_Append;
 begin
   filename[7] := 'B';
   Cf_Fat_Assign(filename, 0);
   Cf_Fat_Set_File_Date(2005,6,21,10,35,0);
   Cf_Fat_Append;
   file_contents := ' for mikroElektronika 2005';     // Prepare file
for append
   file_contents[26] := 10;                           // LF
   Cf_Fat_Write(file_contents, 27);    // Write data to assigned file
 end;

//-------------- Opens an existing file, reads data from it and puts
it to USART
procedure Open_File_Read;
begin
  filename[7] := 'B';
  Cf_Fat_Assign(filename, 0);
  Cf_Fat_Reset(size);             // To read file, procedure returns
size of file
  while size > 0 do begin
    Cf_Fat_Read(character);
    UART1_Write(character);     // Write data to USART
    Dec(size);
  end;
end;

//-------------- Deletes a file. If file doesn't exist, it will first
be created
//                 and then deleted.
procedure Delete_File;
begin
  filename[7] := 'F';
  Cf_Fat_Assign(filename, 0);
  Cf_Fat_Delete;
end;

//-------------- Deletes a file. If file doesn't exist, it will first
be created
```

```
//                  and then deleted.
procedure Delete_File;
begin
  filename[ 7] := 'F';
  Cf_Fat_Assign(filename, 0);
  Cf_Fat_Delete;
end;

//-------------- Tests whether file exists, and if so sends its cre-
ation date
//                  and file size via USART
procedure Test_File_Exist(fname : byte);
var
  fsize: longint;
  year: word;
  month, day, hour, minute: byte;
  outstr: array[ 12] of byte;
begin
  filename[ 7] := 'B';           //uncomment this line to search for
file that DOES exists
//  filename[ 7] := 'F';         //uncomment this line to search for
file that DOES NOT exist
  if Cf_Fat_Assign(filename, 0) <> 0 then begin
    //--- file has been found - get its date
    Cf_Fat_Get_File_Date(year,month,day,hour,minute);
    WordToStr(year, outstr);
    Write_Str(outstr);
    ByteToStr(month, outstr);
    Write_Str(outstr);
    WordToStr(day, outstr);
    Write_Str(outstr);
    WordToStr(hour, outstr);
    Write_Str(outstr);
    WordToStr(minute, outstr);
    Write_Str(outstr);
    //--- get file size
    fsize := Cf_Fat_Get_File_Size;
    LongIntToStr(fsize, outstr);
    Write_Str(outstr);
  end
  else begin
    //--- file was not found - signal it
    UART1_Write(0x55);
    Delay_ms(1000);
    UART1_Write(0x55);
  end;
end;

//-------------- Tries to create a swap file, whose size will be at
least 100
```

```
//                 sectors (see Help for details)
procedure M_Create_Swap_File;
  var i : word;

  begin
    for i:=0 to 511 do
      Buffer[ i]  := i;

    size := Cf_Fat_Get_Swap_File(5000, 'mikroE.txt', 0x20);   // see
help on this function for details

    if (size <> 0) then
      begin
        LongIntToStr(size, fat_txt);
        Write_Str(fat_txt);

        for i:=0 to 4999 do
          begin
            Cf_Write_Sector(size, Buffer);
            size := size+1;
            UART1_Write('.');
          end;
      end;
  end;


//--------------- Main. Uncomment the function(s) to test the desired
operation(s)
begin
      FAT_TXT := 'FAT16 not found';
      file_contents := 'XX CF FAT16 library by Anton Rieckert';
      file_contents[ 37]  := 10;            // newline
      filename := 'MIKRO00xTXT';

      // we will use PORTC to signal test end
      DDRC   := 0xFF;
      PORTC  := 0;

      UART1_Init(19200);            // Set up USART for file reading
      delay_ms(100);
      UART1_Write_Text(':Start:');

      // --- Init the FAT library
     // --- use Cf_Fat_QuickFormat instead of init routine if a for-
mat is needed
      if Cf_Fat_Init() = 0 then
        begin
          //--- test functions
           //----- test group #1
           Open_File_Read();
          Create_Multiple_Files();
          //----- test group #2
          Open_File_Rewrite();
          Open_File_Append();
          Delete_File;
          //----- test group #3
          Open_File_Read();
          Test_File_Exist('F');
          M_Create_Swap_File();
          //--- Test termination
          UART1_Write(0xAA);
        end
      else
        begin
          UART1_Write_Text(FAT_TXT);
        end;
      //--- signal end-of-test
      UART1_Write_Text(':End:');
end.
```

HW Connection



Pin diagram of CF memory card

EEPROM LIBRARY

EEPROM data memory is available with a number of AVR family. The mikroPascal PRO for AVR includes a library for comfortable work with MCU's internal EEPROM.

Note: EEPROM Library functions implementation is MCU dependent, consult the appropriate MCU datasheet for details about available EEPROM size and constrains.

Library Routines

- EEPROM_Read
- EEPROM_Write

EEPROM_Read

Prototype	`function EEPROM_Read(address: word) : byte;`
Returns	Byte from the specified address.
Description	Reads data from specified `address`. Parameters : - `address`: address of the EEPROM memory location to be read.
Requires	Nothing.

Example	`var eeAddr : word;`
	`temp : byte;`
	`...`
	`eeAddr := 2`
	`temp := EEPROM_Read(eeAddr);`

EEPROM_Write

Prototype	`procedure EEPROM_Write(address: word; wrdata: byte);`
Returns	Nothing.
Description	Writes wrdata to specified address. Parameters : - `address`: address of the EEPROM memory location to be written. - `wrdata`: data to be written. **Note:** Specified memory location will be erased before writing starts.
Requires	Nothing.

Example	`var eeWrite : byte;`
	`    wrAddr : word;`
	`...`
	`address := 0x02;`
	`wrdata := 0xAA;`
	`EEPROM_Write(wrAddr, eeWrite);`

Library Example

This example demonstrates using the EEPROM Library with ATmega16 MCU.

First, some data is written to EEPROM in byte and block mode; then the data is read from the same locations and displayed on PORTA, PORTB and PORTC.

```
program Eeprom;

var counter : byte;                                 // loop variable
begin
  DDRA := 0xFF;
  DDRB := 0xFF;
  DDRC := 0xFF;

  for counter := 0 to 31 do                          // Fill data buffer
   EEPROM_Write(0x100 + counter, counter);           // Write data to address
0x100+counter

  EEPROM_Write(0x02,0xAA);          // Write some data at address 2
  EEPROM_Write(0x150,0x55);         // Write some data at address 0x150

  Delay_ms(1000);                   // Blink PORTA and PORTB diodes
  PORTA := 0xFF;                    // to indicate reading start
PORTB := 0xFF;
```

```
    Delay_ms(1000);
  PORTA := 0x00;
  PORTB := 0x00;
  Delay_ms(1000);

  PORTA := EEPROM_Read(0x02);                     // Read data from
address 2 and display it on PORT0
  PORTB := EEPROM_Read(0x150);                    // Read data from
address 0x150 and display it on PORT1

  Delay_ms(1000);

  for counter := 0 to 31 do                        // Read 32 bytes
block from address 0x100
   begin
     PORTC := EEPROM_Read(0x100+counter);          //   and display
data on PORT2
     Delay_ms(100);
   end;
end.
```

FLASH MEMORY LIBRARY

This library provides routines for accessing microcontroller Flash memory. Note that prototypes differ for MCU to MCU due to the amount of Flash memory.

Note: Due to the AVR family flash specifics, flash library is MCU dependent. Since some AVR MCU's have more or less than 64kb of Flash memory, prototypes may be different from chip to chip.
Please refer to datasheet before using flash library.

Note: Currently, Write operations are not supported. See mikroPascal PRO for AVR specifics for details.

Library Routines

- FLASH_Read_Byte
- FLASH_Read_Bytes
- FLASH_Read_Word
- FLASH_Read_Words
- Flash_Write
- Flash_Write_8
- Flash_Write_16
- Flash_Write_32
- Flash_Write_64
- Flash_Erase
- Flash_Erase_64
- Flash_Erase_1024
- Flash_Erase_Write
- Flash_Erase_Write_64
- Flash_Erase_Write-1024

FLASH_Read_Byte

Prototype	`// for MCUs with 64kb of Flash memory or less`
	`function FLASH_Read_Byte(address : word) : byte;`
	`// for MCUs with Flash memory larger than 64kb`
	`function FLASH_Read_Byte(address : dword) : byte;`

Returns	Returns data byte from Flash memory.
Description	Reads data from the specified address in Flash memory.
Requires	Nothing.

Example	`// for MCUs with Flash memory larger than 64kb`
	`var tmp : dword;`
	`...`
	`begin`
	`  tmp := Flash_Read(0x0D00);`
	`end`

FLASH_Read_Bytes

Prototype	`// for MCUs with 64kb of Flash memory or less`
	`procedure FLASH_Read_Bytes(address : word; buffer : ^byte; NoBytes : word);`
	
	`// for MCUs with Flash memory larger than 64kb`
	`procedure FLASH_Read_Bytes(address : dword; buffer : ^byte; NoBytes : word)`

Returns	Nothing.
Description	Reads number of data bytes defined by NoBytes parameter from the specified address in Flash memory to variable pointed by buffer.
Requires	Nothing.

Example	`// for MCUs with Flash memory larger than 64kb`
	`const F_ADDRESS : long = 0x200;`
	`var dat_buff : array[ 32]  of word;`
	`...`
	`begin`
	`  FLASH_Read_Bytes(F_ADDRESS, dat_buff, 64);`
	`end.`

FLASH_Read_Word

Prototype	`// for MCUs with 64kb of Flash memory or less`
	`function FLASH_Read_Word(address : word) : word;`
	
	`// for MCUs with Flash memory larger than 64kb`
	`function FLASH_Read_Word(address : dword) : word;`

Returns	Returns data word from Flash memory.
Description	Reads data from the specified address in Flash memory.
Requires	Nothing.

Example	`// for MCUs with Flash memory larger than 64kb`
	`var tmp : word;`
	`...`
	`begin`
	`  tmp := Flash_Read(0x0D00);`
	`begin`

FLASH_Read_Words

Prototype	`// for MCUs with 64kb of Flash memory or less`
	`procedure FLASH_Read_Words(address : word; buffer : ^word; NoWords : word);`
	
	`// for MCUs with Flash memory larger than 64kb`
	`procedure FLASH_Read_Words(address : dword; buffer : ^word; NoWords : word);`

Returns	Nothing.
Description	Reads number of data words defined by NoWords parameter from the specified address in Flash memory to variable pointed by buffer.
Requires	Nothing.

Example	`// for MCUs with Flash memory larger than 64kb`
	`const F_ADDRESS : dword = 0x200;`
	`var dat_buff : array[32] of word;`
	`...`
	`begin`
	`  FLASH_Read_Words(F_ADDRESS,dat_buff, 32);`
	`end.`

Library Example

The example demonstrates simple write to the flash memory for AVR, then reads the data and displays it on PORTB and PORTD.

```
program Flash_MCU_test;

const F_ADDRESS : longint = 0x200;

const data_ : array[32] of word = (                      // constant table
  0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007,
  0x0008,0x0009,0x000A,0x000B,0x000C,0x000D,0x000E,0x000F,
  0x0000,0x0100,0x0200,0x0300,0x0400,0x0500,0x0600,0x0700,
  0x0800,0x0900,0x0A00,0x0B00,0x0C00,0x0D00,0x0E00,0x0F00
 ); org 0x200;

var counter : byte;
    word_ : word;
    dat_buff : array[32] of word;

begin
  DDRD := 0xFF;                                  // set direction to be output
  DDRB := 0xFF;                                  // set direction to be output
  word_ := data_[0];                            //  link const table
 counter := 0;
  while ( counter < 64 ) do              //  reading 64 bytes in loop
    begin
      PORTD := FLASH_Read_Byte(F_ADDRESS + counter);     //  demon-
stration of reading single byte
      Inc(counter);
      PORTB := FLASH_Read_Byte(F_ADDRESS + counter);     //  demon-
stration of reading single byte
      Inc(counter);
      Delay_ms(200);
    end;

 FLASH_Read_Bytes(F_ADDRESS, @dat_buff, 64);             //  demon-
stration of reading 64 bytes
  for counter := 0 to 31 do
    begin
     PORTD := dat_buff[ counter] ;        //  output low byte to PORTD
      PORTB := word((dat_buff[ counter]  shr 8));        //  output
higher byte to PORTB
      Delay_ms(200);
    end;

  counter := 0;
  while (counter <= 63) do                 // reading 32 words in loop
    begin
      word_  := FLASH_Read_Word(F_ADDRESS + counter);    //  demon-
stration of reading single word
     PORTD  := word_;                    //  output low byte to PORTD
      PORTB  := word(word_ shr 8);                       //  output
higher byte to PORTB
      counter := counter + 2;
      Delay_ms(200);
    end;


  FLASH_Read_Words(F_ADDRESS, @dat_buff, 32);            //  demon-
stration of reading 64 bytes
  for counter := 0 to 31 do
    begin
     PORTD := dat_buff[ counter] ;        //  output low byte to PORTD
      PORTB := word((dat_buff[ counter]  shr 8));        //  output
higher byte to PORTB
      Delay_ms(200);
    end;

end.
```

GRAPHIC LCD LIBRARY

The mikroPascal PRO for AVR provides a library for operating Graphic Lcd 128x64 (with commonly used Samsung KS108/KS107 controller).

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

External dependencies of Graphic Lcd Library

The following variables must be defined in all projects using Sound Library:	Description:	Example :
`var GLCD_DataPort : byte; sfr; external;`	Glcd Data Port.	`var GLCD_DataPort : byte at PORTC;`
`varGLCD_DataPort_Direction : byte; sfr; external;`	Direction of the Glcd Data Port.	`var GLCD_DataPort_Direction : byte at DDRC;`
`var GLCD_CS1 : sbit; sfr; external;`	Chip Select 1 line.	`var GLCD_CS1 : sbit at PORTD.B2;`
`var GLCD_CS2 : sbit; sfr; external;`	Chip Select 2 line.	`var GLCD_CS2 : sbit at PORTD.B3;`
`var GLCD_RS : sbit; sfr; external;`	Register select line.	`var GLCD_RS : sbit at PORTD.B4;`
`var GLCD_RW : sbit; sfr; external;`	Read/Write line.	`var GLCD_RW : sbit at PORTD.B5;`
`var GLCD_EN : sbit; sfr; external;`	Enable line.	`var GLCD_EN : sbit at PORTD.B6;`
`var GLCD_RST : sbit; sfr; external;`	Reset line.	`var GLCD_RST : sbit at PORTD.B7;`
`var GLCD_CS1_Direction : sbit; sfr; external;`	Direction of the Chip Select 1 pin.	`var GLCD_CS1_Direction : sbit at DDRD.B2;`
`var GLCD_CS2_Direction : sbit; sfr; external;`	Direction of the Chip Select 2 pin.	`var GLCD_CS2_Direction : sbit at DDRD.B3;`
`var GLCD_RS_Direction : sbit; sfr; external;`	Direction of the Register select pin.	`var GLCD_RS_Direction : sbit at DDRD.B4;`
`var GLCD_RW_Direction : sbit; sfr; external;`	Direction of the Read/Write pin.	`var GLCD_RW_Direction : sbit at DDRD.B5;`
`var GLCD_EN_Direction : sbit; sfr; external;`	Direction of the Enable pin.	`var GLCD_EN_Direction : sbit at DDRD.B6;`
`var GLCD_RST_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var GLCD_RST_Direction : sbit at DDRD.B7;`

Library Routines

Basic routines:

- Glcd_Init
- Glcd_Set_Side
- Glcd_Set_X
- Glcd_Set_Page
- 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

<table>
<tr><td><b>Prototype</b></td><td>

```
procedure Glcd_Init();
```

</td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>Initializes the Glcd module. Each of the control lines is both port and pin configurable, while data lines must be on a single port (pins <0:7&gt;).</td></tr>
<tr><td><b>Requires</b></td><td>

Global variables :

- GLCD_CS1 : Chip select 1 signal pin
- GLCD_CS2 : Chip select 2 signal pin
- GLCD_RS : Register select signal pin
- GLCD_RW : Read/Write Signal pin
- GLCD_EN : Enable signal pin
- GLCD_RST : Reset signal pin
- GLCD_DataPort : Data port
- GLCD_CS1_Direction : Direction of the Chip select 1 pin
- GLCD_CS2_Direction : Direction of the Chip select 2 pin
- GLCD_RS_Direction : Direction of the Register select signal pin
- GLCD_RW_Direction : Direction of the Read/Write signal pin
- GLCD_EN_Direction : Direction of the Enable signal pin
- GLCD_RST_Direction : Direction of the Reset signal pin
- GLCD_DataPort_Direction : Direction of the Data port

must be defined before using this function.

</td></tr>
<tr><td><b>Example</b></td><td>

```
// Glcd module connections
var GLCD_DataPort : byte at PORTC;
    GLCD_DataPort_Direction : byte at DDRC;

var GLCD_CS1  : sbit at PORTD.B2;
    GLCD_CS2  : sbit at PORTD.B3;
    GLCD_RS   : sbit at PORTD.B4;
    GLCD_RW   : sbit at PORTD.B5;
    GLCD_EN   : sbit at PORTD.B6;
    GLCD_RST  : sbit at PORTD.B7;

var GLCD_CS1_Direction : sbit at DDRD.B2;
    GLCD_CS2_Direction : sbit at DDRD.B3;
    GLCD_RS_Direction  : sbit at DDRD.B4;
    GLCD_RW_Direction  : sbit at DDRD.B5;
    GLCD_EN_Direction  : sbit at DDRD.B6;
    GLCD_RST_Direction : sbit at DDRD.B7;
// End Glcd module connections

...

Glcd_Init();
```

</td></tr>
</table>

Glcd_Set_Side

Prototype	`procedure Glcd_Set_Side(x_pos: byte);`
Returns	Nothing.
Description	Selects Glcd side. Refer to the Glcd datasheet for detailed explaination. Parameters : - `x_pos`: position on x-axis. Valid values: 0..127 The parameter `x_pos` specifies the Glcd side: values from 0 to 63 specify the left side, values from 64 to 127 specify the right side. Note: For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`The following two lines are equivalent, and both of them select the left side of Glcd:`
	`Glcd_Select_Side(0);`
	`Glcd_Select_Side(10);`

Glcd_Set_X

Prototype	`procedure Glcd_Set_X(x_pos: byte);`
Returns	Nothing.
Description	Sets x-axis position to x_pos dots from the left border of Glcd within the selected side. Parameters : - `x_pos`: position on x-axis. Valid values: 0..63 **Note:** For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`Glcd_Set_X(25);`

Glcd_Set_Page

Prototype	`procedure Glcd_Set_Page(page: byte);`
Returns	Nothing.
Description	Selects page of the Glcd. Parameters : - `page`: page number. Valid values: 0..7 **Note:** For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized, see Glcd_Init routine.
Example	`Glcd_Set_Page(5);`

Glcd_Read_Data

Prototype	`function Glcd_Read_Data(): byte;`
Returns	One byte from Glcd memory.
Description	Reads data from from the current location of Glcd memory and moves to the next location.
Requires	Glcd needs to be initialized, see Glcd_Init routine. Glcd side, x-axis position and page should be set first. See functions Glcd_Set_Side, Glcd_Set_X, and Glcd_Set_Page.

Example	`var data: byte;`
	`...`
	`data := Glcd_Read_Data();`

Glcd_Write_Data

Prototype	`procedure Glcd_Write_Data(ddata: byte);`
Returns	Nothing.
Description	Writes one byte to the current location in Glcd memory and moves to the next location. Parameters : - `ddata`: data to be written
Requires	Glcd needs to be initialized, see Glcd_Init routine. Glcd side, x-axis position and page should be set first. See functions Glcd_Set_Side, Glcd_Set_X, and Glcd_Set_Page.

Example	`var data: byte;`
	`...`
	`Glcd_Write_Data(data);`

Glcd_Fill

Prototype	`procedure Glcd_Fill(pattern: byte);`
Returns	Nothing.
Description	Fills Glcd memory with the byte pattern. Parameters : - `pattern`: byte to fill Glcd memory with To clear the Glcd screen, use `Glcd_Fill(0)`. To fill the screen completely, use `Glcd_Fill(0xFF)`.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Clear screen`
	`Glcd_Fill(0);`

Glcd_Dot

Prototype	`procedure Glcd_Dot(x_pos: byte; y_pos: byte; color: byte);`
Returns	Nothing.
Description	Draws a dot on Glcd at coordinates (`x_pos, y_pos`). Parameters : - `x_pos`: x position. Valid values: 0..127 - `y_pos`: y position. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines a dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state. **Note:** For x and y axis layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Invert the dot in the upper left corner`
	`Glcd_Dot(0, 0, 2);`

Glcd_Line

Prototype	`procedure Glcd_Line(x_start: integer; y_start: integer; x_end: integer; y_end: integer; color: byte);`
Returns	Nothing.
Description	Draws a line on Glcd. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `y_start`: y coordinate of the line start. Valid values: 0..63 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Draw a line between dots (0,0) and (20,30)`
	`Glcd_Line(0, 0, 20, 30, 1);`

Glcd_V_Line

Prototype	`procedure Glcd_V_Line(y_start: byte; y_end: byte; x_pos: byte; color: byte);`
Returns	Nothing.
Description	Draws a vertical line on Glcd. Parameters : - `y_start`: y coordinate of the line start. Valid values: 0..63 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `x_pos`: x coordinate of vertical line. Valid values: 0..127 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Draw a vertical line between dots (10,5) and (10,25)`
	`Glcd_V_Line(5, 25, 10, 1);`

Glcd_H_Line

Prototype	`procedure Glcd_V_Line(x_start: byte; x_end: byte; y_pos: byte; color: byte);`
Returns	Nothing.
Description	Draws a horizontal line on Glcd. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_pos`: y coordinate of horizontal line. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter `color` determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Draw a horizontal line between dots (10,20) and (50,20)`
	`Glcd_H_Line(10, 50, 20, 1);`

Glcd_Rectangle

Prototype	`procedure Glcd_Rectangle(x_upper_left: byte; y_upper_left: byte; x_bottom_right: byte; y_bottom_right: byte; color: byte);`
Returns	Nothing.
Description	Draws a rectangle on Glcd. Parameters : - `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Draw a rectangle between dots (5,5) and (40,40)`
	`Glcd_Rectangle(5, 5, 40, 40, 1);`

Glcd_Box

Prototype	`procedure Glcd_Box(x_upper_left: byte; y_upper_left: byte; x_bottom_right: byte; y_bottom_right: byte; color: byte);`
Returns	Nothing.
Description	Draws a box on Glcd. Parameters : - `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Draw a box between dots (5,15) and (20,40)`
	`Glcd_Box(5, 15, 20, 40, 1);`

Glcd_Circle

Prototype	`procedure Glcd_Circle(x_center: integer; y_center: integer; radius: integer; color: byte);`
Returns	Nothing.
Description	Draws a circle on Glcd. Parameters : - `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Draw a circle with center in (50,50) and radius=10`
	`Glcd_Circle(50, 50, 10, 1);`

Glcd_Set_Font

Prototype	`procedure Glcd_Set_Font(const ActiveFont: ^byte; FontWidth: byte; FontHeight: byte; FontOffs: word);`
Returns	Nothing.
Description	Sets font that will be used with Glcd_Write_Char and Glcd_Write_Text routines. Parameters : - activeFont: font to be set. Needs to be formatted as an array of byte - aFontWidth: width of the font characters in dots. - aFontHeight: height of the font characters in dots. - aFontOffs: number that represents difference between the mikroPascal PRO for AVR character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroPascal PRO for AVR character set, aFontOffs is 20). Demo fonts supplied with the library have an offset of 32, which means that they start with space. The user can use fonts given in the file “__Lib_GLCDFonts.mpas” file located in the Uses folder or create his own fonts.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Use the custom 5x7 font "myfont" which starts with space (32):`
	`Glcd_Set_Font(myfont, 5, 7, 32);`

Glcd_Write_Char

Prototype	`procedure Glcd_Write_Char(chr: byte; x_pos: byte; page_num: byte; color: byte);`
Returns	Nothing.
Description	Prints character on the Glcd. Parameters : - `chr`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the character: 0 white, 1 black, and 2 inverts each dot. **Note:** For x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized, see Glcd_Init routine. Use Glcd_Set_Font to specify the font for display; if no font is specified, then default 5x8 font supplied with the library will be used.

Example	`' Write character 'C' on the position 10 inside the page 2:`
	`Glcd_Write_Char('C', 10, 2, 1);`

Glcd_Write_Text

Prototype	`procedure Glcd_Write_Text(var text: array[20] of char; x_pos: byte; page_num: byte; color: byte);`
Returns	Nothing.
Description	Prints text on Glcd. Parameters : - `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the text: 0 white, 1 black, and 2 inverts each dot. **Note:** For x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized, see Glcd_Init routine. Use Glcd_Set_Font to specify the font for display; if no font is specified, then default 5x8 font supplied with the library will be used.

Example	`' Write text "Hello world!" on the position 10 inside the page 2:`
	`Glcd_Write_Text("Hello world!", 10, 2, 1);`

Glcd_Image

Prototype	`procedure Glcd_Image(const image: ^byte);`
Returns	Nothing.
Description	Displays bitmap on Glcd. Parameters : - `image`: image to be displayed. Bitmap array must be located in code memory. Use the mikroPascal PRO for AVR integrated Glcd Bitmap Editor to convert image to a constant array suitable for displaying on Glcd.
Requires	Glcd needs to be initialized, see Glcd_Init routine.

Example	`' Draw image my_image on Glcd`
	`Glcd_Image(my_image);`

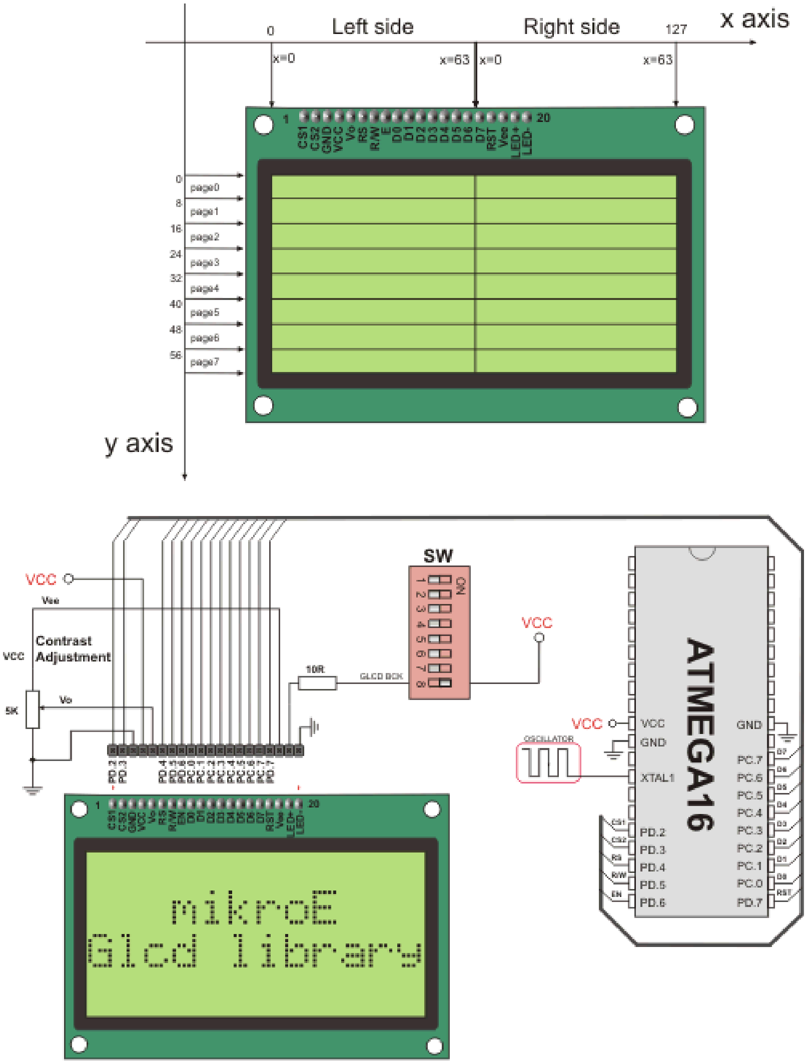
Library Example

The following example demonstrates routines of the Glcd library: initialization, clear(pattern fill), image displaying, drawing lines, circles, boxes and rectangles, text displaying and handling.

```
program Glcd_Test;

uses bitmap;

// Glcd module connections
var GLCD_DataPort : byte at PORTC;
    GLCD_DataPort_Direction : byte at DDRC;
//  end Glcd module connections

var GLCD_CS1  : sbit at PORTD.B2;
    GLCD_CS2  : sbit at PORTD.B3;
    GLCD_RS   : sbit at PORTD.B4;
    GLCD_RW   : sbit at PORTD.B5;
    GLCD_EN   : sbit at PORTD.B6;
    GLCD_RST  : sbit at PORTD.B7;

var GLCD_CS1_Direction : sbit at DDRD.B2;
    GLCD_CS2_Direction : sbit at DDRD.B3;
    GLCD_RS_Direction  : sbit at DDRD.B4;
    GLCD_RW_Direction  : sbit at DDRD.B5;
    GLCD_EN_Direction  : sbit at DDRD.B6;
    GLCD_RST_Direction : sbit at DDRD.B7;
// End Glcd module connections

var counter : byte;
    someText : array[ 18]  of char

procedure Delay2S();                    // 2 seconds delay function
  begin
    Delay_ms(2000);
  end;

begin

  Glcd_Init();                          // Initialize Glcd
  Glcd_Fill(0x00);                      // Clear Glcd

 while TRUE do
    begin
      Glcd_Image(@truck_bmp);           // Draw image
      Delay2S(); delay2S();

Glcd_Fill(0x00);                        // Clear Glcd
```

```
       Glcd_Box(62,40,124,63,1);                // Draw box
       Glcd_Rectangle(5,5,84,35,1);             // Draw rectangle
       Glcd_Line(0, 0, 127, 63, 1);             // Draw line
       Delay2S();
       counter := 5;

      while (counter <= 59) do // Draw horizontal and vertical lines
         begin
           Delay_ms(250);
           Glcd_V_Line(2, 54, counter, 1);
           Glcd_H_Line(2, 120, counter, 1);
           Counter := counter + 5;
         end;

       Delay2S();

       Glcd_Fill(0x00);                         // Clear Glcd

          Glcd_Set_Font(@Character8x7,  8,  7,  32);   //  Choose  font
"Character8x7"
       Glcd_Write_Text('mikroE', 1, 7, 2);      // Write string

       for counter := 1 to 10 do                // Draw circles
         Glcd_Circle(63,32, 3*counter, 1);
       Delay2S();

   Glcd_Box(12,20, 70,57, 2);                   // Draw box}
       Delay2S();

       Glcd_Fill(0xFF);                         // Fill Glcd
       Glcd_Set_Font(@Character8x7, 8, 7, 32);  // Change font
       someText := '8x7 Font';
       Glcd_Write_Text(someText, 5, 0, 2);       // Write string
       delay2S();

       Glcd_Set_Font(@System3x6, 3, 5, 32);      // Change font
       someText := '3X5 CAPITALS ONLY';
       Glcd_Write_Text(someText, 60, 2, 2);      // Write string
       delay2S();

       Glcd_Set_Font(@font5x7, 5, 7, 32);        // Change font
       someText := '5x7 Font';
       Glcd_Write_Text(someText, 5, 4, 2);       // Write string
       delay2S();
Glcd_Set_Font(@FontSystem5x7_v2, 5, 7, 32);     // Change font
       someText := '5x7 Font (v2)';
Glcd_Write_Text(someText, 5, 6, 2);              // Write string
       delay2S();
 end;
end.
```

HW Connection



Glcd HW connection

KEYPAD LIBRARY

The mikroPascal PRO for AVR provides a library for working with 4x4 keypad. The library routines can also be used with 4x1, 4x2, or 4x3 keypad. For connections explanation see schematic at the bottom of this page.

Note: Since sampling lines for AVR MCUs are activated by logical zero Keypad Library can not be used with hardwares that have protective diodes connected with anode to MCU side, such as mikroElektronika's Keypad extra board HW.Rev v1.20

External dependencies of Keypad Library

The following variables must be defined in all projects using Sound Library:	Description:	Example :
`var keypadPort : byte; sfr; external;`	Keypad Port.	`var keypadPort : byte at PORTB;`
`var keypadPort_Direction : byte; sfr; external;`	Direction of the Keypad Port.	`var keypadPort_Direction : byte at DDRB;`

Library Routines

- Keypad_Init
- Keypad_Key_Press
- Keypad_Key_Click

Keypad_Init

Prototype	`procedure Keypad_Init();`
Returns	Nothing.
Description	Initializes port for working with keypad.
Requires	Global variables : - `keypadPort_Reg` - Keypad port - `keypadPort_Reg_Direction` - Direction of the Keypad port must be defined before using this function.

Example	`// Initialize PORTB for communication with keypad`
	`var keypadPort : byte at PORTB;`
	`var keypadPort_Direction : byte at DDRB;`
	`...`
	`Keypad_Init();`

Keypad_Key_Press

Prototype	`function Keypad_Key_Press(): byte;`
Returns	The code of a pressed key (1..16). If no key is pressed, returns 0.
Description	Reads the key from keypad when key gets pressed.
Requires	Port needs to be initialized for working with the Keypad library, see Keypad_Init.

Example	`var kp : byte;`
	`...`
	`kp := Keypad_Key_Press();`

Keypad_Key_Click

Prototype	`function Keypad_Key_Click(): byte;`
Returns	The code of a clicked key (1..16). If no key is clicked, returns 0.
Description	Call to Keypad_Key_Click is a blocking call: the function waits until some key is pressed and released. When released, the function returns 1 to 16, depending on the key. If more than one key is pressed simultaneously the function will wait until all pressed keys are released. After that the function will return the code of the first pressed key.
Requires	Port needs to be initialized for working with the Keypad library, see Keypad_Init.

Example	`var kp : byte;`
	`...`
	`kp := Keypad_Key_Click();`

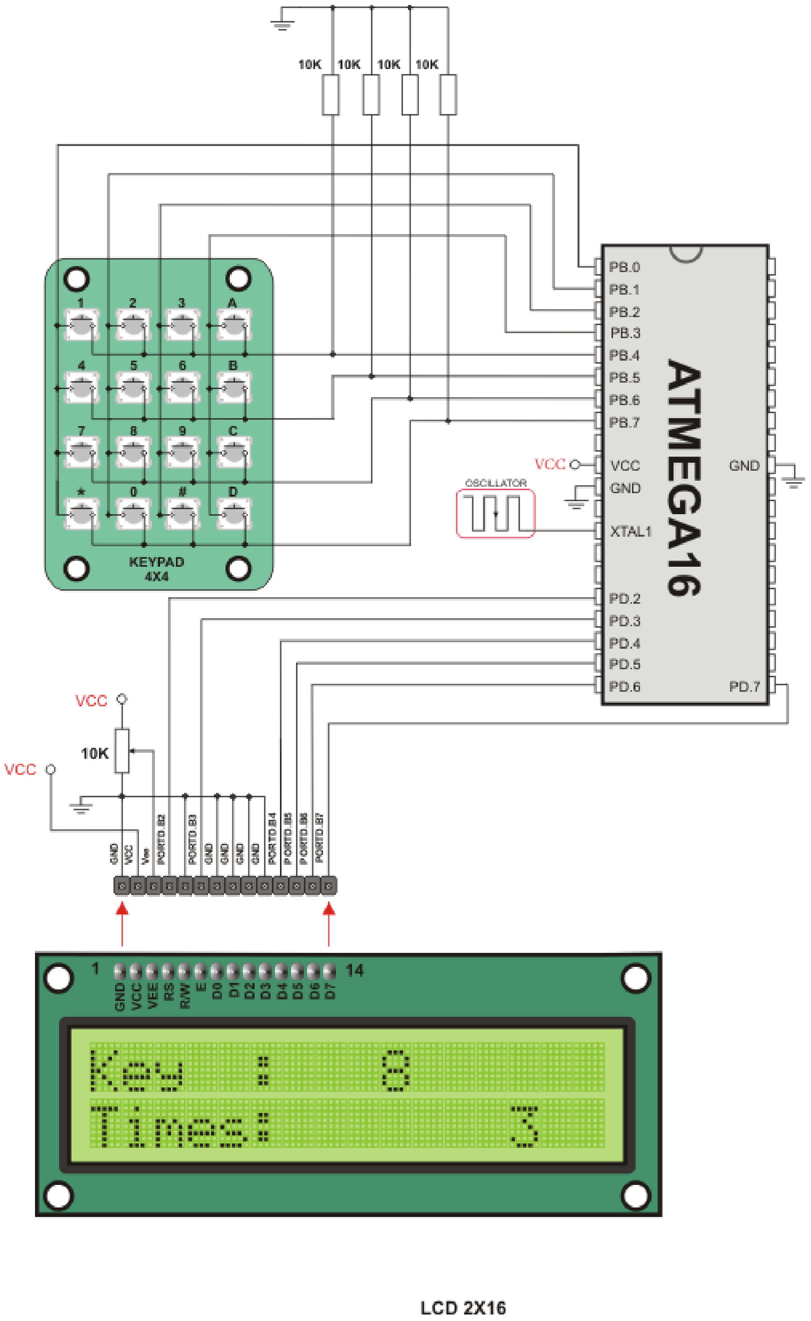
Library Example

This is a simple example of using the Keypad Library. It supports keypads with 1..4 rows and 1..4 columns. The code being returned by Keypad_Key_Click() function is in range from 1..16. In this example, the code returned is transformed into ASCII codes [0..9,A..F] and displayed on Lcd. In addition, a small single-byte counter displays in the second Lcd row number of key presses.

```
program Keypad_Test;
var kp, cnt, oldstate : byte;
    txt : array[ 6]  of byte;

// Keypad module connections
var keypadPort : byte at PORTB;
var keypadPort_Direction : byte at DDRB;
// End Keypad module connections

// Lcd pinout definition
var LCD_RS : sbit at PORTD.B2;
    LCD_EN : sbit at PORTD.B3;
    LCD_D4 : sbit at PORTD.B4;
    LCD_D5 : sbit at PORTD.B5;
    LCD_D6 : sbit at PORTD.B6;
    LCD_D7 : sbit at PORTD.B7;

var LCD_RS_Direction : sbit at DDRD.B2;
    LCD_EN_Direction : sbit at DDRD.B3;
    LCD_D4_Direction : sbit at DDRD.B4;
    LCD_D5_Direction : sbit at DDRD.B5;
    LCD_D6_Direction : sbit at DDRD.B6;
    LCD_D7_Direction : sbit at DDRD.B7;
// end Lcd pinout definitions

 begin
    oldstate := 0;
    cnt := 0;                          // Reset counter
    Keypad_Init();                     // Initialize Keypad
    Lcd_Init();                        // Initialize Lcd
    Lcd_Cmd(LCD_CLEAR);                // Clear display
    Lcd_Cmd(LCD_CURSOR_OFF);           // Cursor off
    Lcd_Out(1, 1, 'Key  :');           // Write message text on Lcd
    Lcd_Out(2, 1, 'Times:');

 while TRUE do
      begin
         kp := 0;                      // Reset key code variable

         // Wait for key to be pressed and released
         while ( kp = 0 ) do
   kp := Keypad_Key_Click();       // Store key code in kp variable
      // Prepare value for output, transform key to it's ASCII value
         case kp of
          //case 10: kp = 42;   // '*'   // Uncomment this block for
keypad4x3
           //case 11: kp = 48;   // '0'
         //case 12: kp = 35;   // '#'
           //default: kp += 48;
```

```
            1: kp := 49;  // 1             // Uncomment this block for keypad4x4
                    2: kp := 50;  // 2
                    3: kp := 51;  // 3
                    4: kp := 65;  // A
                    5: kp := 52;  // 4
                    6: kp := 53;  // 5
                    7: kp := 54;  // 6
                    8: kp := 66;  // B
                    9: kp := 55;  // 7
                   10: kp := 56;  // 8
                   11: kp := 57;  // 9
                   12: kp := 67;  // C
                   13: kp := 42;  // *
                   14: kp := 48;  // 0
                   15: kp := 35;  // #
                   16: kp := 68;  // D

                 end;

                  if (kp <> oldstate) then      // Pressed key differs from
            previous
                   begin
                     cnt := 1;
                     oldstate := kp;
                   end
                 else                       // Pressed key is same as previous
                   Inc(cnt);

                 Lcd_Chr(1, 10, kp);       // Print key ASCII value on Lcd

                 if (cnt = 255) then      // If counter variablble overflow
                   begin
                     cnt := 0;
                     Lcd_Out(2, 10, '      ');
                   end;

                WordToStr(cnt, txt);     // Transform counter value to string


                 Lcd_Out(2, 10, txt);    // Display counter value on Lcd
              end;
             end.
```

HW Connection



4x4 Keypad connection scheme

LCD LIBRARY

The mikroPascal PRO for AVR provides a library for communication with Lcds (with HD44780 compliant controllers) through the 4-bit interface. An example of Lcd connections is given on the schematic at the bottom of this page.

For creating a set of custom Lcd characters use Lcd Custom Character Tool.

External dependencies of Lcd Library

The following variables must be defined in all projects using Lcd Library :	Description:	Example :
`var LCD_RS : sbit; sfr; external;`	Register Select line.	`var LCD_RS : sbit at PORTD.B2;`
`var LCD_EN : sbit; sfr; external;`	Enable line.	`var LCD_EN : sbit at PORTD.B3;`
`var LCD_D7 : sbit; sfr; external;`	Data 7 line.	`var LCD_D7 : sbit at PORTD.B4;`
`var LCD_D6 : sbit; sfr; external;`	Data 6 line.	`var LCD_D6 : sbit at PORTD.B5;`
`var LCD_D5 : sbit; sfr; external;`	Data 5 line.	`var LCD_D5 : sbit at PORTD.B6;`
`var LCD_D4 : sbit; sfr; external;`	Data 4 line.	`var LCD_D4 : sbit at PORTD.B7;`
`var LCD_RS_Direction : sbit; sfr; external;`	Register Select direction pin.	`var LCD_RS_Direction : sbit at DDRD.B2;`
`var LCD_EN_Direction : sbit; sfr; external;`	Enable direction pin.	`var LCD_EN_Direction : sbit at DDRD.B3;`
`var LCD_D7_Direction : sbit; sfr; external;`	Data 7 direction pin.	`var LCD_D7_Direction : sbit at DDRD.B4;`
`var LCD_D6_Direction : sbit; sfr; external;`	Data 6 direction pin.	`var LCD_D6_Direction : sbit at DDRD.B5;`
`var LCD_D5_Direction : sbit; sfr; external;`	Data 5 direction pin.	`var LCD_D5_Direction : sbit at DDRD.B6;`
`var LCD_D4_Direction : sbit; sfr; external;`	Data 4 direction pin.	`var LCD_D4_Direction : sbit at DDRD.B7;`

Library Routines

- Lcd_Init
- Lcd_Out
- Lcd_Out_Cp
- Lcd_Chr
- Lcd_Chr_Cp
- Lcd_Cmd

Lcd_Init

Prototype	`procedure Lcd_Init()`
Returns	Nothing.
Description	Initializes Lcd module.
Requires	Global variables: - `LCD_D7`: Data bit 7 - `LCD_D6`: Data bit 6 - `LCD_D5`: Data bit 5 - `LCD_D4`: Data bit 4 - `LCD_RS`: Register Select (data/instruction) signal pin - `LCD_EN`: Enable signal pin - `LCD_D7_Direction`: Direction of the Data 7 pin - `LCD_D6_Direction`: Direction of the Data 6 pin - `LCD_D5_Direction`: Direction of the Data 5 pin - `LCD_D4_Direction`: Direction of the Data 4 pin - `LCD_RS_Direction`: Direction of the Register Select pin - `LCD_EN_Direction`: Direction of the Enable signal pin must be defined before using this function.

Example	`// Lcd module connections`
	`var LCD_RS : sbit at PORTD.B2;`
	`var LCD_EN : sbit at PORTD.B3;`
	`var LCD_D4 : sbit at PORTD.B4;`
	`var LCD_D5 : sbit at PORTD.B5;`
	`var LCD_D6 : sbit at PORTD.B6;`
	`var LCD_D7 : sbit at PORTD.B7;`
	
	`var LCD_RS_Direction : sbit at DDRD.B2;`
	`var LCD_EN_Direction : sbit at DDRD.B3;`
	`var LCD_D4_Direction : sbit at DDRD.B4;`
	`var LCD_D5_Direction : sbit at DDRD.B5;`
	`var LCD_D6_Direction : sbit at DDRD.B6;`
	`var LCD_D7_Direction : sbit at DDRD.B7;`
	`// End Lcd module connections`
	
	`...`
	
	`Lcd_Init();`

Lcd_Out

Prototype	`procedure Lcd_Out(row: byte; column: byte; var text: array [20] of char);`
Returns	Nothing.
Description	Prints text on Lcd starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.

Example	`// Write text "Hello!" on Lcd starting from row 1, column 3:`
	`Lcd_Out(1, 3, "Hello!");`

Lcd_Out_Cp

Prototype	`procedure Lcd_Out_Cp(var text: array [20] of char);`
Returns	Nothing.
Description	Prints text on Lcd at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.

Example	`// Write text "Here!" at current cursor position:`
	`Lcd_Out_Cp("Here!");`

Lcd_Chr

Prototype	`procedure Lcd_Chr(row: byte; column: byte; out_char: byte);`
Returns	Nothing.
Description	Prints character on Lcd at specified position. Both variables and literals can be passed as a character. Parameters : - `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.

Example	`// Write character "i" at row 2, column 3:`
	`Lcd_Chr(2, 3, 'i');`

Lcd_Chr_Cp

Prototype	`procedure Lcd_Chr_Cp(out_char: byte);`
Returns	Nothing.
Description	Prints character on Lcd at current cursor position. Both variables and literals can be passed as a character. Parameters : - `out_char`: character to be written
Requires	The Lcd module needs to be initialized. See Lcd_Init routine.

Example	`// Write character "e" at current cursor position:`
	`Lcd_Chr_Cp('e');`

Lcd_Cmd

Prototype	`procedure Lcd_Cmd(out_char: byte);`
Returns	Nothing.
Description	Sends command to Lcd. Parameters : - `out_char`: command to be sent Note: Predefined constants can be passed to the function, see Available Lcd Commands.
Requires	The Lcd module needs to be initialized. See Lcd_Init table.

Example	`// Clear Lcd display:`
	`Lcd_Cmd(LCD_CLEAR);`

Available Lcd Commands

Lcd Command	Purpose
`LCD_FIRST_ROW`	Move cursor to the 1st row
`LCD_SECOND_ROW`	Move cursor to the 2nd row
`LCD_THIRD_ROW`	Move cursor to the 3rd row
`LCD_FOURTH_ROW`	Move cursor to the 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to its 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

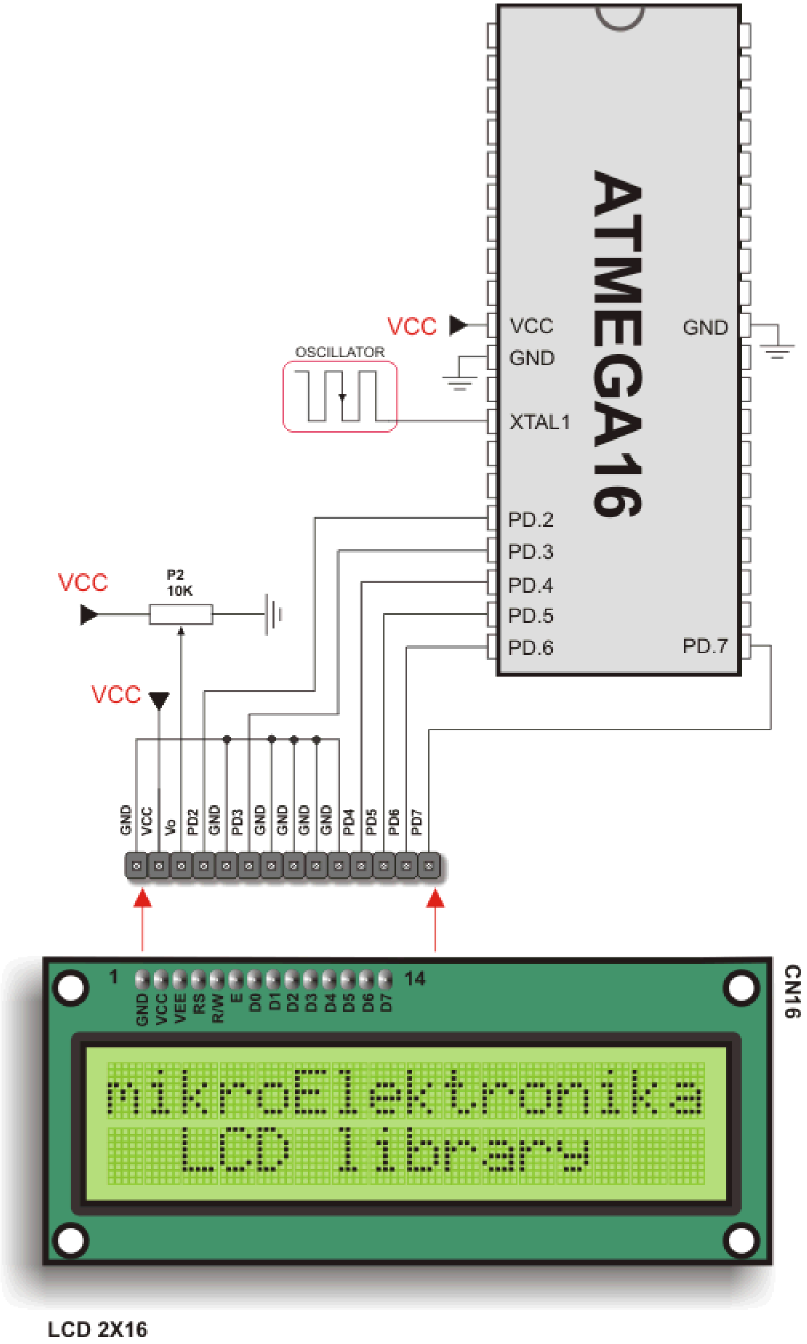
Library Example

The following code demonstrates usage of the Lcd Library routines:

```
program  Lcd_Test;

// Lcd module connections
var LCD_RS : sbit at PORTD.B2;
var LCD_EN : sbit at PORTD.B3;
var LCD_D4 : sbit at PORTD.B4;
var LCD_D5 : sbit at PORTD.B5;
var LCD_D6 : sbit at PORTD.B6;
var LCD_D7 : sbit at PORTD.B7;

var LCD_RS_Direction : sbit at DDRD.B2;
var LCD_EN_Direction : sbit at DDRD.B3;
var LCD_D4_Direction : sbit at DDRD.B4;
var LCD_D5_Direction : sbit at DDRD.B5;
var LCD_D6_Direction : sbit at DDRD.B6;
var LCD_D7_Direction : sbit at DDRD.B7;
// End Lcd module connections

var txt1 : array[ 16]  of char;
    txt2 : array[ 9]   of char;
    txt3 : array[ 8]   of char;
    txt4 : array[ 7]   of char;
    i    : byte;             // Loop variable

procedure Move_Delay();      // Function used for text moving
  begin
    Delay_ms(500);           // You can change the moving speed here
  end;

  begin

    txt1 := 'mikroElektronika';
    txt2 := 'EasyAVR5A';
    txt3 := 'Lcd4bit';
    txt4 := 'example';
    Lcd_Init();                        // Initialize Lcd
    Lcd_Cmd(LCD_CLEAR);                // Clear display
    Lcd_Cmd(LCD_CURSOR_OFF);           // Cursor off
    LCD_Out(1,6,txt3);                 // Write text in first row
    LCD_Out(2,6,txt4);                 // Write text in second row
    Delay_ms(2000);
    Lcd_Cmd(LCD_CLEAR);                // Clear display

    LCD_Out(1,1,txt1);                 // Write text in first row
    LCD_Out(2,4,txt2);                 // Write text in second row
    Delay_ms(500);
```

```
// Moving text
    for i:=0 to 3 do                // Move text to the right 4 times
      begin
        Lcd_Cmd(LCD_SHIFT_RIGHT);
        Move_Delay();
      end;

    while TRUE do                   // Endless loop
      begin
        for i:=0 to 6 do             // Move text to the left 7 times
          begin
            Lcd_Cmd(LCD_SHIFT_LEFT);
            Move_Delay();
          end;

        for i:=0 to 6 do            // Move text to the right 7 times
          begin
            Lcd_Cmd(LCD_SHIFT_RIGHT);
            Move_Delay();
          end;

      end;
  end.
```

HW connection

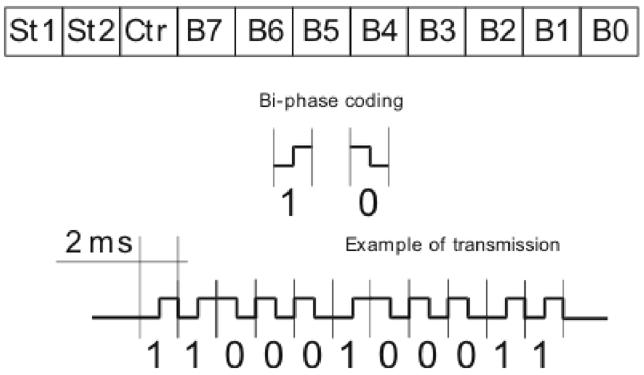


Lcd HW connection

MANCHESTER CODE LIBRARY

The mikroPascal PRO for AVR provides a library for handling Manchester coded signals. The Manchester code is a code in which data and clock signals are combined to form a single self-synchronizing data stream; each encoded bit contains a transition at the midpoint of a bit period, the direction of transition determines whether the bit is 0 or 1; the second half is the true bit value and the first half is the complement of the true bit value (as shown in the figure below).

Manchester RF_Send_Byte format



Notes: The Manchester receive routines are blocking calls (Man_Receive_Init and Man_Synchro). This means that MCU will wait until the task has been performed (e.g. byte is received, synchronization achieved, etc).

Note: Manchester code library implements time-based activities, so interrupts need to be disabled when using it.

External dependencies of Manchester Code Library

The following variables must be defined in all projects using Sound Library:	Description:	Example :
`var MANRXPIN : sbit; sfr; external;`	Receive line.	`var MANRXPIN : sbit at PINB.B0;`
`var MANTXPIN : sbit; sfr; external;`	Transmit line.	`var MANTXPIN : sbit at PORTB.B1;`
`var MANRXPIN_Direction : sbit; sfr; external;`	Direction of the Receive pin.	`var MANRXPIN_Direction : sbit at DDRB.B0;`
`var MANTXPIN_Direction : sbit; sfr; external;`	Direction of the Transmit pin.	`var MANTXPIN_Direction : sbit at DDRB.B1;`

Library Routines

- Man_Receive_Init
- Man_Receive
- Man_Send_Init
- Man_Send
- Man_Synchro
- Man_Break

The following routines are for the internal use by compiler only:

- Manchester_0
- Manchester_1
- Manchester_Out

Man_Receive_Init

Prototype	`function Man_Receive_Init(): word;`
Returns	- `0` - if initialization and synchronization were successful. - `1` - upon unsuccessful synchronization.
Description	The function configures Receiver pin and performs synchronization procedure in order to retrieve baud rate out of the incoming signal. **Note:** In case of multiple persistent errors on reception, the user should call this routine once again or Man_Synchro routine to enable synchronization.
Requires	Global variables : - `MANRXPIN` : Receive line - `MANRXPIN_Direction` : Direction of the receive pin must be defined before using this function.

Example	`// Initialize Receiver`
	`var MANRXPIN : sbit at PINB.B0;`
	`var MANRXPIN_Direction : sbit at DDRB.B0;`
	`...`
	`Man_Receive_Init();`

Man_Receive

Prototype	function Man_Receive(var error: byte): byte;
Returns	A byte read from the incoming signal.
Description	The function extracts one byte from incoming signal. Parameters : - error: error flag. If signal format does not match the expected, the error flag will be set to non-zero.
Requires	To use this function, the user must prepare the MCU for receiving. See Man_Receive_Init.

Example	var data, error : byte
	...
	data := 0
	error := 0
	data := Man_Receive(&error);
	
	if (error <> 0) then
	  begin
	      // error handling
	  end;

Man_Send_Init

Prototype	procedure Man_Send_Init();
Returns	Nothing.
Description	The function configures Transmitter pin.
Requires	Global variables : - MANRXPIN : Receive line - MANRXPIN_Direction : Direction of the receive pin must be defined before using this function.

Example	// Initialize Transmitter:
	var MANTXPIN : sbit at PINB.B1;
	var MANTXPIN_Direction : sbit at DDRB.B1;
	...
	Man_Send_Init();

Man_Send

Prototype	`procedure Man_Send(tr_data: byte);`
Returns	Nothing.
Description	Sends one byte. Parameters : - `tr_data`: data to be sent **Note:** Baud rate used is 500 bps.
Requires	To use this function, the user must prepare the MCU for sending. See Man_Send_Init.

Example	`var msg : byte;`
	`...`
	`Man_Send(msg);`

Man_Synchro

Prototype	`function Man_Synchro(): word;`
Returns	- `0` - if synchronization was not successful. - Half of the manchester bit length, given in multiples of 10us - upon successful synchronization.
Description	Measures half of the manchester bit length with 10us resolution.
Requires	To use this function, you must first prepare the MCU for receiving. See Man_Receive_Init.

Example	`var man__half_bit_len : word ;`
	`...`
	`man__half_bit_len := Man_Synchro();`

Man_Break

Prototype	`procedure Man_Break();`
Returns	Nothing.
Description	Man_Receive is blocking routine and it can block the program flow. Call this routine from interrupt to unblock the program execution. This mechanism is similar to WDT. **Note:** Interrupts should be disabled before using Manchester routines again (see note at the top of this page).
Requires	Nothing.

Example	`var data1, error, counter : byte;`
	
	`procedure Timer0Overflow_ISR(); org 0x12;`
	`begin`
	`  counter := 0;`
	`  if (counter >= 20) then`
	`  begin`
	`    Man_Break();`
	`    counter := 0;                  // reset counter`
	`  end`
	`  else`
	`    Inc(counter);                  // increment counter`
	`end;`
	
	`begin`
	`  TOIE0_bit  := 1;           // Timer0 overflow interrupt enable`
	`  TCCR0_bit  := 5;           // Start timer with 1024 prescaler`
	
	`  SREG_I_bit := 0;                   // Interrupt disable`
	
	`  ...`
	
	`  Man_Receive_Init();`
	
	`  ...`
	
	`  // try Man_Receive with blocking prevention mechanism`
	
	`  SREG_I_bit := 1;                   // Interrupt enable`
	`  data1 := Man_Receive(@error);`
	`  SREG_I_bit := 0;                   // Interrupt disable`
	
	`  ...`
	
	`end;`

Library Example

The following code is code for the Manchester receiver, it shows how to use the Manchester Library for receiving data:

```
program Manchester_Receiver;

// Lcd module connections
var LCD_RS : sbit at PORTD.B2;
    LCD_EN : sbit at PORTD.B3;
    LCD_D4 : sbit at PORTD.B4;
    LCD_D5 : sbit at PORTD.B5;
    LCD_D6 : sbit at PORTD.B6;
    LCD_D7 : sbit at PORTD.B7;

var LCD_RS_Direction : sbit at DDRD.B2;
    LCD_EN_Direction : sbit at DDRD.B3;
    LCD_D4_Direction : sbit at DDRD.B4;
    LCD_D5_Direction : sbit at DDRD.B5;
    LCD_D6_Direction : sbit at DDRD.B6;
    LCD_D7_Direction : sbit at DDRD.B7;
// End Lcd module connections

// Manchester module connections
var MANRXPIN : sbit at PINB.B0;
    MANRXPIN_Direction : sbit at DDRB.B0;
    MANTXPIN : sbit at PORTB.B1;
    MANTXPIN_Direction : sbit at DDRB.B1;
// End Manchester module connections

var error, ErrorCount, temp : byte;

  begin
    ErrorCount := 0;
    Delay_10us();
    Lcd_Init();                              // Initialize Lcd
    Lcd_Cmd(LCD_CLEAR);                      // Clear Lcd display

    Man_Receive_Init();                      // Initialize Receiver

    while TRUE do                            // Endless loop
      begin
        Lcd_Cmd(LCD_FIRST_ROW);         // Move cursor to the 1st row

         while TRUE do                 // Wait for the "start" byte
           begin
temp := Man_Receive(error);            // Attempt byte receive

 if (temp = 0x0B) then      // "Start" byte, see Transmitter example
```

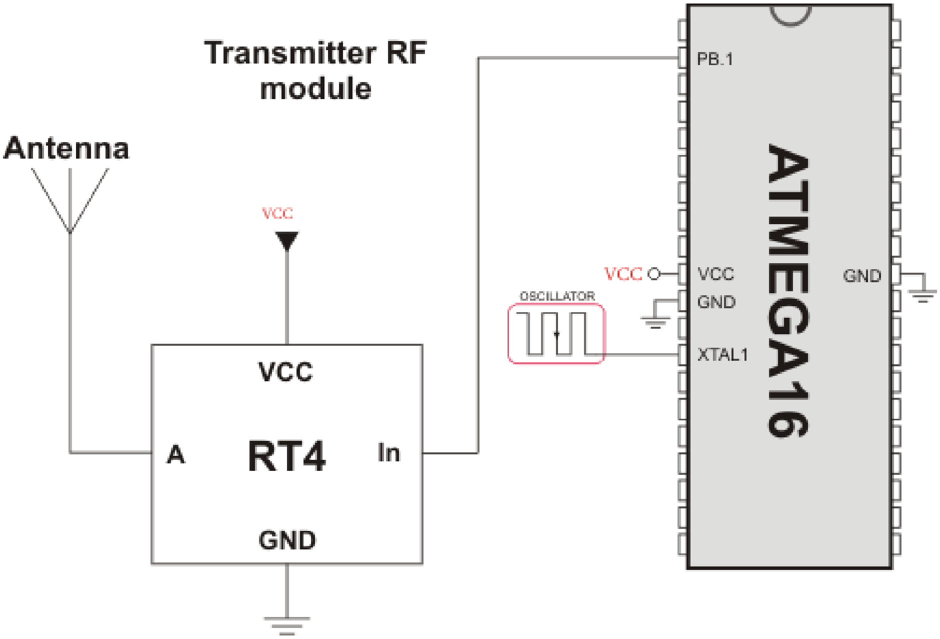
```
               break;                  // We got the starting sequence
            if (error <> 0) then       // Exit so we do not loop forever
               break;
           end;

         repeat
           begin
             temp := Man_Receive(error);      // Attempt byte receive
             if (error <> 0) then             // If error occured
               begin
                 Lcd_Chr_CP('?');       // Write question mark on Lcd
                  Inc(ErrorCount);      // Update error counter
                  if (ErrorCount > 20) then     // In case of multi-
ple errors
                    begin
                      temp := Man_Synchro();    // Try to synchronize
again
                     //Man_Receive_Init();    // Alternative, try to
Initialize Receiver again
                      ErrorCount := 0;        // Reset error counter
                    end;
               end
             else                               // No error occured
               begin
                  if (temp <> 0x0E) then        // If "End" byte was
received(see Transmitter example)
                  Lcd_Chr_CP(temp);         //  do not write received
byte on Lcd
               end;
             Delay_ms(25);
           end;
         until ( temp = 0x0E );
      end;               // If "End" byte was received exit do loop
  end.
```

The following code is code for the Manchester transmitter, it shows how to use the Manchester Library for transmitting data:

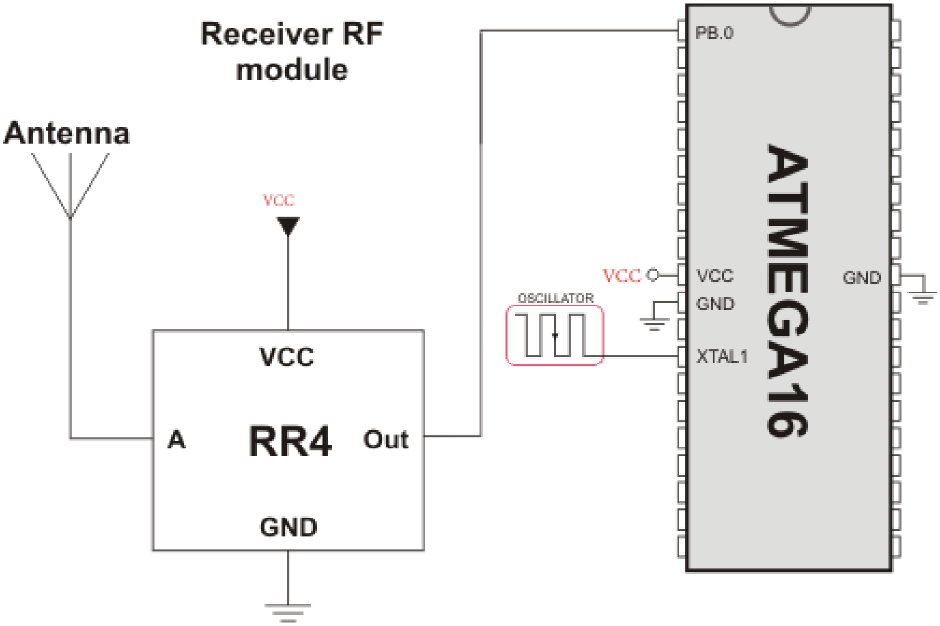
```
program Manchester_Transmitter;

// Manchester module connections
var MANRXPIN : sbit at PORTB.B0;
    MANRXPIN_Direction : sbit at DDRB.B0;
    MANTXPIN : sbit at PORTB.B1;
    MANTXPIN_Direction : sbit at DDRB.B1;
// End Manchester module connections

var index, character : byte;
    s1 : array[ 17] of char;
```

```
begin
   s1 := 'mikroElektronika';
   Man_Send_Init();                    // Initialize transmitter

   while TRUE do                       // Endless loop
     begin
        Man_Send(0x0B);                // Send "start" byte
        Delay_ms(100);                 // Wait for a while

        character := s1[ 0] ;          // Take first char from string
        index := 0;                // Initialize index variable
        while (character <> 0) do      // String ends with zero
          begin
            Man_Send(character);       // Send character
            Delay_ms(90);              // Wait for a while
            Inc(index);                 // Increment index variable
            character := s1[ index] ; // Take next char from string
          end;
        Man_Send(0x0E);             // Send "end" byte
        Delay_ms(1000);
     end;
 end.
```

Connection Example



Simple Transmitter connection



Simple Receiver 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 PRO for AVR 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:

- Routines for file handling can be used only with FAT16 file system.
- 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.
- Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

External dependencies of MMC Library

The following variables must be defined in all projects using Sound Library:	Description:	Example :
`var Mmc_Chip_Select : sbit; sfr; external;`	Chip select pin.	`var Mmc_Chip_Select : sbit at PINB.B0;`
`var Mmc_Chip_Select_Direction : sbit; sfr; external;`	Direction of the chip select pin.	`var Mmc_Chip_Select_Direction : sbit at DDRB.B0;`

Library Routines

- Mmc_Init
- Mmc_Read_Sector
- Mmc_Write_Sector
- Mmc_Read_Cid
- Mmc_Read_Csd

Routines for file handling:

- Mmc_Fat_Init
- Mmc_Fat_QuickFormat
- Mmc_Fat_Assign
- Mmc_Fat_Reset
- Mmc_Fat_Read
- Mmc_Fat_Rewrite
- Mmc_Fat_Append
- Mmc_Fat_Delete
- 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(): byte;`
Returns	- `0` - if MMC/SD card was detected and successfully initialized - `1` - otherwise
Description	Initializes MMC through hardware SPI interface. Parameters: - `port`: chip select signal port address. - `cspin`: chip select pin.
Requires	Global variables : - `Mmc_Chip_Select`: Chip Select line - `Mmc_Chip_Select_Direction`: Direction of the Chip Select pin must be defined before using this function. The appropriate hardware SPI module must be previously initialized. See the SPI1_Init, SPI1_Init_Advanced routines.

Example	`// MMC module connections`
	`var Mmc_Chip_Select : sbit; sfr; at PORTB.B2;`
	`var Mmc_Chip_Select_Direction : sbit; sfr; at DDRB.B2;`
	`// MMC module connections`
	
	`error = Mmc_Init();  // Init with CS line at PORTB.B2`
	`var i : byte;`
	`...`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV2, _SPI_CLK_LO_LEADING);`
	`Spi_Rd_Ptr := @SPI1_Read;    // Pass pointer to SPI Read function of used SPI module`
	`i = Mmc_Init();`

Mmc_Read_Sector

Prototype	function Mmc_Read_Sector(sector: longint; var dbuff: array[512] of byte): byte;
Returns	- 0 - if reading was successful - 1 - if an error occurred
Description	The function reads one sector (512 bytes) from MMC card. Parameters: - sector: MMC/SD card sector to be read. - data: buffer of minimum 512 bytes in length for data storage.
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	// read sector 510 of the MMC/SD card
	var error : byte;
	    sectorNo : longint;
	    dataBuffer : array[512] of byte;
	...
	sectorNo := 510;
	error := Mmc_Read_Sector(sectorNo, dataBuffer);

Mmc_Write_Sector

Prototype	function Mmc_Write_Sector(sector: longint; var data_: array[512] of byte): byte;
Returns	- 0 - if writing was successful - 1 - if there was an error in sending write command - 2 - if there was an error in writing (data rejected)
Description	The function writes 512 bytes of data to one MMC card sector. Parameters: - sector: MMC/SD card sector to be written to. - data: data to be written (buffer of minimum 512 bytes in length).
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	// write to sector 510 of the MMC/SD card
	var error : byte;
	    sectorNo : longint;
	    dataBuffer : array[512] of byte;
	...
	sectorNo := 510;
	error := Mmc_Write_Sector(sectorNo, dataBuffer);

Mmc_Read_Cid

Prototype	`function Mmc_Read_Cid(var data_cid: array[16] of byte): byte;`
Returns	- `0` - if CID register was read successfully - `1` - if there was an error while reading
Description	The function reads 16-byte CID register. Parameters: - `data_cid`: buffer of minimum 16 bytes in length for storing CID register content.
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`var error : byte;`
	`    dataBuffer : array[16] of byte;`
	`...`
	`error := Mmc_Read_Cid(dataBuffer);`

Mmc_Read_Csd

Prototype	`function Mmc_Read_Csd(var data_for_registers: array[16] of byte): byte;`
Returns	- `0` - if CSD register was read successfully - `1` - if there was an error while reading
Description	The function reads 16-byte CSD register. Parameters: - `data_for_registers`: buffer of minimum 16 bytes in length for storing CSD register content.
Requires	MMC/SD card must be initialized. See Mmc_Init.

Example	`var error : word;`
	`    data_for_registers : array[16] of byte;`
	`...`
	`error := Mmc_Read_Csd(data_for_registers);`

Mmc_Fat_Init

Prototype	`function Mmc_Fat_Init(): byte;`
Returns	- `0` - if MMC/SD card was detected and successfully initialized - `1` - if FAT16 boot sector was not found - `255` - if MMC/SD card was not detected
Description	Initializes MMC/SD card, reads MMC/SD FAT16 boot sector and extracts necessary data needed by the library. **Note**: MMC/SD card has to be formatted to FAT16 file system.
Requires	Global variables : - `Mmc_Chip_Select`: Chip Select line - `Mmc_Chip_Select_Direction`: Direction of the Chip Select pin must be defined before using this function. The appropriate hardware SPI module must be previously initialized. See the SPI1_Init, SPI1_Init_Advanced routines.

Example	`// init the FAT library`
	
	`if (Mmc_Fat_Init() = 0) then`
	`  begin`
	`  ...`
	`  end`

Mmc_Fat_QuickFormat

Prototype	`function Mmc_Fat_QuickFormat(var port : word; pin : word; var mmc_fat_label : string[11]) : byte;`
Returns	- `0` - if MMC/SD card was detected, successfully formated and initialized - `1` - if FAT16 format was unseccessful - `255` - if MMC/SD card was not detected
Description	Formats to FAT16 and initializes MMC/SD card. Parameters: - `port`: chip select signal port address. - `pin`: chip select pin. - `mmc_fat_label`: volume label (11 characters in length). If less than 11 charac ters are provided, the label will be padded with spaces. If an empty string is passed, the volume will not be labeled. **Note**: This routine can be used instead or in conjunction with the Mmc_Fat_Init routine. **Note**: If MMC/SD card already contains a valid boot sector, it will remain unchanged (except volume label field) and only FAT and ROOT tables will be erased. Also, the new volume label will be set.
Requires	The appropriate hardware SPI module must be previously initialized.

Example	`// format and initialize the FAT library`
	`if (Mmc_Fat_QuickFormat('mikroE') = 0) then`
	`begin`
	`   ...`
	`end;`

Mmc_Fat_Assign

<table>
<tr><td><b>Prototype</b></td><td><code>function Mmc_Fat_Assign(var filename: array[12] of char; file_cre_attr: byte): byte;</code></td></tr>
<tr><td><b>Returns</b></td><td>- <code>1</code> - if file already exists or file does not exist but a new file is created.
- <code>0</code> - if file does not exist and no new file is created.</td></tr>
<tr><td><b>Description</b></td><td>Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied on an assigned file.
Parameters:
- <code>filename</code>: name of the file that should be assigned for file operations. File name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case last 3 characters of the string are considered to be file extension.
- <code>file_cre_attr</code>: file creation and attributs flags. Each bit corresponds to the appropriate file attribut:
<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th></tr>
<tr><td>0</td><td>0x01</td><td>Read Only</td></tr>
<tr><td>1</td><td>0x02</td><td>Hidden</td></tr>
<tr><td>2</td><td>0x04</td><td>System</td></tr>
<tr><td>3</td><td>0x08</td><td>Volume Label</td></tr>
<tr><td>4</td><td>0x10</td><td>Subdirectory</td></tr>
<tr><td>5</td><td>0x20</td><td>Archive</td></tr>
<tr><td>6</td><td>0x40</td><td>Device (internal use only, never found on disk)</td></tr>
<tr><td>7</td><td>0x80</td><td>File creation flag. If the file does not exist and this flag is set, a new file with specified name will be created.</td></tr>
</table>
<b>Note:</b> Long File Names (LFN) are not supported.</td></tr>
<tr><td><b>Requires</b></td><td>MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init.</td></tr>
<tr><td><b>Example</b></td><td><code>// create file with archive attribut if it does not already exist</code>
<code>Mmc_Fat_Assign('MIKRO007.TXT',0xA0);</code></td></tr>
</table>

Mmc_Fat_Reset

Prototype	`procedure Mmc_Fat_Reset(var size: dword);`
Returns	Nothing.
Description	Opens currently assigned file for reading. Parameters: - `size`: buffer to store file size to. After file has been open for reading, its size is returned through this parameter.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`var size : dword;`
	`...`
	`Mmc_Fat_Reset(size);`

Mmc_Fat_Read

Prototype	`procedure Mmc_Fat_Read(var bdata: byte);`
Returns	Nothing.
Description	Reads a byte from the currently assigned file opened for reading. Upon function execution, file pointers will be set to the next character in the file. Parameters: - `bdata`: buffer to store read byte to. Upon this function execution read byte is returned through this parameter.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for reading. See Mmc_Fat_Reset.

Example	`var character : byte;`
	`...`
	`Mmc_Fat_Read(character);`

Mmc_Fat_Rewrite

Prototype	`procedure Mmc_Fat_Rewrite();`
Returns	Nothing.
Description	Opens the currently assigned file for writing. If the file is not empty its content will be erased.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`// open file for writing`
	`Mmc_Fat_Rewrite();`

Mmc_Fat_Append

Prototype	`procedure Mmc_Fat_Append();`
Returns	Nothing.
Description	Opens the currently assigned file for appending. Upon this function execution file pointers will be positioned after the last byte in the file, so any subsequent file writing operation will start from there.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`// open file for appending`
	`Mmc_Fat_Append();`

Mmc_Fat_Delete

Prototype	`procedure Mmc_Fat_Delete();`
Returns	Nothing.
Description	Deletes currently assigned file from MMC/SD card.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`// delete current file`
	`Mmc_Fat_Delete();`

Mmc_Fat_Write

Prototype	`procedure Mmc_Fat_Write(var fdata: array[512] of byte; data_len: word);`
Returns	Nothing.
Description	Writes requested number of bytes to the currently assigned file opened for writing. Parameters: - `fdata`: data to be written. - `data_len`: number of bytes to be written.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for writing. See Mmc_Fat_Rewrite or Mmc_Fat_Append.

Example	`var file_contents : array[42] of byte;`
	`...`
	`Mmc_Fat_Write(file_contents, 42); // write data to the assigned file`

Mmc_Fat_Set_File_Date

Prototype	`procedure Mmc_Fat_Set_File_Date(year: word; month: byte; day: byte; hours: byte; mins: byte; seconds: byte);`
Returns	Nothing.
Description	Sets the date/time stamp. Any subsequent file writing operation will write this stamp to the currently assigned file's time/date attributs. Parameters: - `year`: year attribute. Valid values: 1980-2107 - `month`: month attribute. Valid values: 1-12 - `day`: day attribute. Valid values: 1-31 - `hours`: hours attribute. Valid values: 0-23 - `mins`: minutes attribute. Valid values: 0-59 - `seconds`: seconds attribute. Valid values: 0-59
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign. The file must be opened for writing. See Mmc_Fat_Rewrite or Mmc_Fat_Append.
Example	`Mmc_Fat_Set_File_Date(2005,9,30,17,41,0);`

Mmc_Fat_Get_File_Date

Prototype	`procedure Mmc_Fat_Get_File_Date(var year: word; var month: byte; var day: byte; var hours: byte; var mins: byte);`
Returns	Nothing.
Description	Reads time/date attributes of the currently assigned file. Parameters: - `year`: buffer to store year attribute to. Upon function execution year attribute is returned through this parameter. - `month`: buffer to store month attribute to. Upon function execution month attrib ute is returned through this parameter. - `day`: buffer to store day attribute to. Upon function execution day attribute is returned through this parameter. - `hours`: buffer to store hours attribute to. Upon function execution hours attribute is returned through this parameter. - `mins`: buffer to store minutes attribute to. Upon function execution minutes attribute is returned through this parameter.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`var year : word;`
	`    month, day, hours, mins : byte;`
	`...`
	`Mmc_Fat_Get_File_Date(year, month, day, hours, mins);`

Mmc_Fat_Get_File_Size

Prototype	`function Mmc_Fat_Get_File_Size(): dword;`
Returns	Size of the currently assigned file in bytes.
Description	This function reads size of the currently assigned file in bytes.
Requires	MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init. The file must be previously assigned. See Mmc_Fat_Assign.

Example	`var my_file_size : dword;`
	`...`
	`my_file_size := Mmc_Fat_Get_File_Size();`

Mmc_Fat_Get_Swap_File

Prototype	`function Mmc_Fat_Get_Swap_File(sectors_cnt: longint; var filename : string[ 11] ; file_attr : byte) : dword;`
Returns	- Number of the start sector for the newly created swap file, if there was enough free space on the MMC/SD card to create file of required size. - `0` - otherwise.
Description	This function is used to create a swap file of predefined name and size on the MMC/SD media. If a file with specified name already exists on the media, search for consecutive sectors will ignore sectors occupied by this file. Therefore, it is recommended to erase such file if it already exists before calling this function. If it is not erased and there is still enough space for a new swap file, this function will delete it after allocating new memory space for a new swap file. The purpose of the swap file is to make reading and writing to MMC/SD media as fast as possible, by using the Mmc_Read_Sector() and Mmc_Write_Sector() functions directly, without potentially damaging the FAT system. The swap file can be considered as a "window" on the media where the user can freely write/read data. Its main purpose in the mikroPascal's 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. Parameters: - `sectors_cnt`: number of consecutive sectors that user wants the swap file to have. - `filename`: name of the file that should be assigned for file operations. File name should be in DOS 8.3 (file_name.extension) format. The file name and extension will be automatically padded with spaces by the library if they have less than length required (i.e. "mikro.tx" -> "mikro .tx "), so the user does not have to take care of that. The file name and extension are case insensitive. The library will convert them to proper case automatically, so the user does not have to take care of that. Also, in order to keep backward compatibility with the first version of this library, file names can be entered as UPPERCASE string of 11 bytes in length with no dot character between file name and extension (i.e. "MIKROELETXT" -> MIKROELE.TXT). In this case the last 3 characters of the string are considered to be file extension. - `file_attr`: file creation and attributs flags. Each bit corresponds to the appropriate file attribut:

Description

Bit	Mask	Description
0	0x01	Read Only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume Label
4	0x10	Subdirectory
5	0x20	Archive
6	0x40	Device (internal use only, never found on disk)
7	0x80	Not used

Note: Long File Names (LFN) are not supported.

Requires

MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init.

Example

```
//-------------- Try to create a swap file with archive atribute,
whose size will be at least 1000 sectors.
//                 If it succeeds, it sends No. of start sector
over USART
var size : dword;
...
size := Mmc_Fat_Get_Swap_File(1000, 'mikroE.txt', 0x20);
if (size <> 0) then
begin
  UART1_Write(0xAA);
  UART1_Write(Lo(size));
  UART1_Write(Hi(size));
  UART1_Write(Higher(size));
  UART1_Write(Highest(size));
  UART1_Write(0xAA);
end;
```

Library Example

The following example demonstrates MMC library test. Upon flashing, insert a MMC/SD card into the module, when you should receive the "Init-OK" message. Then, you can experiment with MMC read and write functions, and observe the results through the Usart Terminal.

```
// if defined, we have a debug messages on PC terminal
program MMC_Test;

{ $DEFINE RS232_debug}

var MMC_chip_select : sbit at PORTB.B2;
var MMC_chip_select_direction : sbit at DDRB.B2;

// universal variables
var  k, i : word; // universal for loops and other stuff

// Variables for MMC routines
 dData : array[ 512] of byte;// Buffer for MMC sector reading/writing
  data_for_registers : array[ 16] of byte; // buffer for CID and CSD
registers

// Display byte in hex
procedure printhex(i : byte) ;
var bHi, bLo : byte;
begin
  bHi := i and 0xF0;                // High nibble
  bHi := bHi shr 4;
  bHi := bHi + '0';
  if (bHi>'9') then
    bHi := bHi + 7;
  bLo := (i and 0x0F) + '0';        // Low nibble
  if (bLo>'9') then
    bLo := bLo+7;
  UART1_Write(bHi);
  UART1_Write(bLo);
end;

begin

  DDRC := 255;
  PORTC := 0;
  { $IFDEF RS232_debug}
    UART1_Init(19200);
  { $ENDIF}

Delay_ms(10);
  DDRA := 255;
  PORTA := 1;
  {$IFDEF RS232_debug}
    UART1_Write_Text('PIC-Started'); // If PIC present report
    UART1_Write(13);
    UART1_Write(10);
  {$ENDIF}

  // Before all, we must initialize a MMC card
   SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV2, _SPI_CLK_LO_LEAD-
ING);
  Spi_Rd_Ptr := @SPI1_Read;

  i := Mmc_Init();
  PORTC := i;
  {$IFDEF RS232_debug}
    if(i = 0) then
      begin
        UART1_Write_Text('MMC Init-OK'); // If MMC present report
        UART1_Write(13);
        UART1_Write(10);
      end;
    if(i) then
      begin
        UART1_Write_Text('MMC Init-error'); // If error report
        UART1_Write(13);
        UART1_Write(10);
      end;
  {$ENDIF}

  for i:=0 to 511 do
    dData[i] := 'E'; // Fill MMC buffer with same characters
  i := Mmc_Write_Sector(55, dData);

  {$IFDEF RS232_debug}
    if(i = 0) then
      UART1_Write_Text('Write-OK')
    else  // if there are errors.....
      UART1_Write_Text('Write-Error');
    UART1_Write(13);
    UART1_Write(10);
  {$ENDIF}

 // Reading of CID and CSD register on MMC card.....
  {$IFDEF RS232_debug}
    i := Mmc_Read_Cid(data_for_registers);
    if(i = 0) then

begin
        for k:=0 to 15 do
begin
```

```
            printhex(data_for_registers[ k] );
            if(k <> 15) then
              UART1_Write('-');
          end;
        UART1_Write(13);
      end
    else
      begin
        UART1_Write_Text('CID-error');
      end;
    i := Mmc_Read_Csd(data_for_registers);
    if(i = 0) then
      begin
        for k:=0 to 15 do
          begin
            printhex(data_for_registers[ k] );
            if(k <> 15) then
              UART1_Write('-');
          end;
        UART1_Write(13);
        UART1_Write(10);
      end
    else
      begin
        UART1_Write_Text('CSD-error');
      end;
  { $ENDIF}
end.
```

Next example consists of several blocks that demonstrate various aspects of usage of the Mmc_Fat16 library, creation of new file and writing down to it, opening existing file and re-writing it, opening existing file and appending data to it, opening a file and reading data it, creating and modifying several files at once, reading file contents, deleting file(s) and creating the swap file.

```
Program MMC_FAT_Test;

var
  Mmc_Chip_Select : sbit at PORTG.B1;
  Mmc_Chip_Select_Direction : sbit at DDRG.B1;

var
  FAT_TXT : string[ 20] ;
  file_contents : string[ 50] ;

  filename : string[ 14] ; // File names

character : byte;
  loop, loop2 : byte;
```

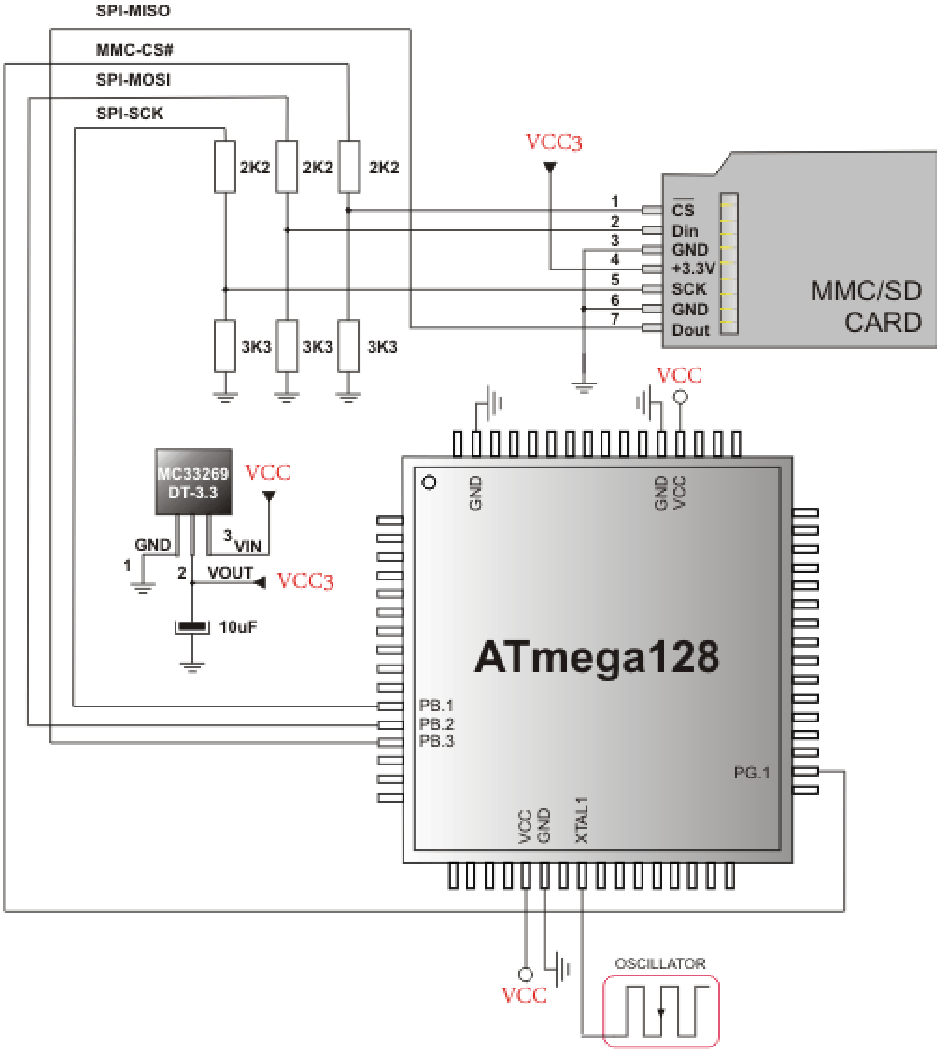
```
size : longint;

  buffer : array[ 512] of byte;

//-------------- Writes string to USART
procedure Write_Str(var ostr: array[ 2] of byte);
var
  i : byte;
begin
  i := 0;
  while ostr[ i] <> 0 do begin
    UART1_Write (ostr[ i] );
    Inc(i);
  end;
  UART1_Write($0A);
end;//~

//-------------- Creates new file and writes some data to it
procedure Create_New_File;
begin
  filename[ 7] := 'A';           // Set filename for single-file tests
  Mmc_Fat_Assign(filename, 0xA0);    // Will not find file and then
create file
  Mmc_Fat_Rewrite;            // To clear file and start with new data
  for loop:=1 to 99 do      //  We want 5 files on the MMC card
    begin
      UART1_Write('.');
      file_contents[ 0] := loop div 10 + 48;
      file_contents[ 1] := loop mod 10 + 48;
        Mmc_Fat_Write(file_contents, 42);     // write data to the
assigned file
    end;
end;//~

//-------------- Creates many new files and writes data to them
procedure Create_Multiple_Files;
begin
  for loop2 := 'B' to 'Z' do
    begin
      UART1_Write(loop2);// this line can slow down the performance
       filename[ 7] := loop2;                // set filename
       Mmc_Fat_Assign(filename, 0xA0);      // find existing file or
create a new one
       Mmc_Fat_Rewrite;     // To clear file and start with new data
        for loop := 1 to 44 do
          begin
 file_contents[ 0] := byte(loop div 10 + 48);
 file_contents[ 1] := byte(loop mod 10 + 48);
end;
  end;
```

```
end;//~

//-------------- Opens an existing file and rewrites it
procedure Open_File_Rewrite;
 begin
  filename[ 7]  := 'C';          // Set filename for single-file tests
  Mmc_Fat_Assign(filename, 0);
  Mmc_Fat_Rewrite;
  for loop := 1 to 55 do
   begin
    file_contents[ 0]  := byte(loop div 10 + 48);
    file_contents[ 1]  := byte(loop mod 10 + 48);
 Mmc_Fat_Write(file_contents, 42); // write data to the assigned file
   end;
 end;//~

//-------------- Opens an existing file and appends data to it
//                 (and alters the date/time stamp)
procedure Open_File_Append;
 begin
   filename[ 7]  := 'B';
   Mmc_Fat_Assign(filename, 0);
   Mmc_Fat_Set_File_Date(2005,6,21,10,35,0);
   Mmc_Fat_Append();                      // Prepare file for append
   file_contents := ' for mikroElektronika 2007';     // Prepare file
for append
   file_contents[ 26]  := 10;          // LF
   Mmc_Fat_Write(file_contents, 27);  // Write data to assigned file
 end;//~

//-------------- Opens an existing file, reads data from it and puts
it to USART
procedure Open_File_Read;
begin
  filename[ 7]  := 'B';
  Mmc_Fat_Assign(filename, 0);
  Mmc_Fat_Reset(size);                     // To read file, procedure
returns size of file
  while size > 0 do
   begin
       Mmc_Fat_Read(character);
 UART1_Write(character);                  // Write data to USART
       Dec(size);
    end;
end;//~

//-------------- Deletes a file. If file doesn't exist, it will first
be created
//                and then deleted.
```

```
procedure Delete_File;
begin
  filename[ 7] := 'F';
  Mmc_Fat_Assign(filename, 0);
  Mmc_Fat_Delete;
end;//~

//-------------- Tests whether file exists, and if so sends its cre-
ation date
//                 and file size via USART
procedure Test_File_Exist;
var
  fsize: longint;
  year: word;
  month, day, hour, minute: byte;
  outstr: array[ 12] of byte;
begin
  filename[ 7] := 'B';
  if Mmc_Fat_Assign(filename, 0) <> 0 then begin
    //--- file has been found - get its date
    Mmc_Fat_Get_File_Date(year,month,day,hour,minute);
    WordToStr(year, outstr);
    Write_Str(outstr);
    ByteToStr(month, outstr);
    Write_Str(outstr);
    WordToStr(day, outstr);
    Write_Str(outstr);
    WordToStr(hour, outstr);
    Write_Str(outstr);
    WordToStr(minute, outstr);
    Write_Str(outstr);
    //--- get file size
    fsize := Mmc_Fat_Get_File_Size;
    LongIntToStr(fsize, outstr);
    Write_Str(outstr);
 end
  else begin
    //--- file was not found - signal it
    UART1_Write(0x55);
    Delay_ms(1000);
    UART1_Write(0x55);
end;
end;//~

//-------------- Tries to create a swap file, whose size will be at
least 100
//                 sectors (see Help for details)
procedure M_Create_Swap_File() ;
  var  i : word;
  begin
    for i:=0 to 511 do
      Buffer[ i]  := i;

    size := Mmc_Fat_Get_Swap_File(5000, 'mikroE.txt', 0x20);   // see
help on this function for details

    if (size <> 0) then
      begin
        LongIntToStr(size, fat_txt);
        UART1_Write_Text(fat_txt);

        for i:=0 to 4999 do
          begin
            Mmc_Write_Sector(size, Buffer);
            size := size + 1;
            UART1_Write('.');
          end;
      end;
  end;

//-------------- Main. Uncomment the function(s) to test the desired
operation(s)
begin
      FAT_TXT := 'FAT16 not found';
      file_contents := 'XX MMC/SD FAT16 library by Anton Rieckert#';
      file_contents[ 41]  := 10;            // newline
      filename := 'MIKRO00xTXT';

      // we will use PORTC to signal test end
      DDRC  := 0xFF;
      PORTC := 0;
      UART1_Init(19200);
//delay_ms(100);               // Set up USART for file reading
      UART1_Write_Text('Start');
      //--- Init the FAT library
 SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV128, _SPI_CLK_LO_LEAD-
ING);
      Spi_Rd_Ptr := @SPI1_Read;
// use fat16 quick format instead of init routine if a formatting is
needed
if Mmc_Fat_Init() = 0 then begin
          PORTC := 0xF0;
          // reinitialize spi at higher speed
                     SPI1_Init_Advanced(_SPI_MASTER,  _SPI_FCY_DIV2,
_SPI_CLK_LO_LEADING);
          //--- signal start-of-test
          //--- test functions
          Create_New_File;
          Create_Multiple_Files;
```

```
               Open_File_Rewrite;
                Open_File_Append;
                Open_File_Read;
                Delete_File;
                Test_File_Exist;
                M_Create_Swap_File();
                UART1_Write('e');
              end
            else
              begin
                UART1_Write_Text(FAT_TXT);
              end;
            //--- signal end-of-test
            PORTC   := $0F;
            UART1_Write_Text('End');
       end.
```

HW Connection



Pin diagram of MMC memory card

ONEWIRE LIBRARY

The OneWire library provides routines for communication via the Dallas OneWire protocol, e.g. with DS18x20 digital thermometer. OneWire is a Master/Slave protocol, and all communication cabling required is a single wire. OneWire enabled devices should have open collector drivers (with single pull-up resistor) on the shared data line.

Slave devices on the OneWire bus can even get their power supply from data line. For detailed schematic see device datasheet.

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 has also 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: Oscillator frequency Fosc needs to be at least 8MHz in order to use the routines with Dallas digital thermometers.

External dependencies of OneWire Library

The following variables must be defined in all projects using OneWire Library :	Description:	Example :
`var OW_Bit_Read : sbit; sfr; external;`	OneWire read line.	`var OW_Bit_Read : sbit at PINB.B2;`
`var OW_Bit_Write : sbit; sfr; external;`	OneWire write line.	`var OW_Bit_Write : sbit at PORTB.B2;`
`var OW_Bit_Direction : sbit; sfr; external;`	Direction of the OneWire pin.	`var OW_Bit_Direction : sbit at DDRB.B2;`

Library Routines

- Ow_Reset
- Ow_Read
- Ow_Write

Ow_Reset

Prototype	`function Ow_Reset(): word;`
Returns	- `0` if the device is present - `1` if the device is not present
Description	Issues OneWire reset signal for DS18x20. Parameters : - None.
Requires	Devices compliant with the Dallas OneWire protocol. Global variables : - `OW_Bit_Read`: OneWire read line - `OW_Bit_Write`: OneWire write line. - `OW_Bit_Direction`: Direction of the OneWire pin must be defined before using this function.

Example	`// OneWire pinout`
	`var OW_Bit_Read : sbit at PINB.B2;`
	`var OW_Bit_Write : sbit at PORTB.B2;`
	`var OW_Bit_Direction : sbit at DDRB.B2;`
	`// end of OneWire pinout`
	
	`// Issue Reset signal on One-Wire Bus`
	`Ow_Reset();`

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.
Requires	Devices compliant with the Dallas OneWire protocol. Global variables : - `OW_Bit_Read`: OneWire read line - `OW_Bit_Write`: OneWire write line. - `OW_Bit_Direction`: Direction of the OneWire pin must be defined before using this function.

Example	`// OneWire pinout`
	`var OW_Bit_Read : sbit at PINB.B2;`
	`var OW_Bit_Write : sbit at PORTB.B2;`
	`var OW_Bit_Direction : sbit at DDRB.B2;`
	`// end of OneWire pinout`
	
	`// Read a byte from the One-Wire Bus`
	`var read_data : byte;`
	`...`
	`read_data := Ow_Read();`

Ow_Write

Prototype	`procedure Ow_Write(par: byte);`
Returns	Nothing.
Description	Writes one byte of data via the OneWire bus. Parameters : - `par`: data to be written
Requires	Devices compliant with the Dallas OneWire protocol. Global variables : - `OW_Bit_Read`: OneWire read line - `OW_Bit_Write`: OneWire write line. - `OW_Bit_Direction`: Direction of the OneWire pin must be defined before using this function.

Example	`// OneWire pinout`
	`var OW_Bit_Read : sbit at PINB.B2;`
	`var OW_Bit_Write : sbit at PORTB.B2;`
	`var OW_Bit_Direction : sbit at DDRB.B2;`
	`// end of OneWire pinout`
	`// Send a byte to the One-Wire Bus`
	`Ow_Write(0xCC);`

Library Example

This example reads the temperature using DS18x20 connected to pin PORTB.2. After reset, MCU obtains temperature from the sensor and prints it on the Lcd. Make sure to pull-up PORTB.2 line and to turn off the PORTB leds.

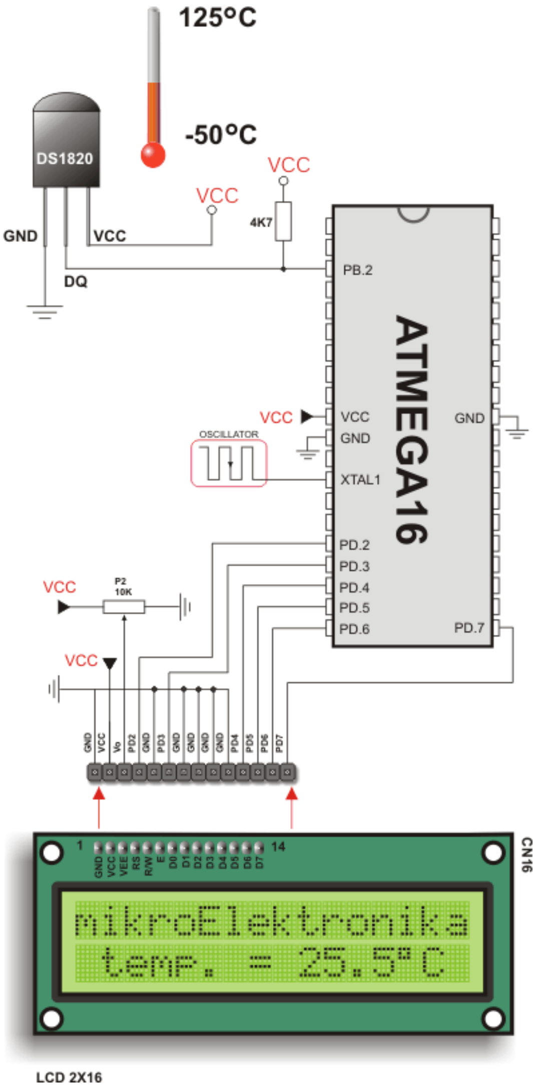
```
program OneWire;

// Lcd module connections
var LCD_RS : sbit at PORTD.B2;
    LCD_EN : sbit at PORTD.B3;
    LCD_D4 : sbit at PORTD.B4;
    LCD_D5 : sbit at PORTD.B5;
    LCD_D6 : sbit at PORTD.B6;
    LCD_D7 : sbit at PORTD.B7;

    LCD_RS_Direction : sbit at DDRD.B2;
    LCD_EN_Direction : sbit at DDRD.B3;
    LCD_D4_Direction : sbit at DDRD.B4;
```

```
LCD_D5_Direction : sbit at DDRD.B5;
    LCD_D6_Direction : sbit at DDRD.B6;
    LCD_D7_Direction : sbit at DDRD.B7;
// End Lcd module connections

// OneWire pinout
var OW_Bit_Write : sbit at PORTB.B2;
    OW_Bit_Read : sbit at PINB.B2;
    OW_Bit_Direction : sbit at DDRB.B2;
// end OneWire definition

//   Set  TEMP_RESOLUTION  to  the  corresponding  resolution  of  used
DS18x20 sensor:
//  18S20: 9  (default setting; can be 9,10,11,or 12)
//  18B20: 12
const TEMP_RESOLUTION : byte = 9;


var text : array[ 9] of byte;
    temp : word;

procedure Display_Temperature( temp2write : word );
const RES_SHIFT = TEMP_RESOLUTION - 8;

var temp_whole : byte;
    temp_fraction : word;

  begin
    text := '000.0000';
    // check if temperature is negative
    if (temp2write and 0x8000) then
      begin
        text[ 0] := '-';
        temp2write := not temp2write + 1;
      end;

    // extract temp_whole
    temp_whole := word(temp2write shr RES_SHIFT);

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

text[ 1] := (temp_whole div 10)mod 10 + 48;      // Extract tens digit
    text[ 2] :=  temp_whole mod 10        + 48;

 // extract temp_fraction and convert it to unsigned int
```

```
    temp_fraction  :=  word(temp2write shl (4-RES_SHIFT));
    temp_fraction  := temp_fraction and 0x000F;
    temp_fraction  := temp_fraction * 625;

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

    // print temperature on Lcd
    Lcd_Out(2, 5, text);
  end;

begin
  text := '000.0000';
  UART1_Init(9600);
  Lcd_Init();                                     // Initialize Lcd
  Lcd_Cmd(LCD_CLEAR);                             // Clear Lcd
  Lcd_Cmd(LCD_CURSOR_OFF);                        // Turn cursor off
  Lcd_Out(1, 1, ' Temperature:   ');
  // Print degree character, 'C' for Centigrades
  Lcd_Chr(2,13,223);   // different Lcd displays have different char
code for degree
                         // if you see greek alpha letter try typing
178 instead of 223
  Lcd_Chr(2,14,'C');

  //--- main loop
  while (TRUE) do
    begin
      //--- perform temperature reading
      Ow_Reset();                        // Onewire reset signal
      Ow_Write(0xCC);                    // Issue command SKIP_ROM
      Ow_Write(0x44);                    // Issue command CONVERT_T
      Delay_us(120);

      Ow_Reset();
      Ow_Write(0xCC);                     // Issue command SKIP_ROM
      Ow_Write(0xBE);                     // Issue command READ_SCRATCHPAD

      temp :=  Ow_Read();
      temp := (Ow_Read() shl 8) + temp;

//--- Format and display result on Lcd
```

```
      Display_Temperature(temp);

      Delay_ms(520);
    end;
end.
```

HW Connection



Example of DS1820 connection

Port Expander Library

The mikroPascal PRO for AVR provides a library for communication with the Microchip's Port Expander MCP23S17 via SPI interface. Connections of the AVR compliant MCU and MCP23S17 is given on the schematic at the bottom of this page.

Note: Library uses the SPI module for communication. The user must initialize SPI module before using the Port Expander Library.

Note: Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

Note: Library does not use Port Expander interrupts.

External dependencies of Port Expander Library

The following variables must be defined in all projects using Port Expander Library:	Description:	Example :
`var SPExpanderRST : sbit; sfr; external;`	Reset line.	`var SPExpanderRST : sbit at PORTB.B0;`
`var SPExpanderCS : sbit; sfr; external;`	Chip Select line.	`var SPExpanderCS : sbit at PORTB.B1;`
`var SPExpanderCS_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var SPExpanderRST_Direction : sbit at DDRB.B0;`
`var SPExpanderCS_Direction : sbit; sfr; external;`	Direction of the Chip Select pin.	`var SPExpanderCS_Direction : sbit at DDRB.B1;`

Library Routines

- Expander_Init
- Expander_Read_Byte
- Expander_Write_Byte
- Expander_Read_PortA
- Expander_Read_PortB
- Expander_Read_PortAB
- Expander_Write_PortA
- Expander_Write_PortB
- Expander_Write_PortAB

- Expander_Set_DirectionPortA
- Expander_Set_DirectionPortB
- Expander_Set_DirectionPortAB
- Expander_Set_PullUpsPortA
- Expander_Set_PullUpsPortB
- Expander_Set_PullUpsPortAB

Expander_Init

Prototype	`procedure Expander_Init(ModuleAddress : byte);`
Returns	Nothing.
Description	Initializes Port Expander using SPI communication. Port Expander module settings : - hardware addressing enabled - automatic address pointer incrementing disabled (byte mode) - BANK_0 register adressing - slew rate enabled Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Requires	Global variables : - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.

Example	`// Port Expander module connections`
	`var SPExpanderCS   : sbit at PORTB.B1;`
	`    SPExpanderRST : sbit at PORTB.B0;`
	`    SPExpanderCS_Direction   : sbit at DDRB.B1;`
	`    SPExpanderRST_Direction : sbit at DDRB.B0;`
	`// End of Port Expander module connections`
	`...`
	`SPI1_Init();                // initialize SPI module`
	`Spi_Rd_Ptr := @SPI1_Read;  // Pass pointer to SPI Read function of used SPI module`
	`Expander_Init(0);          // initialize port expander`

Expander_Read_Byte

Prototype	`function Expander_Read_Byte(ModuleAddress : byte; RegAddress : byte) : byte;`
Returns	Byte read.
Description	The function reads byte from Port Expander. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `RegAddress`: Port Expander's internal register address
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Read a byte from Port Expander's register`
	`var read_data : byte;`
	`...`
	`read_data := Expander_Read_Byte(0,1);`

Expander_Write_Byte

Prototype	`procedure Expander_Write_Byte(ModuleAddress: byte; RegAddress: byte; Data_: byte);`
Returns	Nothing.
Description	Routine writes a byte to Port Expander. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bot tom of this page - `RegAddress`: Port Expander's internal register address - `Data_`: data to be written
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Write a byte to the Port Expander's register`
	`Expander_Write_Byte(0,1,0xFF);`

Expander_Read_PortA

Prototype	`function Expander_Read_PortA(ModuleAddress: byte): byte;`
Returns	Byte read.
Description	The function reads byte from Port Expander's PortA. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bot tom of this page
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA should be configured as input. See Expander_Set_DirectionPortA and Expander_Set_DirectionPortAB routines.

Example	`// Read a byte from Port Expander's PORTA`
	`var read_data : byte;`
	`...`
	`Expander_Set_DirectionPortA(0,0xFF);         // set expander's porta to be input`
	`...`
	`read_data := Expander_Read_PortA(0);`

Expander_Read_PortB

Prototype	`function Expander_Read_PortB(ModuleAddress: byte): byte;`
Returns	Byte read.
Description	The function reads byte from Port Expander's PortB. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortB should be configured as input. See Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`// Read a byte from Port Expander's PORTB`
	`var read_data : byte;`
	`...`
	`Expander_Set_DirectionPortB(0,0xFF);         // set expander's portb to be input`
	`...`
	`read_data := Expander_Read_PortB(0);`

Expander_Read_PortAB

Prototype	**function** Expander_Read_PortAB(ModuleAddress: byte): word;
Returns	Word read.
Description	The function reads word from Port Expander's ports. PortA readings are in the higher byte of the result. PortB readings are in the lower byte of the result. Parameters : - ModuleAddress: Port Expander hardware address, see schematic at the bottom of this page
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA and PortB should be configured as inputs. See Expander_Set_DirectionPortA, Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	// Read a byte from Port Expander's PORTA and PORTB
	var read_data : word;
	...
	Expander_Set_DirectionPortAB(0,0xFFFF); // set expander's porta and portb to be input
	...
	read_data := Expander_Read_PortAB(0);

Expander_Write_PortA

Prototype	**procedure** Expander_Write_PortA(ModuleAddress: byte; Data_: byte);
Returns	Nothing.
Description	The function writes byte to Port Expander's PortA. Parameters : - ModuleAddress: Port Expander hardware address, see schematic at the bottom of this page - Data_: data to be written
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA should be configured as output. See Expander_Set_DirectionPortA and Expander_Set_DirectionPortAB routines.

Example	// Write a byte to Port Expander's PORTA
	
	...
	Expander_Set_DirectionPortA(0,0x00); // set expander's porta to be output
	...
	Expander_Write_PortA(0, 0xAA);

Expander_Write_PortB

Prototype	`procedure Expander_Write_PortB(ModuleAddress: byte; Data_: byte);`
Returns	Nothing.
Description	The function writes byte to Port Expander's PortB. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Data_`: data to be written
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortB should be configured as output. See Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	`// Write a byte to Port Expander's PORTB`
	`...`
	`Expander_Set_DirectionPortB(0,0x00);         // set expander's portb to be output`
	`...`
	`Expander_Write_PortB(0, 0x55);`

Expander_Write_PortAB

Prototype	**procedure** Expander_Write_PortAB(ModuleAddress: byte; Data_: word);
Returns	Nothing.
Description	The function writes word to Port Expander's ports. Parameters : - ModuleAddress: Port Expander hardware address, see schematic at the bottom of this page - Data_: data to be written. Data to be written to PortA are passed in Data's higher byte. Data to be written to PortB are passed in Data's lower byte
Requires	Port Expander must be initialized. See Expander_Init. Port Expander's PortA and PortB should be configured as outputs. See Expander_Set_DirectionPortA, Expander_Set_DirectionPortB and Expander_Set_DirectionPortAB routines.

Example	// Write a byte to Port Expander's PORTA and PORTB
	
	...
	Expander_Set_DirectionPortAB(0,0x0000); // set expander's porta and portb to be output
	...
	Expander_Write_PortAB(0, 0xAA55);

Expander_Set_DirectionPortA

Prototype	**procedure** Expander_Set_DirectionPortA(ModuleAddress: byte; Data_: byte);
Returns	Nothing.
Description	The function sets Port Expander's PortA direction. Parameters : - ModuleAddress: Port Expander hardware address, see schematic at the bottom of this page - Data_: data to be written to the PortA direction register. Each bit corresponds to the appropriate pin of the PortA register. Set bit designates corresponding pin as input. Cleared bit designates corresponding pin as output.
Requires	Port Expander must be initialized. See Expander_Init.

Example	// Set Port Expander's PORTA to be output
	Expander_Set_DirectionPortA(0,0x00);

Expander_Set_DirectionPortB

Prototype	`procedure Expander_Set_DirectionPortB(ModuleAddress: byte; Data_: byte);`
Returns	Nothing.
Description	The function sets Port Expander's PortB direction. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bot tom of this page - `Data_`: data to be written to the PortB direction register. Each bit corresponds to the appropriate pin of the PortB register. Set bit designates corresponding pin as input. Cleared bit designates corresponding pin as output.
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Set Port Expander's PORTB to be input`
	`Expander_Set_DirectionPortB(0,0xFF);`

Expander_Set_DirectionPortAB

Prototype	`procedure Expander_Set_DirectionPortAB(ModuleAddress: byte; Direction: word);`
Returns	Nothing.
Description	The function sets Port Expander's PortA and PortB direction. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `Direction`: data to be written to direction registers. Data to be written to the PortA direction register are passed in `Direction's` higher byte. Data to be written to the PortB direction register are passed in `Direction's` lower byte. Eachbit corresponds to the appropriate pin of the PortA/PortB register. Set bit designates corresponding pin as input. Cleared bit designates corresponding pin as output.
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Set Port Expander's PORTA to be output and PORTB to be input`
	`Expander_Set_DirectionPortAB(0,0x00FF);`

Expander_Set_PullUpsPortA

Prototype	`procedure Expander_Set_PullUpsPortA(ModuleAddress: byte; Data_: byte);`
Returns	Nothing.
Description	The function sets Port Expander's PortA pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bot tom of this page - `Data_`: data for choosing pull up/down resistors configuration. Each bit corre sponds to the appropriate pin of the PortA register. Set bit enables pull-up for corresponding pin.
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Set Port Expander's PORTA pull-up resistors`
	`Expander_Set_PullUpsPortA(0, 0xFF);`

Expander_Set_PullUpsPortB

Prototype	`procedure Expander_Set_PullUpsPortB(ModuleAddress: byte; Data_: byte);`
Returns	Nothing.
Description	The function sets Port Expander's PortB pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bot tom of this page - `Data_`: data for choosing pull up/down resistors configuration. Each bit corre sponds to the appropriate pin of the PortB register. Set bit enables pull-up for corresponding pin.
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Set Port Expander's PORTB pull-up resistors`
	`Expander_Set_PullUpsPortB(0, 0xFF);`

Expander_Set_PullUpsPortAB

Prototype	`procedure Expander_Set_PullUpsPortAB(ModuleAddress: byte; PullUps: word);`
Returns	Nothing.
Description	The function sets Port Expander's PortA and PortB pull up/down resistors. Parameters : - `ModuleAddress`: Port Expander hardware address, see schematic at the bottom of this page - `PullUps`: data for choosing pull up/down resistors configuration. PortA pull up/down resistors configuration is passed in PullUps's higher byte. PortB pull up/down resistors configuration is passed in PullUps's lower byte. Each bit corresponds to the appropriate pin of the PortA/PortB register. Set bit enables pull-up for corresponding pin.
Requires	Port Expander must be initialized. See Expander_Init.

Example	`// Set Port Expander's PORTA and PORTB pull-up resistors`
	`Expander_Set_PullUpsPortAB(0, 0xFFFF);`

Library Example

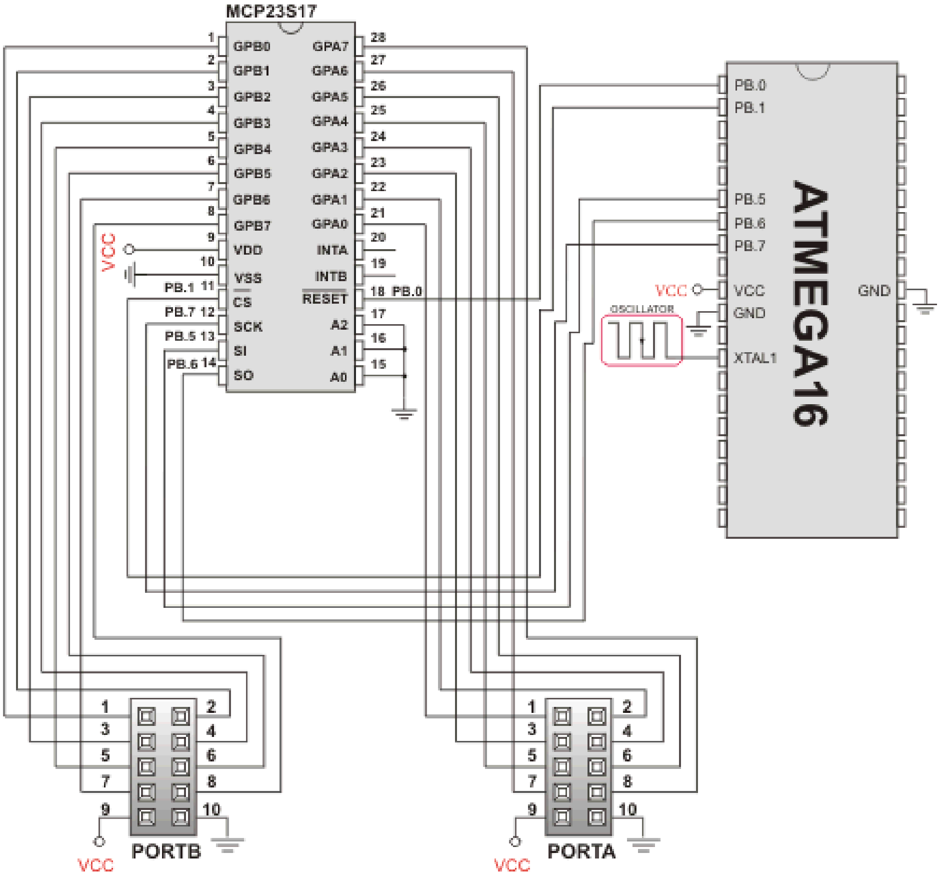
The example demonstrates how to communicate with Port Expander MCP23S17.

Note that Port Expander pins A2 A1 A0 are connected to GND so Port Expander Hardware Address is 0.

```
program PortExpander;

// Port Expander module connections
var SPExpanderRST : sbit at PORTB.B0;
    SPExpanderCS   : sbit at PORTB.B1;
    SPExpanderRST_Direction : sbit at DDRB.B0;
    SPExpanderCS_Direction  : sbit at DDRB.B1;
// End Port Expander module connections

var counter : byte;// = 0;

begin
counter := 0;
  DDRC := 0xFF;                            // Set PORTC as output

  // If Port Expander Library uses SPI1 module
  SPI1_Init();                    // Initialize SPI module used with PortExpander
Spi_Rd_Ptr := @SPI1_Read// Pass pointer to SPI Read function of used SPI module
```

```
//   // If Port Expander Library uses SPI2 module
//    SPI2_Init();              //  Initialize  SPI  module  used  with
PortExpander
//   Spi_Rd_Ptr := @SPI2_Read;   // Pass pointer to SPI Read function
of used SPI module

  Expander_Init(0);            // Initialize Port Expander

  Expander_Set_DirectionPortA(0, 0x00);  // Set Expander's PORTA to
be output

  Expander_Set_DirectionPortB(0,0xFF);   // Set Expander's PORTB to
be input
  Expander_Set_PullUpsPortB(0,0xFF);      // Set pull-ups to all of
the Expander's PORTB pins

  while ( TRUE ) do                        // Endless loop
    begin
       Expander_Write_PortA(0, counter);   // Write i to expander's
PORTA
      Inc(counter);
       PORTC := Expander_Read_PortB(0);    // Read expander's PORTB
and write it to LEDs
      Delay_ms(100);
    end;

end.
```

HW Connection



Port Expander HW connection

PS/2 LIBRARY

The mikroPascal PRO for AVR provides a library for communication with the common PS/2 keyboard.

Note: The library does not utilize interrupts for data retrieval, and requires the oscillator clock to be at least 6MHz.

Note: The pins to which a PS/2 keyboard is attached should be connected to the pull-up resistors.

Note: Although PS/2 is a two-way communication bus, this library does not provide MCU-to-keyboard communication; e.g. pressing the Caps Lock key will not turn on the Caps Lock LED.

External dependencies of PS/2 Library

The following variables must be defined in all projects using PS/2 Library:	Description:	Example :
`var PS2_Data : sbit; sfr; external;`	PS/2 Data line.	`var PS2_Data : sbit at PINC.B0;`
`var PS2_In_Clock : sbit; sfr; external;`	PS/2 Clock line in.	`var PS2_In_Clock : sbit at PINC.B1;`
`var PS2_Out_Clock : sbit; sfr; external;`	PS/2 Clock line out.	`var PS2_Out_Clock : sbit at PORTC.B1;`
`var PS2_Data_Direction : sbit; sfr; external;`	Direction of the PS/2 Data pin.	`var PS2_Data_Direction : sbit at DDRC.B0;`
`var PS2_Clock_Direction : sbit; sfr; external;`	Direction of the PS/2 Clock pin.	`var PS2_Clock_Direction : sbit at DDRC.B1;`

Library Routines

- Ps2_Config
- Ps2_Key_Read

Ps2_Config

Prototype	`procedure Ps2_Config();`
Returns	Nothing.
Description	Initializes the MCU for work with the PS/2 keyboard.
Requires	Global variables : - `PS2_Data`: Data signal line - `PS2_In_Clock`: Clock signal line in - `PS2_Out_Clock`: Clock signal line out - `PS2_Data_Direction`: Direction of the Data pin - `PS2_Clock_Direction`: Direction of the Clock pin must be defined before using this function.

Example	`// PS2 pinout definition`
	`var PS2_Data : sbit at PINC.B0;`
	`var PS2_In_Clock : sbit at PINC.B1;`
	`var PS2_Out_Clock : sbit at PORTC.B1;`
	`var PS2_Data_Direction : sbit at DDRC.B0;`
	`var PS2_Clock_Direction : sbit at DDRC.B1;`
	`// End of PS2 pinout definition`
	
	`...`
	`Ps2_Config();          // Init PS/2 Keyboard`

Ps2_Key_Read

Prototype	`function Ps2_Key_Read(var value: byte; var special: byte; var pressed: byte): byte;`
Returns	- `1` if reading of a key from the keyboard was successful - `0` if no key was pressed
Description	The function retrieves information on key pressed. Parameters : - `value`: holds the value of the key pressed. For characters, numerals, punctuation marks, and space value will store the appropriate ASCII code. Routine “recognizes” the function of Shift and Caps Lock, and behaves appropriately. For special function keys see Special Function Keys Table. special: is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of these, special will be set to 1, otherwise 0. - `pressed`: is set to 1 if the key is pressed, and 0 if it is released.
Requires	PS/2 keyboard needs to be initialized. See Ps2_Config routine.

Example	`var value, special, pressed: byte;`
	`...`
	`// Press Enter to continue:`
	`repeat`
	`  if (Ps2_Key_Read(value, special, pressed)) then`
	`    if ((value = 13) and (special = 1)) then break;`
	`until (0=1);`

Special Function Keys

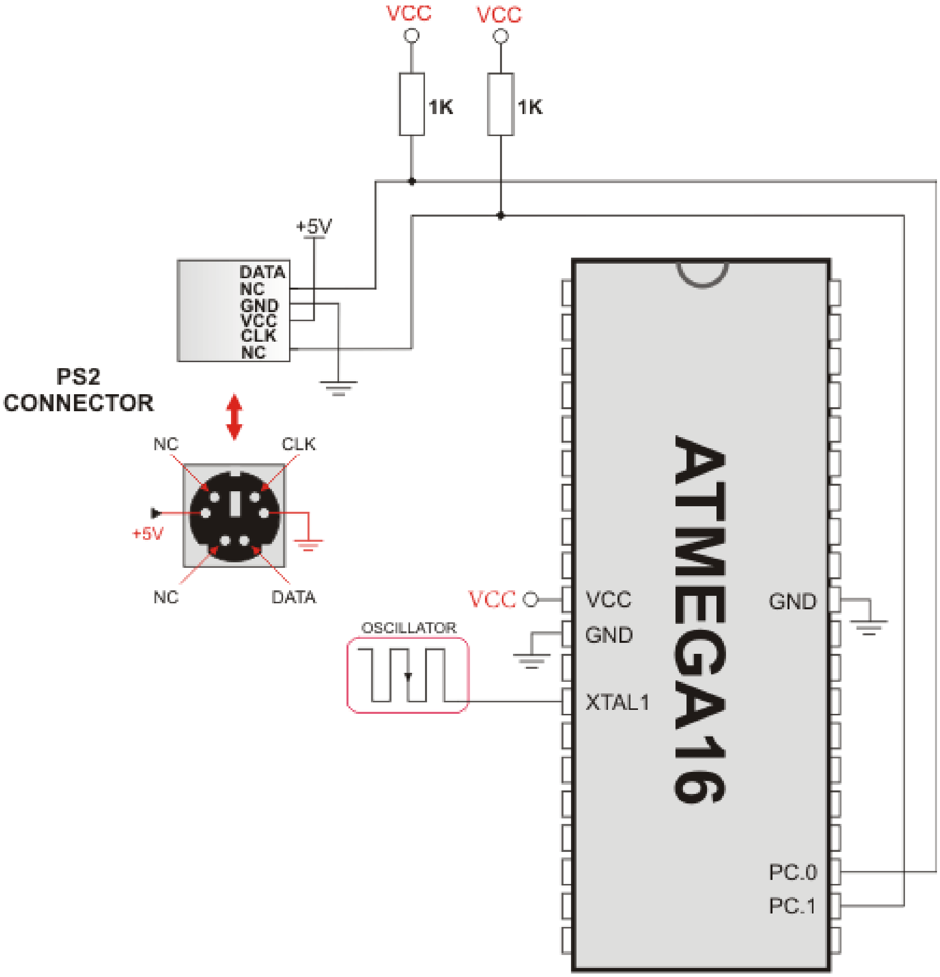
Key	Value returned
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9
F10	10
F11	11
F12	12
Enter	13
Page Up	14
Page Down	15
Backspace	16
Insert	17
Delete	18
Windows	19
Ctrl	20
Shift	21
Alt	22
Print Screen	23
Pause	24
Caps Lock	25
End	26
Home	27
Scroll Lock	28
Num Lock	29
Left Arrow	30
Right Arrow	31
Up Arrow	32
Down Arrow	33
Escape	34
Tab	35

Library Example

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

```
program PS2_Example;

var keydata, special, down : byte;

var PS2_Data              : sbit at PINC.B0;
    PS2_Clock_Input       : sbit at PINC.B1;
    PS2_Clock_Output      : sbit at PORTC.B1;

    PS2_Data_Direction    : sbit at DDRC.B0;
    PS2_Clock_Direction   : sbit at DDRC.B1;

begin
  UART1_Init(19200);        // Initialize UART module at 19200 bps
  Ps2_Config();             // Init PS/2 Keyboard
  Delay_ms(100);            // Wait for keyboard to finish
  UART1_Write('R');    `    // Ready
    while TRUE do           // Endless loop
      begin

        if Ps2_Key_Read(keydata, special, down) then      // If data
was read from PS/2
           begin

            if (down <> 0) and (keydata = 16) then // Backspace read
               begin
                 UART1_Write(0x08);                             // Send
Backspace to USART terminal
               end
        else if (down <> 0) and (keydata = 13) then   // Enter read
               begin
                 UART1_Write(10);                               // Send
carriage return to usart terminal
                   UART1_Write(13);                                 //
Uncomment this line if usart terminal also expects line feed
                                       //    for new line transition
               end
             else if (down <> 0) and (special = 0) and (keydata <>
0) then    // Common key read
               begin
                 UART1_Write(keydata); // Send key to usart terminal
               end;
      end;
       Delay_ms(10);                                // Debounce period
      end;
end.
```

HW Connection



Example of PS2 keyboard connection

PWM LIBRARY

CMO module is available with a number of AVR MCUs. mikroPascal PRO for AVR provides library which simplifies using PWM HW Module.

Note: For better understanding of PWM module it would be best to start with the example provided in Examples folder of our mikroPascal PRO for AVR compiler. When you select a MCU, mikroPascal PRO for AVR automatically loads the correct PWM library (or libraries), which can be verified by looking at the Library Manager. PWM library handles and initializes the PWM module on the given AVR MCU, but it is up to user to set the correct pins as PWM output. This topic will be covered later in this section. mikroPascal PRO for AVR does not support enhanced PWM modules.

Library Routines

- PWM_Init
- PWM_Set_Duty
- PWM_Start
- PWM_Stop
- PWM1_Init
- PWM1_Set_Duty
- PWM1_Start
- PWM1_Stop

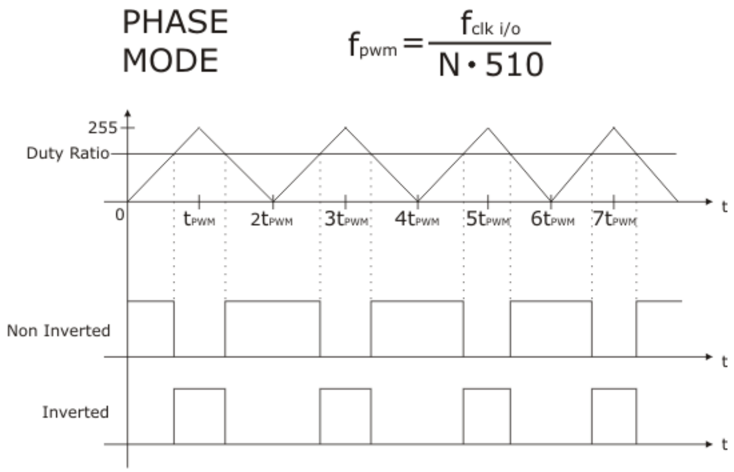
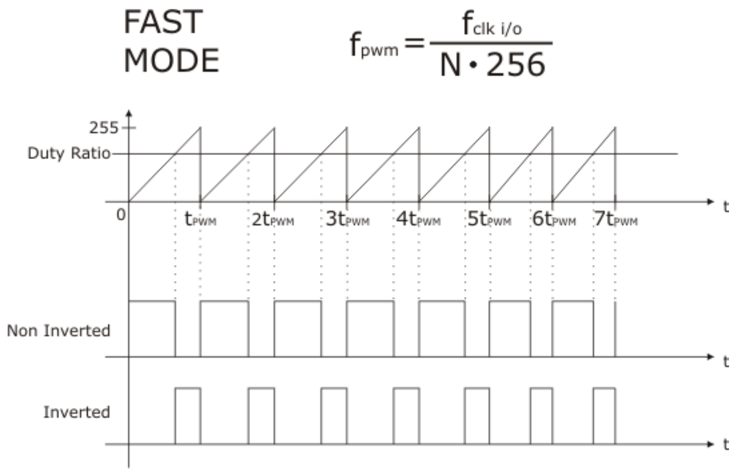
Predefined constants used in PWM library

The following variables are used in PWM library functions:	Description:
_PWM_PHASE_CORRECT_MODE	Selects Phase Correct PWM mode on first PWM library.
_PWM1_PHASE_CORRECT_MODE	Selects Phase Correct PWM mode on second PWM library (if it exists in Library Manager.
_PWM_FAST_MODE	Selects Fast PWM mode on first PWM library.
_PWM1_FAST_MODE	Selects Fast PWM mode on second PWM library (if it exists in Library Manager.
_PWM_PRESCALER_1	Sets prescaler value to 1 (No prescaling).
_PWM_PRESCALER_8	Sets prescaler value to 8.
_PWM_PRESCALER_32	Sets prescaler value to 32 (this value is not available on every MCU. Please use Code Assistant to see if this value is available for the given MCU.

`_PWM_PRESCALER_64`	Sets prescaler value to 64.
`_PWM_PRESCALER_128`	Sets prescaler value to 128 (this value is not available on every MCU. Please use Code Assistant to see if this value is available for the given MCU.
`_PWM_PRESCALER_256`	Sets prescaler value to 256.
`_PWM_PRESCALER_1024`	Sets prescaler value to 1024.
`_PWM1_PRESCALER_1`	Sets prescaler value to 1 on second PWM library (if it exists in Library Manager).
`_PWM1_PRESCALER_8`	Sets prescaler value to 8 on second PWM library (if it exists in Library Manager).
`_PWM1_PRESCALER_32`	Sets prescaler value to 32 on second PWM library (if it exists in Library Manager). This value is not available on every MCU. Please use Code Assistant to see if this value is available for the given MCU.
`_PWM1_PRESCALER_64`	Sets prescaler value to 64 on second PWM library (if it exists in Library Manager).
`_PWM1_PRESCALER_128`	Sets prescaler value to 128 on second PWM library (if it exists in Library Manager). This value is not available on every MCU. Please use Code Assistant to see if this value is available for the given MCU.
`_PWM1_PRESCALER_256`	Sets prescaler value to 256 on second PWM library (if it exists in Library Manager).
`_PWM1_PRESCALER_1024`	Sets prescaler value to 1024 on second PWM library (if it exists in Library Manager).
`_PWM_INVERTED`	Selects the inverted PWM mode.
`_PWM1_INVERTED`	Selects the inverted PWM mode on second PWM library (if it exists in Library Manager).
`_PWM_NON_INVERTED`	Selects the normal (non inverted) PWM mode.
`_PWM1_NON_INVERTED`	Selects the normal (non inverted) PWM mode on second PWM library (if it exists in Library Manager).

Note: Not all of the MCUs have both PWM and PWM1 library included. Sometimes, like its the case with ATmega8515, MCU has only PWM library. Therefore constants that have in their name PWM1 are invalid (for ATmega8515) and will not be visible from Code Assistant. It is highly advisable to use this feature, since it handles all the constants (available) and eliminates any chance of typing error.

PWM_Init

Prototype	`procedure PWM_Init(wave_mode : byte; prescaler : byte; inverted : byte; duty : byte);`
Returns	Nothing.
Description	Initializes the PWM module. Parameter wave_mode is a desired PWM mode. There are two modes: Phase Correct and Fast PWM. Parameter prescaler chooses prescale value N = 1,8,64,256 or 1024 (some modules support 32 and 128, but for this you will need to check the datasheet for the desired MCU). Paremeter inverted is for choosing between inverted and non inverted PWM signal. Parameter duty sets duty ratio from 0 to 255. PWM signal graphs and formulas are shown below.  PHASE MODE $f_{pwm} = \frac{f_{clk\ i/o}}{N \cdot 510}$ 255 Duty Ratio 0 t_{PWM} $2t_{PWM}$ $3t_{PWM}$ $4t_{PWM}$ $5t_{PWM}$ $6t_{PWM}$ $7t_{PWM}$ t Non Inverted t Inverted t  FAST MODE $f_{pwm} = \frac{f_{clk\ i/o}}{N \cdot 256}$ 255 Duty Ratio 0 t_{PWM} $2t_{PWM}$ $3t_{PWM}$ $4t_{PWM}$ $5t_{PWM}$ $6t_{PWM}$ $7t_{PWM}$ t Non Inverted t Inverted t

Description	The N variable represents the prescaler factor (1, 8, 64, 256, or 1024). Some modules also support 32 and 128 prescaler value, but for this you will need to check the datasheet for the desired MCU) PWM_Init must be called before using other functions from PWM Library.

Requires	You need a CMO on the given MCU (that supports PWM).
	Before calling this routine you must set the output pin for the PWM (according to the datasheet):
	`DDRB.3 = 1; // set PORTB pin 3 as output for the PWM`
	This code example is for ATmega16, for different MCU please consult datasheet for the correct pinout of the PWM module or modules.

Example	Initialize PWM module: `PWM_Init(_PWM_FAST_MODE, _PWM_PRESCALER_8, _PWM_NON_INVERTED, 127);`

PWM_Set_Duty

Prototype	`procedure PWM_Set_Duty(duty : byte);`
Returns	Nothing.
Description	Changes PWM duty ratio. Parameter duty takes values from 0 to 255, where 0 is 0%, 127 is 50%, and 255 is 100% duty ratio. Other specific values for duty ratio can be calculated as `(Percent*255)/100`.
Requires	PWM module must to be initialised (PWM_Init) before using PWM_Set_Duty function.
Example	For example lets set duty ratio to 75%: `PWM_Set_Duty(192);`

PWM_Start

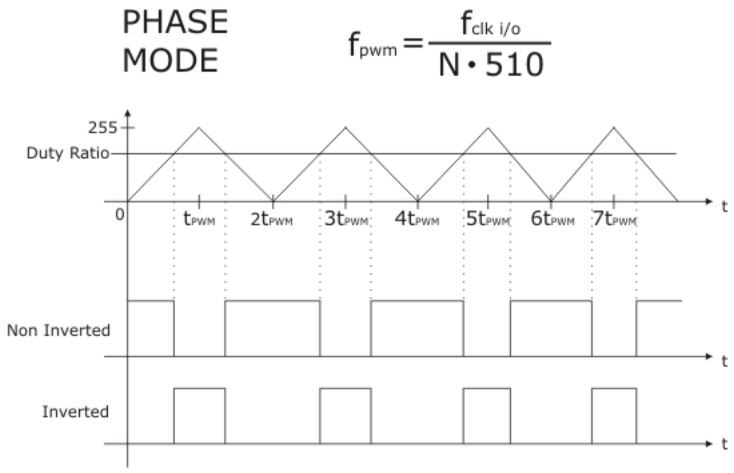
Prototype	`procedure PWM_Start();`
Returns	Nothing.
Description	Starts PWM
Requires	MCU must have CMO module to use this library. PWM_Init must be called before using this routine.
Example	`PWM_Start();`

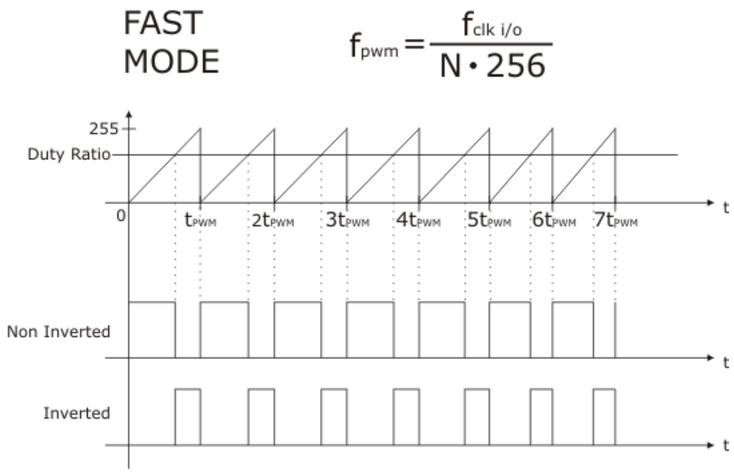
PWM_Stop

Prototype	`procedure PWM_Stop();`
Returns	Nothing.
Description	Stops the PWM.
Requires	MCU must have CMO module to use this library. PWM_Init and PWM_Start must be called before using this routine using this routine, otherwise it will have no effect as the PWM module is not running.
Example	`PWM_Stop();`

Note: Not all the AVR MCUs support both PWM and PWM1 library. The best way to verify this is by checking the datasheet for the desired MCU. Also you can check this by selecting a MCU in mikroPascal PRO for AVR looking at the Library Manager. If library manager loads both PWM and PWM1 library (you are able to check them) then this MCU supports both PWM libraries. Here you can take full advantage of our Code Assistant and Parameter Assistant feature of our compiler.

PWM1_Init

Prototype	`procedure PWM1_Init(wave_mode : byte; prescaler : byte; inverted : byte; duty : byte);`
Returns	Nothing.
Description	Initializes the PWM module. Parameter wave_mode is a desired PWM mode. There are two modes: Phase Correct and Fast PWM. Parameter prescaler chooses prescale value N = 1,8,64,256 or 1024 (some modules support 32 and 128, but for this you will need to check the datasheet for the desired MCU). Paremeter inverted is for choosing between inverted and non inverted PWM signal. Parameter duty sets duty ratio from 0 to 255. PWM signal graphs and formulas are shown below.  PHASE MODE $f_{pwm} = \frac{f_{clk\ i/o}}{N \cdot 510}$ 255 Duty Ratio 0 t_{PWM} $2t_{PWM}$ $3t_{PWM}$ $4t_{PWM}$ $5t_{PWM}$ $6t_{PWM}$ $7t_{PWM}$ t Non Inverted t Inverted t

Description	 FAST MODE $f_{pwm} = \frac{f_{clk\ i/o}}{N \cdot 256}$ 255 Duty Ratio 0 t_{PWM} $2t_{PWM}$ $3t_{PWM}$ $4t_{PWM}$ $5t_{PWM}$ $6t_{PWM}$ $7t_{PWM}$ t Non Inverted t Inverted t The N variable represents the `prescaler` factor (1, 8, 64, 256, or 1024). Some modules also support 32 and 128 `prescaler` value, but for this you will need to check the datasheet for the desired MCU) PWM1_Init must be called before using other functions from PWM Library.

Requires	You need a CMO on the given MCU (that supports PWM).
	Before calling this routine you must set the output pin for the PWM (according to the datasheet):
	`DDRB.7 = 1; // set PORTB pin 7 as output for the PWM1`
	This code cxample is for ATmega16 (second PWM module), for different MCU please consult datasheet for the correct pinout of the PWM module or modules.

Example	Initialize PWM module: `PWM1_Init(_PWM1_FAST_MODE,_PWM1_PRESCALER_8, _PWM1_NON_INVERTED,127);`

PWM1_Set_Duty

Prototype	`procedure PWM1_Set_Duty(duty : byte);`
Returns	Nothing.
Description	Changes PWM duty ratio. Parameter duty takes values from 0 to 255, where 0 is 0%, 127 is 50%, and 255 is 100% duty ratio. Other specific values for duty ratio can be calculated as `(Percent*255)/100`.
Requires	PWM module must to be initialised (PWM1_Init) before using PWM_Set_Duty function.
Example	For example lets set duty ratio to 75%: `PWM1_Set_Duty(192);`

PWM1_Start

Prototype	`procedure PWM1_Start();`
Returns	Nothing.
Description	Starts PWM.
Requires	MCU must have CMO module to use this library. PWM1_Init must be called before using this routine.
Example	`PWM1_Start();`

PWM1_Stop

Prototype	`procedure PWM1_Stop();`
Returns	Nothing.
Description	Stops the PWM.
Requires	MCU must have CMO module to use this library. PWM1_Init and PWM1_Start must be called before using this routine using this routine, otherwise it will have no effect as the PWM module is not running.
Example	`PWM1_Stop();`

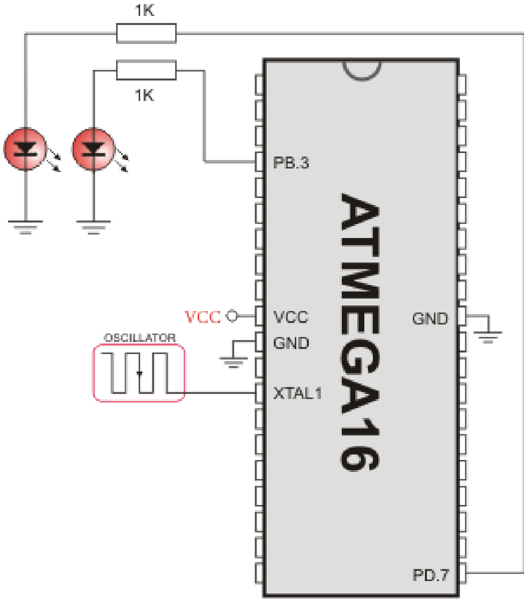
Library Example

The example changes PWM duty ratio on PB3 and PB7 pins continually. If LED is connected to PB3 and PB7, you can observe the gradual change of emitted light.

```
program PWM_Test;

var current_duty : byte;
    current_duty1 : byte;

begin
  DDRB.B0 := 0;                    // Set PORTB pin 0 as input
  DDRB.B1 := 0;                    // Set PORTB pin 1 as input

  DDRC.B0 := 0;                    // Set PORTC pin 0 as input
  DDRC.B1 := 0;                    // Set PORTC pin 1 as input

  current_duty     := 127;         // initial value for current_duty
  current_duty1    := 127;         // initial value for current_duty

  DDRB.B3 := 1;                    // Set PORTB pin 3 as output pin
for the PWM (according to datasheet)
  DDRD.B7 := 1;                    // Set PORTD pin 7 as output pin
for the PWM1 (according to datasheet)

           PWM_Init(_PWM_PHASE_CORRECT_MODE,      _PWM_PRESCALER_8,
_PWM_NON_INVERTED, 127);

         PWM1_Init(_PWM1_PHASE_CORRECT_MODE,     _PWM1_PRESCALER_8,
_PWM1_NON_INVERTED, 127);

  while TRUE do
 begin
    if (PINB.0 <> 0) then
        begin               // Detect if PORTB pin 0 is pressed
          Delay_ms(40);     // Small delay to avoid deboucing effect
          Inc(current_duty);                // Increment duty ratio
          PWM_Set_Duty(current_duty);       // Set incremented duty
         end
      else
if (PINB.1 <> 0) then            // Detect if PORTB pin 1 is pressed
           begin
           Delay_ms(40);    // Small delay to avoid deboucing effect
            Dec(current_duty);              // Decrement duty ratio
             PWM_Set_Duty(current_duty);         // Set decremented
duty ratio
           end
```

```
else
         if (PINC.0 <> 0) then   // Detect if PORTC pin 0 is pressed
            begin
             Delay_ms(40); // Small delay to avoid debroucing effect
             Inc(current_duty1);           // Increment duty ratio
             PWM1_Set_Duty(current_duty1);  // Set incremented duty
            end
           else
           if (PINC.1 <> 0) then // Detect if PORTC pin 1 is pressed
              begin
                Delay_ms(40);                       // Small delay to
avoid debroucing effect
               Dec(current_duty1);          // Decrement duty ratio
                PWM1_Set_Duty(current_duty1);    // Set decremented
duty ratio
              end;
    end;

end.
```

HW Connection



PWM demonstration

PWM 16 BIT LIBRARY

CMO module is available with a number of AVR MCUs. mikroPascal PRO for AVR provides library which simplifies using PWM HW Module.

Note: For better understanding of PWM module it would be best to start with the example provided in Examples folder of our mikroPascal PRO for AVR compiler. When you select a MCU, mikroPascal PRO for AVR automaticaly loads the correct PWM-16bit library, which can be verified by looking at the Library Manager. PWM library handles and initializes the PWM module on the given AVR MCU, but it is up to user to set the correct pins as PWM output, this topic will be covered later in this section.

Library Routines

- PWM16bit_Init
- PWM16bit_Change_Duty
- PWM16bit_Start
- PWM16bit_Stop

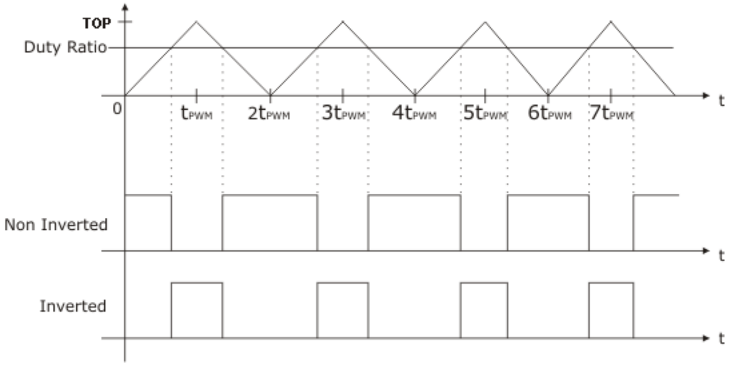
Predefined constants used in PWM-16bit library

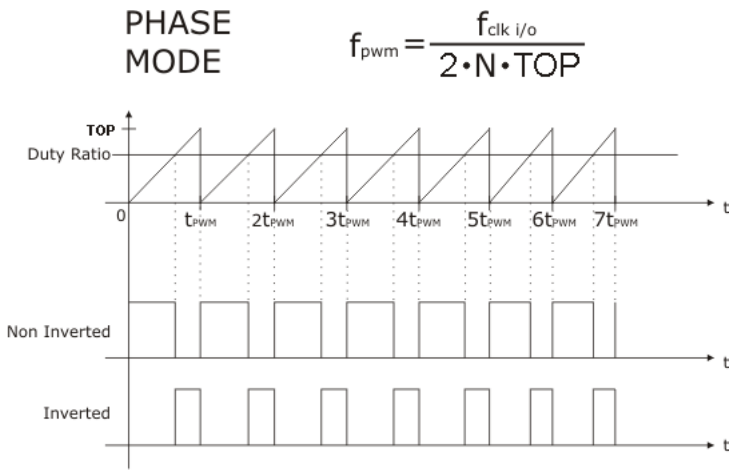
The following variables are used in PWM library functions:	Description:
_PWM16_PHASE_CORRECT_MODE_8BIT	Selects Phase Correct, 8-bit mode.
_PWM16_PHASE_CORRECT_MODE_9BIT	Selects Phase Correct, 9-bit mode.
_PWM16_PHASE_CORRECT_MODE_10BIT	Selects Phase Correct, 10-bit mode.
_PWM16_FAST_MODE_8BIT	Selects Fast, 8-bit mode.
_PWM16_FAST_MODE_9BIT	Selects Fast, 9-bit mode.
_PWM16_FAST_MODE_10BIT	Selects Fast, 10-bit mode.
_PWM16_PRESCALER_16bit_1	Sets prescaler value to 1 (No prescaling).
_PWM16_PRESCALER_16bit_8	Sets prescaler value to 8.
_PWM16_PRESCALER_16bit_64	Sets prescaler value to 64.
_PWM16_PRESCALER_16bit_256	Sets prescaler value to 256.
_PWM16_PRESCALER_16bit_1024	Sets prescaler value to 1024.
_PWM16_INVERTED	Selects the inverted PWM-16bit mode.
_PWM16__NON_INVERTED	Selects the normal (non inverted) PWM-16bit mode.

`_TIMER1`	Selects the Timer/Counter1 (used with PWM16bit_Start and PWM16bit_Stop.
`_TIMER3`	Selects the Timer/Counter3 (used with PWM16bit_Start and PWM16bit_Stop.
`_TIMER1_CH_A`	Selects the channel A on Timer/Counter1 (used with PWM16bit_Change_Duty).
`_TIMER1_CH_B`	Selects the channel B on Timer/Counter1 (used with PWM16bit_Change_Duty).
`_TIMER1_CH_C`	Selects the channel C on Timer/Counter1 (used with PWM16bit_Change_Duty).
`_TIMER3_CH_A`	Selects the channel A on Timer/Counter3 (used with PWM16bit_Change_Duty).
`_TIMER3_CH_B`	Selects the channel B on Timer/Counter3 (used with PWM16bit_Change_Duty).
`_TIMER3_CH_C`	Selects the channel C on Timer/Counter3 (used with PWM16bit_Change_Duty).

Note: Not all of the MCUs have 16bit PWM, and not all of the MCUs have both Timer/Counter1 and Timer/Counter3. Sometimes, like its the case with ATmega168, MCU has only Timer/Counter1 and channels A and B. Therefore constants that have in their name Timer3 or channel C are invalid (for ATmega168) and will not be visible from Code Assistant. It is highly advisable to use this feature, since it handles all the constants (available) and eliminates any chance of typing error.

PWM16bit_Init

Prototype	`procedure PWM16bit_Init(wave_mode : byte; prescaler : byte; inverted : byte; duty : word; timer : byte);`
Returns	Nothing.
Description	Initializes the PWM module. Parameter wave_mode is a desired PWM-16bit mode. There are several modes included : - PWM, Phase Correct, 8-bit - PWM, Phase Correct, 9-bit - PWM, Phase Correct, 10-bit - Fast PWM, 8-bit - Fast PWM, 9-bit - Fast PWM, 10-bit Parameter prescaler chooses prescale value N = 1,8,64,256 or 1024 (some modules support 32 and 128, but for this you will need to check the datasheet for the desired MCU). Paremeter inverted is for choosing between inverted and non inverted PWM signal. Parameter duty sets duty ratio from 0 to TOP value (this value varies on the PWM wave mode selected). PWM signal graphs and formulas are shown below. FAST MODE $f_{pwm} = \frac{f_{clk\ i/o}}{N \cdot (1+TOP)}$ 

Description	 PHASE MODE $f_{pwm} = \frac{f_{clk\ i/o}}{2 \cdot N \cdot TOP}$ TOP Duty Ratio 0 tPWM 2tPWM 3tPWM 4tPWM 5tPWM 6tPWM 7tPWM t Non Inverted Inverted The N variable represents the `prescaler` factor (1, 8, 64, 256, or 1024). PWM16bit_Init must be called before using other functions from PWM Library.

Requires	You need a CMO on the given MCU (that supports PWM-16bit).
	Before calling this routine you must set the output pin for the PWM (according to the datasheet):
	`DDRB.B1 = 1; // set PORTB pin 1 as output for the PWM-16bit`
	This code example is for ATmega168, for different MCU please consult datasheet for the correct pinout of the PWM module or modules.

Example	Initialize PWM-16bit module: `PWM16bit_Init(_PWM16_PHASE_CORRECT_MODE_8BIT, _PWM16_PRESCALER_16bit_8, _PWM16_NON_INVERTED, 255, _TIMER1);`

PWM16bit_Change_Duty

<table>
<tr><td><b>Prototype</b></td><td><code>procedure PWM16bit_Change_Duty(duty : word; channel : byte);</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>Changes PWM duty ratio. Parameter duty takes values shown on the table below. Where 0 is 0%, and TOP value is 100% duty ratio. Other specific values for duty ratio can be calculated as (Percent*TOP)/100.
<table>
<tr><th>Timer/Counter Mode of Operation :</th><th>TOP :</th><th>Update of OCRnX at :</th><th>TOVn Flag Set on :</th></tr>
<tr><td>PWM, Phase Correct, 8 bit</td><td>0x00FF</td><td>TOP</td><td>BOTTOM</td></tr>
<tr><td>PWM, Phase Correct, 9 bit</td><td>0x01FF</td><td>TOP</td><td>BOTTOM</td></tr>
<tr><td>PWM, Phase Correct, 10 bit</td><td>0x03FF</td><td>TOP</td><td>BOTTOM</td></tr>
<tr><td>Fast PWM, 8 bit</td><td>0x00FF</td><td>TOP</td><td>TOP</td></tr>
<tr><td>Fast PWM, 9 bit</td><td>0x01FF</td><td>TOP</td><td>TOP</td></tr>
<tr><td>Fast PWM, 10 bit</td><td>0x03FF</td><td>TOP</td><td>TOP</td></tr>
</table></td></tr>
<tr><td><b>Requires</b></td><td>PWM module must to be initialised (PWM16bit_Init) before using PWM_Set_Duty function.</td></tr>
<tr><td><b>Example</b></td><td>Example lets set duty ratio to :

<code>PWM16bit_Change_Duty(300, _TIMER1_CH_A);</code></td></tr>
</table>

PWM16bit_Start

Prototype	`procedure PWM16bit_Start(timer : byte);`
Returns	Nothing.
Description	Starts PWM-16bit module with alredy preset values (wave mode, prescaler, inverted and duty) given in the PWM16bit_Init.
Requires	MCU must have CMO module to use this library. PWM16bit_Init must be called before using this routine, otherwise it will have no effect as the PWM module is not initialised.

Example	`PWM16bit_Start(_TIMER1 );        // Starts the PWM-16bit module on Timer/Counter1   or`
	
	`PWM16bit_Start(_TIMER3 );        // Starts the PWM-16bit module on Timer/Counter3`

PWM16bit_Stop

Prototype	`procedure PWM16_Stop(timer : byte);`
Returns	Nothing.
Description	Stops the PWM-16bit module, connected to Timer/Counter set in this stop function.
Requires	MCU must have CMO module to use this library. Like in PWM16bit_Start before, PWM16bit_Init must be called before using this routine , otherwise it will have no effect as the PWM module is not running.

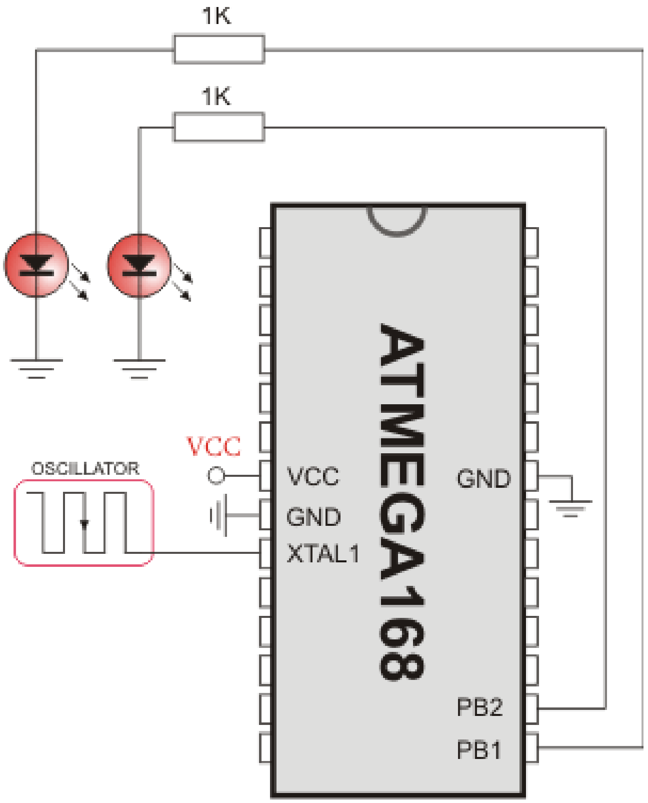
Example	`PWM16bit_Stop(_TIMER1 );         // Stops the PWM-16bit module on Timer/Counter1      or`
	`PWM16bit_Stop(_TIMER3 );         // Stops the PWM-16bit module on Timer/Counter3`

Library Example

The example changes PWM duty ratio continually by pressing buttons on PORTC (0-3). If LED is connected to PORTB.1 or PORTB.2 ,you can observe the gradual change of emitted light. This example is written for ATmega168. This AVR MCU has only Timer/Counter1 split over two channels A and B. In this example we are changing the duty ratio on both of these channels.

```
program PWM16bit_Test;

var current_duty : byte;
     current_duty1 : byte;

begin
   DDRC.B0  := 0;                      // Set PORTC pin 0 as input
   DDRC.B1  := 0;                      // Set PORTC pin 1 as input

   DDRC.B2  := 0;                      // Set PORTC pin 2 as input
   DDRC.B3  := 0;                      // Set PORTC pin 3 as input

   current_duty := 255;               // initial value for current_duty
   current_duty1 := 255;              // initial value for current_duty

   DDRB.B1  := 1;                      // Set PORTB pin 1 as output pin for the PWM
(according to datasheet)
   DDRB.B2  := 1;                    // Set PORTB pin 2 as output pin for the PWM
(according to datasheet)
PWM16bit_Init(_PWM16_FAST_MODE_9BIT, _PWM16_PRESCALER_16bit_1, _PWM16_INVERTED,
255, 1);

   while TRUE do
     begin
    if (PINC.B0 <> 0) then      // Detect if PORTC pin 0 is pressed
 begin
 Delay_ms(40);                               // Small delay to avoid
deboucing effect
          Inc(current_duty);                  // Increment duty ratio
          PWM16bit_Set_Duty(current_duty);    // Set incremented duty
         end
       else
        if (PINC.B1 <> 0) then    // Detect if PORTC pin 1 is pressed
           begin
             Delay_ms(40);                    // Small delay to avoid
deboucing effect
            Dec(current_duty);                // Decrement duty ratio
              PWM16bit_Set_Duty(current_duty);     // Set decremented
duty ratio
           end
         else
          if (PINC.B2 <> 0) then  // Detect if PORTC pin 2 is pressed
             begin
               Delay_ms(40);                            // Small delay
to avoid deboucing effect
              Inc(current_duty1);             // Increment duty ratio
                PWM16bit_Set_Duty(current_duty1);      // Set incre-
mented duty
            end
          else
           if (PINC.B3 <> 0) then// Detect if PORTC pin 3 is pressed
              begin
               Delay_ms(40);// Small delay to avoid deboucing effect
                Dec(current_duty1);           // Decrement duty ratio
                   PWM16bit_Set_Duty(current_duty1);   // Set decre-
mented duty ratio
                end;
    end;
end.
```

HW Connection



PWM demonstration

RS-485 LIBRARY

RS-485 is a multipoint communication which allows multiple devices to be connected to a single bus. The mikroPascal PRO for AVR provides a set of library routines for comfortable work with RS485 system using Master/Slave architecture. Master and Slave devices interchange packets of information. Each of these packets contains synchronization bytes, CRC byte, address byte and the data. Each Slave has unique address and receives only packets addressed to it. The Slave can never initiate communication.

It is the user's responsibility to ensure that only one device transmits via 485 bus at a time.

The RS-485 routines require the UART module. Pins of UART need to be attached to RS-485 interface transceiver, such as LTC485 or similar (see schematic at the bottom of this page).

Library constants:

- START byte value = 150
- STOP byte value = 169
- Address 50 is the broadcast address for all Slaves (packets containing address 50 will be received by all Slaves except the Slaves with addresses 150 and 169).

Note:

- Prior to calling any of this library routines, UART_Wr_Ptr needs to be initialized with the appropriate UART_Write routine.
- Prior to calling any of this library routines, UART_Rd_Ptr needs to be initialized with the appropriate UART_Read routine.
- Prior to calling any of this library routines, UART_Rdy_Ptr needs to be initialized with the appropriate UART_Ready routine.
- Prior to calling any of this library routines, UART_TX_Idle_Ptr needs to be initialized with the appropriate UART_TX_Idle routine.

External dependencies of RS-485 Library

The following variables must be defined in all projects using RS-485 Library:	Description:	Example :
`var RS485_rxtx_pin : sbit; sfr; external;`	Control RS-485 Transmit/Receive operation mode	`var RS485_rxtx_pin : sbit at PORTD.B2;`
`var RS485_rxtx_pin_direction : sbit; sfr; external;`	Direction of the RS-485 Transmit/Receive pin	`var RS485_rxtx_pin_direction : sbit at DDRD.B2;`

Library Routines

- RS485Master_Init
- RS485Master_Receive
- RS485Master_Send
- RS485Slave_Init
- RS485Slave_Receive
- RS485Slave_Send

RS485Master_Init

Prototype	`procedure RS485Master_Init();`
Returns	Nothing.
Description	Initializes MCU as a Master for RS-485 communication.
Requires	Global variables : - `RS485_rxtx_pin` - this pin is connected to RE/DE input of RS-485 transceiver(see schematic at the bottom of this page). RE/DE signal controls RS-485 transceiver operation mode. - `RS485_rxtx_pin_direction` - direction of the RS-485 Transmit/Receive pin must be defined before using this function. UART HW module needs to be initialized. See UARTx_Init.

Example	`// RS485 module pinout`
	`var RS485_rxtx_pin : sbit  at PORTD.B2;`
	`var RS485_rxtx_pin_direction : sbit  at DDRD.B2;`
	`// End of RS485 module pinout`
	
	`// Pass pointers to UART functions of used UART module`
	`UART_Wr_Ptr := @UART1_Write;`
	`UART_Rd_Ptr := @UART1_Read;`
	`UART_Rdy_Ptr := @UART1_Data_Ready;`
	`UART_TX_Idle_Ptr := @UART1_TX_Idle;`
	`...`
	`UART1_Init(9600);                       // initialize UART module`
	`RS485Master_Init();                     // intialize MCU as a`
	`Master for RS-485 communication`

RS485Master_Receive

Prototype	`procedure RS485Master_Receive(var data_buffer: array[5] of byte);`
Returns	Nothing.
Description	Receives messages from Slaves. Messages are multi-byte, so this routine must be called for each byte received. Parameters : - `data_buffer`: 7 byte buffer for storing received data, in the following manner: - `data[0..2]` : message content - `data[3]` : number of message bytes received, 1–3 - `data[4]` : is set to 255 when message is received - `data[5]` : is set to 255 if error has occurred - `data[6]` : address of the Slave which sent the message The function automatically adjusts data[4] and data[5] upon every received message. These flags need to be cleared by software.
Requires	MCU must be initialized as a Master for RS-485 communication. See RS485Master_Init.

Example	`var msg : array[20] of byte;`
	`...`
	`RS485Master_Receive(msg);`

RS485Master_Send

Prototype	`procedure RS485Master_Send(var data_buffer: array[ 20] of byte; datalen: byte; slave_address: byte);`
Returns	Nothing.
Description	Sends message to Slave(s). Message format can be found at the bottom of this page. Parameters : - `data_buffer`: data to be sent - `datalen`: number of bytes for transmition. Valid values: 0...3. - `slave_address`: Slave(s) address
Requires	MCU must be initialized as a Master for RS-485 communication. See RS485Master_Init. It is the user's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`var msg : array[ 20] of byte;`
	`...`
	`// send 3 bytes of data to Slave with address 0x12`
	`RS485Master_Send(msg, 3, 0x12);`

RS485Slave_Init

Prototype	`procedure RS485Slave_Init(slave_address: byte);`
Returns	Nothing.
Description	Initializes MCU as a Slave for RS-485 communication. Parameters : - `slave_address`: Slave address
Requires	Global variables : - `RS485_rxtx_pin` - this pin is connected to RE/DE input of RS-485 transceiver(see schematic at the bottom of this page). RE/DE signal controls -- - RS-485 transceiver operation mode. Valid values: 1 (for transmitting) and 0 (for receiving) - `RS485_rxtx_pin_direction` - direction of the RS-485 Transmit/Receive pin must be defined before using this function. UART HW module needs to be initialized. See UARTx_Init.

Example	`// RS485 module pinout`
	`var RS485_rxtx_pin : sbit  at PORTD.B2;`
	`var RS485_rxtx_pin_direction : sbit  at DDRD.B2;`
	`// End of RS485 module pinout`
	
	`// Pass pointers to UART functions of used UART module`
	`UART_Wr_Ptr := @UART1_Write;`
	`UART_Rd_Ptr := @UART1_Read;`
	`UART_Rdy_Ptr := @UART1_Data_Ready;`
	`UART_TX_Idle_Ptr := @UART1_TX_Idle;`
	
	`...`
	`UART1_Init(9600);                      // initialize UART module`
	`RS485Slave_Init(160);                  // intialize MCU as a`
	`Slave for RS-485 communication with address 160`

RS485slave_Receive

Prototype	`procedure RS485Slave_Receive(var data_buffer: array[ 20] of byte);`
Returns	Nothing.
Description	Receives messages from Master. If Slave address and Message address field don't match then the message will be discarded. Messages are multi-byte, so this routine must be called for each byte received. Parameters : - `data_buffer`: 6 byte buffer for storing received data, in the following manner: - `data[ 0..2]` : message content - `data[ 3]` : number of message bytes received, 1–3 - `data[ 4]` : is set to 255 when message is received - `data[ 5]` : is set to 255 if error has occurred The function automatically adjusts data[4] and data[5] upon every received message. These flags need to be cleared by software.
Requires	MCU must be initialized as a Slave for RS-485 communication. See RS485Slave_Init.

Example	`var msg : array[ 20] of byte;`
	`...`
	`RS485Slave_Read(msg);`

RS485Slave_Send

Prototype	`procedure RS485Slave_Send(var data_buffer: array[ 20] of byte; datalen : byte);`
Returns	Nothing.
Description	Sends message to Master. Message format can be found at the bottom of this page. Parameters : - `data_buffer`: data to be sent - `datalen`: number of bytes for transmition. Valid values: 0...3.
Requires	MCU must be initialized as a Slave for `RS-485` communication. See `RS485Slave_Init`. It is the user's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`var msg : array[ 8] of byte;`
	`...`
	`// send 2 bytes of data to the Master`
	`RS485Slave_Send(msg, 2);`

Library Example

This is a simple demonstration of RS485 Library routines usage.

Master sends message to Slave with address 160 and waits for a response. The Slave accepts data, increments it and sends it back to the Master. Master then does the same and sends incremented data back to Slave, etc.

Master displays received data on PORTB, while error on receive (0xAA) and number of consecutive unsuccessful retries are displayed on PORTC. Slave displays received data on PORTB, while error on receive (0xAA) is displayed on PORTC. Hardware configurations in this example are made for the EasyAVR5A board and ATmega16.

RS485 Master code:

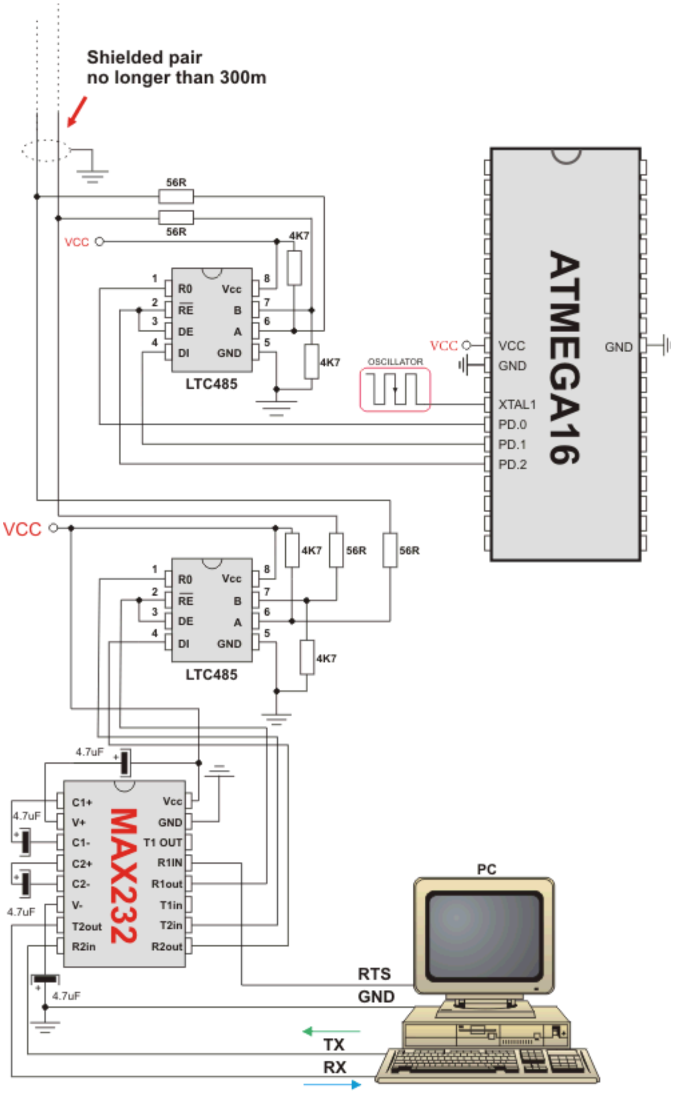
```
program RS485_Master_Example;

uses __Lib_RS485;

var dat : array[ 10] of byte ; // buffer for receving/sending messages
    i, j : byte;
    cnt : longint;

var rs485_rxtx_pin  : sbit at PORTD.B2;        // set transcieve pin
    rs485_rxtx_pin_direction : sbit at DDRD.B2;    // set transcieve
pin direction

// Interrupt routine
procedure interrupt(); org 0x16;
  begin
    RS485Master_Receive(dat);
  end;

begin
  cnt := 0;
  PORTA  := 0;                                // clear PORTA
  PORTB  := 0;                                // clear PORTB
  PORTC  := 0;                                // clear PORTC

  DDRA   := 0xFF;                             // set PORTA as output
  DDRB   := 0xFF;                             // set PORTB as output
  DDRC   := 0xFF;                             // set PORTB as output

  // Pass pointers to UART functions of used UART module
  UART_Wr_Ptr:= @UART1_Write;
  UART_Rd_Ptr := @UART1_Read;
  UART_Rdy_Ptr := @UART1_Data_Ready;
  UART_TX_Idle_Ptr := @UART1_TX_Idle;

  UART1_Init(9600);                       // initialize UART1 module
  Delay_ms(100);

  RS485Master_Init();                    // initialize MCU as Master
  dat[ 0]  := 0xAA;
  dat[ 1]  := 0xF0;
  dat[ 2]  := 0x0F;
  dat[ 4]  := 0;           // ensure that message received flag is 0
  dat[ 5]  := 0;          // ensure that error flag is 0
  dat[ 6]  := 0;

  RS485Master_Send(dat,1,160);

  SREG_I   := 1;                 // enable global interrupt
  RXCIE    := 1;                 // enable interrupt on UART receive

  while (TRUE) do
    begin                  // upon completed valid message receiving
                        //    data[ 4]  is set to 255
      Inc(cnt);
      if (dat[ 5] <> 0) then      // if an error detected, signal it
         PORTC := dat[ 5] ;                 //    by setting PORTC
      if (dat[ 4] <> 0) then     // if message received successfully
         begin
           cnt := 0;
           dat[ 4]  := 0;                 // clear message received flag
           j := dat[ 3] ;
           for i := 1 to dat[ 3]  do    // show data on PORTB
             PORTB := dat[ i-1] ;
           dat[ 0]  := dat[ 0] +1;        // increment received dat[ 0]
           Delay_ms(1);                   // send back to slave
           RS485Master_Send(dat,1,160);
         end;

if (cnt > 100000) then         // if in 100000 poll-cycles the answer
         begin
           Inc(PORTA);       //    was not detected, signal
           cnt := 0;         //    failure of send-message
 RS485Master_Send(dat,1,160);
          if (PORTA > 10) then          // if sending failed 10 times
             begin
               PORTA := 0;
                RS485Master_Send(dat,1,50);    //    send message on
broadcast address
             end;
         end;
    end;
end.
```

RS485 Slave code:

```
program RS485_Slave_Example;

uses __Lib_RS485;

var dat : array[ 20]  of byte;  // buffer for receving/sending messages
    i, j : byte;

var rs485_rxtx_pin : sbit at PORTD.B2;        // set transcieve pin
    rs485_rxtx_pin_direction : sbit at DDRD.B2;   // set transcieve
pin direction

// Interrupt routine
procedure interrupt(); org 0x16;
  begin
    RS485Slave_Receive(dat);

  end;

begin
  PORTB := 0;                           // clear PORTB
  PORTC := 0;                           // clear PORTC

  DDRB := 0xFF;                         // set PORTB as output
  DDRC := 0xFF;                         // set PORTB as output

  // Pass pointers to UART functions of used UART module
  UART_Wr_Ptr := @UART1_Write;
  UART_Rd_Ptr := @UART1_Read;
  UART_Rdy_Ptr := @UART1_Data_Ready;
  UART_TX_Idle_Ptr := @UART1_TX_Idle;

UART1_Init(9600);                    // initialize UART1 module
  Delay_ms(100);
RS485Slave_Init(160);         // Intialize MCU as slave, address 160

dat[ 4]  := 0;            // ensure that message received flag is 0
  dat[ 5]  := 0;          // ensure that message received flag is 0
  dat[ 6]  := 0;                     // ensure that error flag is 0

  SREG_I   := 1;                        // enable global interrupt
  RXCIE   := 1;                 // enable interrupt on UARTs receive

  while (TRUE) do
    begin
 if (dat[ 5]  <> 0)  then        // if an error detected, signal it by
        begin
PORTC := dat[ 5] ;           //   setting PORTC
```

```
 dat[ 5]  := 0;
         end;
      if (dat[ 4]  <> 0) then   // upon completed valid message receive
         begin
           dat[ 4]  := 0;                 //   data[ 4]  is set to 0xFF
           j := dat[ 3] ;

           for i := 1 to dat[ 3]  do    // show data on PORTB
             PORTB := dat[ i-1] ;

           dat[ 0]  := dat[ 0] +1;       // increment received dat[ 0]
           Delay_ms(1);
          RS485Slave_Send(dat,1);    //   and send it back to master
         end;
    end;
end.
```

HW Connection



Example of interfacing PC to AVR MCU via RS485 bus with LTC485 as RS-485 transceiver

Message format and CRC calculations

Q: How is CRC checksum calculated on RS485 master side?

```
START_BYTE := 0x96; // 10010110
STOP_BYTE  := 0xA9; // 10101001

PACKAGE:
--------
START_BYTE 0x96
ADDRESS
DATALEN
[ DATA1]              // if exists
[ DATA2]              // if exists
[ DATA3]              // if exists
CRC
STOP_BYTE  0xA9


DATALEN bits
------------
bit7 := 1  MASTER SENDS
        0  SLAVE  SENDS
bit6 := 1  ADDRESS WAS XORed with 1, IT WAS EQUAL TO START_BYTE or
STOP_BYTE
        0  ADDRESS UNCHANGED
bit5 := 0  FIXED
bit4 := 1  DATA3 (if exists) WAS XORed with 1, IT WAS EQUAL TO
START_BYTE or STOP_BYTE
        0  DATA3 (if exists) UNCHANGED
bit3 := 1  DATA2 (if exists) WAS XORed with 1, IT WAS EQUAL TO
START_BYTE or STOP_BYTE
        0  DATA2 (if exists) UNCHANGED
bit2 := 1  DATA1 (if exists) WAS XORed with 1, IT WAS EQUAL TO
START_BYTE or STOP_BYTE
        0  DATA1 (if exists) UNCHANGED
bit1bit0 := 0 to 3 NUMBER OF DATA BYTES SEND

CRC generation :
----------------
crc_send := datalen xor address;
crc_send := crc_send xor data[ 0] ;     // if exists
crc_send := crc_send xor data[ 1] ;     // if exists
crc_send := crc_send xor data[ 2] ;     // if exists
crc_send := not crc_send;
if ((crc_send = START_BYTE) or (crc_send = STOP_BYTE)) then
   Inc(crc_send);

NOTE:  DATALEN<4..0>  can  not  take  the  START_BYTE<4..0>  or
STOP_BYTE<4..0> values.
```

SOFTWARE I²C LIBRARY

The mikroPascal PRO for AVR provides routines for implementing Software I^{2}C communication. These routines are hardware independent and can be used with any MCU. The Software I^{2}C library enables you to use MCU as Master in I^{2}C communication. Multi-master mode is not supported.

Note: This library implements time-based activities, so interrupts need to be disabled when using Software I^{2}C.

Note: All Software I^{2}C Library functions are blocking-call functions (they are waiting for I^{2}C clock line to become logical one).

Note: The pins used for Software I^{2}C communication should be connected to the pull-up resistors. Turning off the LEDs connected to these pins may also be required.

External dependencies of Soft_I2C Library

The following variables must be defined in all projects using Sound Library:	Description:	Example :
`var Soft_I2C_Scl_Output : sbit; sfr; external;`	Soft I^{2}C Clock output line.	`var Soft_I2C_Scl_Output : sbit at PORTC.B0;`
`var Soft_I2C_Sda_Output : sbit; sfr; external;`	Soft I^{2}C Data output line.	`var Soft_I2C_Sda_Output : sbit at PORTC.B1;`
`var Soft_I2C_Scl_Input : sbit; sfr; external;`	Soft I^{2}C Clock input line.	`var Soft_I2C_Scl_Input : sbit at PINC.B0;`
`var Soft_I2C_Sda_Input : sbit; sfr; external;`	Soft I^{2}C Data input line.	`var Soft_I2C_Sda_Input : sbit at PINC.B1;`
`var Soft_I2C_Scl_Pin_Direction : sbit; sfr; external;`	Direction of the Soft I^{2}C Clock pin.	`var Soft_I2C_Scl_Pin_Direction : sbit at DDRC.B0;`
`var Soft_I2C_Sda_Pin_Direction : sbit; sfr; external;`	Direction of the Soft I^{2}C Data pin.	`var Soft_I2C_Sda_Pin_Direction : sbit at DDRC.B1;`

Library Routines

- Soft_I2C_Init
- Soft_I2C_Start
- Soft_I2C_Read
- Soft_I2C_Write
- Soft_I2C_Stop
- Soft_I2C_Break

Soft_I2C_Init

Prototype	`procedure Soft_I2C_Init();`
Returns	Nothing.
Description	Configures the software I^{2}C module.
Requires	Global variables : - `Soft_I2C_Scl_Output`: Soft I^{2}C clock output line - `Soft_I2C_Sda_Output`: Soft I^{2}C data output line - `Soft_I2C_Scl_Input`: Soft I^{2}C clock input line - `Soft_I2C_Sda_Input`: Soft I^{2}C data input line - `Soft_I2C_Scl_Pin_Direction`: Direction of the Soft I^{2}C clock pin - `Soft_I2C_Sda_Pin_Direction`: Direction of the Soft I^{2}C data pin must be defined before using this function.

Example	`// Soft_I2C pinout definition`
	`var Soft_I2C_Scl_Output        : sbit at PORTC.B0;`
	`var Soft_I2C_Sda_Output        : sbit at PORTC.B1;`
	`var Soft_I2C_Scl_Input         : sbit at PINC.B0;`
	`var Soft_I2C_Sda_Input         : sbit at PINC.B1;`
	`var Soft_I2C_Scl_Pin_Direction : sbit at DDRC.B0;`
	`var Soft_I2C_Sda_Pin_Direction : sbit at DDRC.B1;`
	`// End of Soft_I2C pinout definition`
	`...`
	`Soft_I2C_Init();`

Soft_I2C_Start

Prototype	`procedure Soft_I2C_Start();`
Returns	Nothing.
Description	Determines if the I^{2}C bus is free and issues START signal.
Requires	Software I^{2}C must be configured before using this function. See Soft_I2C_Init routine.

Example	`// Issue START signal`
	`Soft_I2C_Start();`

Soft_I2C_Read

Prototype	`function Soft_I2C_Read(ack: word): byte;`
Returns	One byte from the Slave.
Description	Reads one byte from the slave. Parameters : - `ack`: acknowledge signal parameter. If the `ack==0` not acknowledge signal will be sent after reading, otherwise the acknowledge signal will be sent.
Requires	Soft I^{2}C must be configured before using this function. See Soft_I2C_Init routine. Also, START signal needs to be issued in order to use this function. See Soft_I2C_Start routine.

Example	`var take : word;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take := Soft_I2C_Read(0);`

Soft_I2C_Write

Prototype	`function Soft_I2C_Write(_data: byte): byte;`
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I^{2}C bus.
Description	Sends data byte via the I^{2}C bus. Parameters : - `_Data`: data to be sent
Requires	Soft I^{2}C must be configured before using this function. See Soft_I2C_Init routine. Also, START signal needs to be issued in order to use this function. See Soft_I2C_Start routine.

Example	`var _data, error : byte;`
	`...`
	`error := Soft_I2C_Write(data);`
	`error := Soft_I2C_Write(0xA3);`

Soft_I2C_Stop

Prototype	`procedure Soft_I2C_Stop();`
Returns	Nothing.
Description	Issues STOP signal.
Requires	Soft I^{2}C must be configured before using this function. See Soft_I2C_Init routine.

Example	`// Issue STOP signal`
	`Soft_I2C_Stop();`

Soft_I2C_Break

<table>
<tr><td><b>Prototype</b></td><td><code>procedure Soft_I2C_Break();</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>All Software I2C Library functions can block the program flow (see note at the top of this page). Call this routine from interrupt to unblock the program execution. This mechanism is similar to WDT.
<b>Note</b>: Interrupts should be disabled before using Software I2C routins again (see note at the top of this page).</td></tr>
<tr><td><b>Requires</b></td><td>Nothing.</td></tr>
<tr><td><b>Example</b></td><td>

```
// Soft_I2C pinout definition
var Soft_I2C_Scl_Output         : sbit at PORTC.B0;
var Soft_I2C_Sda_Output         : sbit at PORTC.B1;
var Soft_I2C_Scl_Input          : sbit at PINC.B0;
var Soft_I2C_Sda_Input          : sbit at PINC.B1;
var Soft_I2C_Scl_Pin_Direction : sbit at DDRC.B0;
var Soft_I2C_Sda_Pin_Direction : sbit at DDRC.B1;
// End of Soft_I2C pinout definition

var counter : byte;

procedure Timer0Overflow_ISR(); org 0x12;
begin
  counter := 0;
  if (counter >= 20)
    begin
      Soft_I2C_Break();
      counter := 0;                  // reset counter
    end
  else
    Inc(counter);                    // increment counter
end;

begin

  TOIE0_bit   := 1;         // Timer0 overflow interrupt enable
  TCCR0_bit   := 5;         // Start timer with 1024 prescaler

  SREG_I_bit := 0;                  // Interrupt disable

  ...

  // try Soft_I2C_Init with blocking prevention mechanism
  SREG_I_bit := 1;                  // Interrupt enable
  Soft_I2C_Init();
  SREG_I_bit := 0;                  // Interrupt disable

  ...

end.
```

</td></tr>
</table>

Library Example

The example demonstrates Software I²C Library routines usage. The AVR MCU is connected (SCL, SDA pins) to PCF8583 RTC (real-time clock). Program reads date and time are read from the RTC and prints it on Lcd.

```
program RTC_Read;

var seconds, minutes, hours, day, month, year : byte;    // Global
date/time variables

// Software I2C connections
var Soft_I2C_Scl_Output    : sbit at PORTC.B0;
     Soft_I2C_Sda_Output    : sbit at PORTC.B1;
     Soft_I2C_Scl_Input     : sbit at PINC.B0;
     Soft_I2C_Sda_Input     : sbit at PINC.B1;
     Soft_I2C_Scl_Direction : sbit at DDRC.B0:
     Soft_I2C_Sda_Direction : sbit at DDRC.B1;
// End Software I2C connections

// Lcd module connections
var LCD_RS : sbit at PORTD.B2;
     LCD_EN : sbit at PORTD.B3;
     LCD_D4 : sbit at PORTD.B4;
     LCD_D5 : sbit at PORTD.B5;
     LCD_D6 : sbit at PORTD.B6;
     LCD_D7 : sbit at PORTD.B7;
     LCD_RS_Direction : sbit at DDRD.B2;
     LCD_EN_Direction : sbit at DDRD.B3
     LCD_D4_Direction : sbit at DDRD.B4;
     LCD_D5_Direction : sbit at DDRD.B5;
     LCD_D6_Direction : sbit at DDRD.B6;
     LCD_D7_Direction : sbit at DDRD.B7;
// End Lcd module connections

//--------------------- Reads time and date information from RTC
(PCF8583)
procedure Read_Time();
  begin
     Soft_I2C_Start();       // Issue start signal
    Soft_I2C_Write(0xA0);    // Address PCF8583, see PCF8583 datasheet
     Soft_I2C_Write(2);      // Start from address 2
     Soft_I2C_Start();       // Issue repeated start signal
     Soft_I2C_Write(0xA1);  // Address PCF8583 for reading R/W=1}

     seconds := Soft_I2C_Read(1);        // Read seconds byte
     minutes := Soft_I2C_Read(1);        // Read minutes byte
hours := Soft_I2C_Read(1);               // Read hours byte
day := Soft_I2C_Read(1);                 // Read year/day byte
```

```
    month := Soft_I2C_Read(0);       // Read weekday/month byte}
    Soft_I2C_Stop();                  // Issue stop signal}
  end;

//-------------------- Formats date and time
procedure Transform_Time() ;
  begin
    seconds  :=  ((seconds and 0xF0) shr 4)*10 + (seconds and 0x0F);
// Transform seconds
    minutes  :=  ((minutes and 0xF0) shr 4)*10 + (minutes and 0x0F);
// Transform months
    hours    :=  ((hours and 0xF0)  shr 4)*10  + (hours and 0x0F);
// Transform hours
    year     :=   (day and 0xC0) shr 6;            // Transform year
     day       :=  ((day and 0x30) shr 4)*10     + (day and 0x0F);
// Transform day
     month     :=  ((month and 0x10)  shr 4)*10 + (month and 0x0F);
// Transform month
  end;

//-------------------- Output values to Lcd
procedure Display_Time();
  begin
     Lcd_Chr(1, 6, (day / 10)   + 48);    // Print tens digit of day
variable
      Lcd_Chr(1, 7, (day mod 10)   + 48);  // Print oness digit of
day variable
      Lcd_Chr(1, 9, (month / 10) + 48);
      Lcd_Chr(1,10, (month mod 10) + 48);
      Lcd_Chr(1,15,  year         + 56);    // Print year vaiable +
8 (start from year 2008)

      Lcd_Chr(2, 6, (hours / 10)   + 48);
      Lcd_Chr(2, 7, (hours mod 10)   + 48);
      Lcd_Chr(2, 9, (minutes / 10) + 48);
      Lcd_Chr(2,10, (minutes mod 10) + 48);
      Lcd_Chr(2,12, (seconds / 10) + 48);
      Lcd_Chr(2,13, (seconds mod 10) + 48);
  end;
//------------------ Performs project-wide init
procedure Init_Main();
  begin
    Soft_I2C_Init();            // Initialize Soft I2C communication

    Lcd_Init();                  // Initialize Lcd
    Lcd_Cmd(LCD_CLEAR);         // Clear Lcd display
    Lcd_Cmd(LCD_CURSOR_OFF);    // Turn cursor off

LCD_Out(1,1,'Date:');       // Prepare and output static text on Lcd
    LCD_Chr(1,8,':');
```

```
      LCD_Chr(1,11,':');
      LCD_Out(2,1,'Time:');
      LCD_Chr(2,8,':');
      LCD_Chr(2,11,':');
      LCD_Out(1,12,'200');
    end;

  //----------------- Main procedure
    begin
     Init_Main();                   // Perform initialization

      while TRUE do                  // Endless loop
        begin
          Read_Time();               // Read time from RTC(PCF8583)
          Transform_Time();         // Format date and time
          Display_Time();           // Prepare and display on Lcd}
          Delay_ms(1000);             // Wait 1 second
        end;
    end.
```

SOFTWARE SPI LIBRARY

The mikroPacal PRO for AVR provides routines for implementing Software SPI communication. These routines are hardware independent and can be used with any MCU. The Software SPI Library provides easy communication with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

Library configuration:

- SPI to Master mode
- Clock value = 20 kHz.
- Data sampled at the middle of interval.
- Clock idle state low.
- Data sampled at the middle of interval.
- Data transmitted at low to high edge.

Note: The Software SPI library implements time-based activities, so interrupts need to be disabled when using it.

External dependencies of Software SPI Library

The following variables must be defined in all projects using Software SPI Library:	Description:	Example :
`var Chip_Select : sbit; sfr; external;`	Chip select line.	`var Chip_Select : sbit at PORTB.B0;`
`var SoftSpi_SDI : sbit; sfr; external;`	Data In line.	`var SoftSpi_SDI : sbit at PINB.B6;`
`var SoftSpi_SDO : sbit; sfr; external;`	Data Out line.	`var SoftSpi_SDO : sbit at PORTB.B5;`
`var SoftSpi_CLK : sbit; sfr; external;`	Clock line.	`var SoftSpi_CLK : sbit at PORTB.B7;`
`var Chip_Select_Direction : sbit; sfr; external;`	Direction of the Chip Select pin.	`var Chip_Select_Direction : sbit at DDRB.B0;`
`var SoftSpi_SDI_Direction : sbit; sfr; external;`	Direction of the Data In pin.	`var SoftSpi_SDI_Direction : sbit at DDRB.B6;`
`var SoftSpi_SDO_Direction : sbit; sfr; external;`	Direction of the Data Out pin	`var SoftSpi_SDO_Direction : sbit at DDRB.B5;`
`var SoftSpi_CLK_Direction : sbit; sfr; external;`	Direction of the Clock pin.	`var SoftSpi_CLK_Direction : sbit at DDRB.B7;`

Library Routines

- Soft_SPI_Init
- Soft_SPI_Read
- Soft_SPI_Write

Soft_SPI_Init

Prototype	`procedure Soft_SPI_Init();`
Returns	Nothing.
Description	Configures and initializes the software SPI module.
Requires	Global variables: - `Chip_Select`: Chip select line - `SoftSpi_SDI`: Data in line - `SoftSpi_SDO`: Data out line - `SoftSpi_CLK`: Data clock line - `Chip_Select_Direction`: Direction of the Chip select pin - `SoftSpi_SDI_Direction`: Direction of the Data in pin - `SoftSpi_SDO_Direction`: Direction of the Data out pin - `SoftSpi_CLK_Direction`: Direction of the Data clock pin must be defined before using this function.

Example	`// soft_spi pinout definition`
	`var Chip_Select : sbit at PORTB.B0;`
	`var SoftSpi_SDI : sbit at PINB.B6;`
	`var SoftSpi_SDO : sbit at PORTB.B5;`
	`var SoftSpi_CLK : sbit at PORTB.B7;`
	`var Chip_Select_Direction : sbit at DDRB.B0;`
	`var SoftSpi_SDI_Direction : sbit at DDRB.B6;`
	`var SoftSpi_SDO_Direction : sbit at DDRB.B5;`
	`var SoftSpi_CLK_Direction : sbit at DDRB.B7;`
	`...`
	`Soft_SPI_Init(); // Init Soft_SPI`

Soft_SPI_Read

Prototype	`function Soft_SPI_Read(sdata: byte): word;`
Returns	Byte received via the SPI bus.
Description	This routine performs 3 operations simultaneously. It provides clock for the Software SPI bus, reads a byte and sends a byte. Parameters : - `sdata`: data to be sent.
Requires	Soft SPI must be initialized before using this function. See Soft_SPI_Init routine.

Example	`var data_read : word;`
	`     data_send : byte;`
	`...`
	`// Read a byte and assign it to data_read variable`
	`// (data_send byte will be sent via SPI during the Read operation)`
	`data_read := Soft_SPI_Read(data_send);`

Soft_SPI_Write

Prototype	`procedure Soft_SPI_Write(sdata: byte);`
Returns	Nothing.
Description	This routine sends one byte via the Software SPI bus. Parameters : - `sdata`: data to be sent.
Requires	Soft SPI must be initialized before using this function. See Soft_SPI_Init routine.

Example	`// Write a byte to the Soft SPI bus`
	`Soft_SPI_Write(0xAA);`

Library Example

This code demonstrates using library routines for Soft_SPI communication. Also, this example demonstrates working with Microchip's MCP4921 12-bit D/A converter.

```
program Soft_SPI;

// DAC module connections
var Chip_Select : sbit at PORTB.B0;
    SoftSpi_CLK : sbit at PORTB.B7;
    SoftSpi_SDI : sbit at PINB.B6;    // Note: Input signal
    SoftSpi_SDO : sbit at PORTB.B5;

var Chip_Select_Direction : sbit at DDRB.B0;
    SoftSpi_CLK_Direction : sbit at DDRB.B7;
    SoftSpi_SDI_Direction : sbit at DDRB.B6;
    SoftSpi_SDO_Direction : sbit at DDRB.B5;
// End DAC module connections

var value : word;

procedure InitMain();
  begin
    DDA0 := 0;                                 // Set PA0 pin as input
    DDA1 := 0;                                 // Set PA1 pin as input
    Chip_Select := 1;                          // Deselect DAC
    Chip_Select_Direction := 1;                // Set CS# pin as Output
    Soft_SPI_Init();                           // Initialize Soft_SPI
  end;

// DAC increments (0..4095) --> output voltage (0..Vref)
procedure DAC_Output( valueDAC : word);
var temp : byte;
  begin
    Chip_Select := 0;                              // Select DAC chip

    // Send High Byte
temp := word(valueDAC shr 8) and 0x0F;    // Store valueDAC[ 11..8] to
temp[ 3..0]
     temp := temp or 0x30;                // Define DAC setting, see
MCP4921 datasheet
    Soft_SPI_Write(temp);            // Send high byte via Soft SPI

    // Send Low Byte
temp := valueDAC;              // Store valueDAC[ 7..0] to temp[ 7..0]
    Soft_SPI_Write(temp);              // Send low byte via Soft SPI

    Chip_Select := 1;                              // Deselect DAC chip
  end;
```

```
begin

    InitMain();                       // Perform main initialization

    value := 2048;                    // When program starts, DAC gives
                                    //    the output in the mid-range
    while (TRUE) do                   // Endless loop
      begin

        if ((PINA.B0) and (value < 4095)) then      // If PA0 button
is pressed
          Inc(value)                              //    increment value
        else
          begin
             if ((PINA.B1) and (value > 0)) then     // If PA1 but-
ton is pressed
               Dec(value);                        //    decrement value
          end;

        DAC_Output(value);                // Send value to DAC chip
        Delay_ms(1);                      // Slow down key repeat pace
      end;
  end.
```

SOFTWARE UART LIBRARY

The mikroPascal PRO for AVR provides routines for implementing Software UART communication. These routines are hardware independent and can be used with any MCU. The Software UART Library provides easy communication with other devices via the RS232 protocol.

Note: The Software UART library implements time-based activities, so interrupts need to be disabled when using it.

External dependencies of Software UART Library

The following variables must be defined in all projects using Software UART Library:	Description:	Example :
`var Soft_UART_Rx_Pin : sbit; sfr; external;`	Receive line.	`var Soft_UART_Rx_Pin : sbit at PIND.B0;`
`var Soft_UART_Tx_Pin : sbit; sfr; external;`	Transmit line.	`var Soft_UART_Tx_Pin : sbit at PORTD.B1;`
`var Soft_UART_Rx_Pin_Direction : sbit; sfr; external;`	Direction of the Receive pin.	`var Soft_UART_Rx_Pin_Direction : sbit at DDRD.B0;`
`var Soft_UART_Tx_Pin_Direction : sbit; sfr; external;`	Direction of the Transmit pin.	`var Soft_UART_Tx_Pin_Direction : sbit at DDRD.B1;`

Library Routines

- Soft_UART_Init
- Soft_UART_Read
- Soft_UART_Write
- Soft_UART_Break

Soft_UART_Init

Prototype	`function Soft_UART_Init(baud_rate: dword; inverted: byte): byte;`
Returns	- `2` - error, requested baud rate is too low - `1` - error, requested baud rate is too high - `0` - successfull initialization
Description	Configures and initializes the software UART module. Parameters : - `baud_rate`: baud rate to be set. Maximum baud rate depends on the MCU's clock and working conditions. - `inverted`: inverted output flag. When set to a non-zero value, inverted logic on output is used. Software UART routines use Delay_Cyc routine. If requested baud rate is too low then calculated parameter for calling `Delay_Cyc` exceeeds Delay_Cyc argument range. If requested baud rate is too high then rounding error of `Delay_Cyc` argument corrupts Software UART timings.
Requires	Global variables: - `Soft_UART_Rx_Pin`: Receiver pin - `Soft_UART_Tx_Pin`: Transmiter pin - `Soft_UART_Rx_Pin_Direction`: Direction of the Receiver pin - `Soft_UART_Tx_Pin_Direction`: Direction of the Transmiter pin must be defined before using this function.

Example	`// Soft UART connections`
	`var Soft_UART_Rx_Pin           : sbit at PIND.B0;`
	`var Soft_UART_Tx_Pin           : sbit at PORTD.B1;`
	`var Soft_UART_Rx_Pin_Direction : sbit at DDRD.B0;`
	`var Soft_UART_Tx_Pin_Direction : sbit at DDRD.B1;`
	`// Soft UART connections`
	`...`
	`// Initialize Software UART communication on pins Rx, Tx, at 9600 bps`
	`Soft_UART_Init(9600, 0);`

Soft_UART_Read

Prototype	`function Soft_UART_Read(var error: byte): byte;`
Returns	Byte received via UART.
Description	The function receives a byte via software UART. This is a blocking function call (waits for start bit). Programmer can unblock it by calling Soft_UART_Break routine. Parameters : - `error`: Error flag. Error code is returned through this variable. `0` - no error `1` - stop bit error `255` - user abort, Soft_UART_Break called
Requires	Software UART must be initialized before using this function. See the Soft_UART_Init routine.

Example	`var data  : byte;`
	`    error : byte;`
	`...`
	`// wait until data is received`
	`repeat`
	`  data := Soft_UART_Read(error);`
	`until (error=0);`
	
	`// Now we can work with data:`
	`if ( data ) then`
	`begin`
	`...`
	`end`

Soft_UART_Write

Prototype	`procedure Soft_UART_Write(udata: byte);`
Returns	Nothing.
Description	This routine sends one byte via the Software UART bus. Parameters : - `udata`: data to be sent.
Requires	Software UART must be initialized before using this function. See the Soft_UART_Init routine. 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	`var some_byte : byte;`
	`...`
	`// Write a byte via Soft UART`
	`some_byte := 0x0A;`
	`Soft_UART_Write(some_byte);`

Soft_UART_Break

Prototype	`procedure Soft_UART_Break();`
Returns	Nothing.
Description	Soft_UART_Read is blocking routine and it can block the program flow. Call this routine from interrupt to unblock the program execution. This mechanism is similar to WDT. Note: Interrupts should be disabled before using Software UART routines again (see note at the top of this page).
Requires	Nothing.

Example

```
var data1, error, counter : byte ;

procedure Timer0Overflow_ISR(); org 0x12;
begin
  counter := 0;

  if (counter >= 20) then
  begin
    Soft_UART_Break();
    counter := 0;                  // reset counter
  end
  else
    Inc(counter);                  // increment counter
end;

begin
  TOIE0_bit  := 1;             // Timer0 overflow interrupt enable
  TCCR0_bit  := 5;            // Start timer with 1024 prescaler

  SREG_I_bit := 0;                      // Interrupt disable

  ...

  Soft_UART_Init(9600);
  Soft_UART_Write(0x55);

  ...

  // try Soft_UART_Read with blocking prevention mechanism
  SREG_I_bit := 1;                      // Interrupt enable
  data1 := Soft_UART_Read(&error);
  SREG_I_bit := 0;                      // Interrupt disable

  ...

end;
```

Library Example

This example demonstrates simple data exchange via software UART. If MCU is connected to the PC, you can test the example from the mikroPascal PRO for AVR USART Terminal Tool.

```
program Soft_UART;

// Soft UART connections
var Soft_UART_Rx_Pin : sbit at PIND.B0;
    Soft_UART_Tx_Pin : sbit at PORTD.B1;
    Soft_UART_Rx_Pin_Direction : sbit at DDRD.B0;
    Soft_UART_Tx_Pin_Direction : sbit at DDRD.B1;
// End Soft UART connections

var error : byte;
    counter, byte_read : byte;              // Auxiliary variables

begin

  DDRB := 0xFF;         // Set PORTB as output (error signalization)
  PORTB := 0;                                     // No error

 error := Soft_UART_Init(9600, 0);// Initialize Soft UART at 9600 bps
  if (error > 0) then
    begin
      PORTB := error;                          // Signalize Init error
      while (TRUE) do nop;                     // Stop program
    end;
  Delay_ms(100);

  for counter := 'z' downto 'A' do // Send bytes from 'z' downto 'A'
  begin
    Soft_UART_Write(counter);
    Delay_ms(100);
  end;

  while TRUE do                                   // Endless loop
    begin
       byte_read := Soft_UART_Read(error);   // Read byte, then test
error flag
      if (error <> 0) then                   // If error was detected
        PORTB := error                       //    signal it on PORTB
      else
       Soft_UART_Write(byte_read);        // If error was not detect-
ed, return byte read
    end;
end.
```

SOUND LIBRARY

The mikroPascal PRO for AVR provides a Sound Library to supply users with routines necessary for sound signalization in their applications. Sound generation needs additional hardware, such as piezo-speaker (example of piezo-speaker interface is given on the schematic at the bottom of this page).

External dependencies of Sound Library

The following variables must be defined in all projects using Sound Library:	Description:	Example :
`var Sound_Play_Pin : sbit; sfr; external;`	Sound output pin.	`var Sound_Play_Pin : sbit at PORTC.B3;`
`var Sound_Play_Pin_Direction : sbit; sfr; external;`	Direction of the Sound output pin.	`var Sound_Play_Pin_Direction : sbit at DDRC.B3;`

Library Routines

- Sound_Init
- Sound_Play

Sound_Init

Prototype	`procedure Sound_Init();`
Returns	Nothing.
Description	Configures the appropriate MCU pin for sound generation.
Requires	Global variables: - `Sound_Play_Pin`: Sound output pin - `Sound_Play_Pin_Direction`: Direction of the Sound output pin must be defined before using this function.

Example	`// Sound library connections`
	`var Sound_Play_Pin : sbit at PORTC.B3;`
	`var Sound_Play_Pin_Direction : sbit at DDRC.B3;`
	`// End of Sound library connections`
	
	`...`
	`Sound_Init();`

Sound_Play

Prototype	`procedure Sound_Play(freq_in_Hz: word; duration_ms: word);`
Returns	Nothing.
Description	Generates the square wave signal on the appropriate pin. Parameters : - `freq_in_Hz`: signal frequency in Hertz (Hz) - `duration_ms`: signal duration in miliseconds (ms)
Requires	In order 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	`// Play sound of 1KHz in duration of 100ms`
	`Sound_Play(1000, 100);`

Library Example

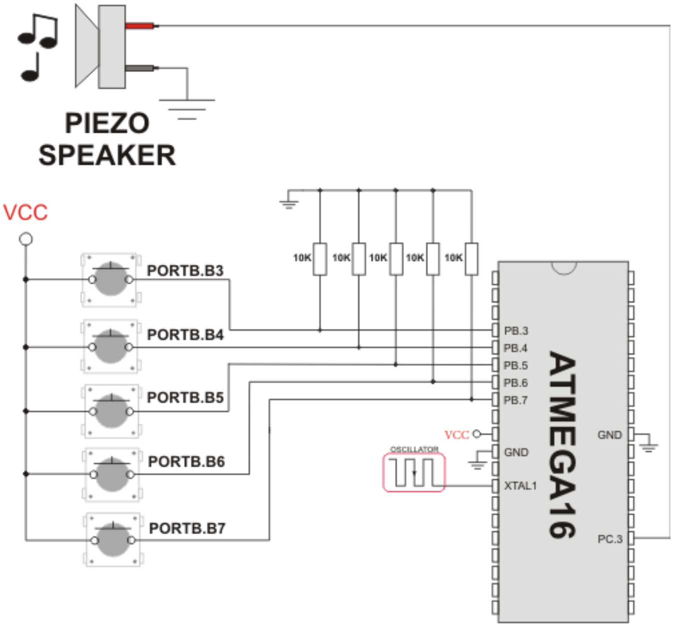
The example is a simple demonstration of how to use the Sound Library for playing tones on a piezo speaker.

```
program Sound;

// Sound connections
var Sound_Play_Pin : sbit at PORTC.B3;
var Sound_Play_Pin_direction : sbit at DDRC.B3;
// End Sound connections

procedure Tone1();
  begin
    Sound_Play(500, 200);                // Frequency = 500Hz, Duration = 200ms
  end;

procedure Tone2();
  begin
    Sound_Play(555, 200);                // Frequency = 555Hz, Duration = 200ms
  end;

procedure Tone3();
  begin
    Sound_Play(625, 200);                // Frequency = 625Hz, Duration = 200ms
  end;

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

```
    Tone1(); Tone2(); Tone3();
    Tone1(); Tone2(); Tone3(); Tone3();
    Tone1(); Tone2(); Tone3();
    Tone3(); Tone3(); Tone2(); Tone2(); Tone1();
  end;

procedure ToneA();                 // Tones used in Melody2 function
  begin
    Sound_Play(1250, 20);
  end;

procedure ToneC();
  begin
    Sound_Play(1450, 20);
  end;

procedure ToneE();
  begin
    Sound_Play(1650, 80);
  end;

procedure Melody2();                  // Plays Melody2
var counter : byte;
  begin
    for counter := 9 downto 1 do
      begin
        ToneA();
        ToneC();
        ToneE();
      end;
  end;

begin

  DDRB := 0x00;                 // Configure PORTB as input
  Delay_ms(2000);
  Sound_Init();                 // Initialize sound pin

  Sound_Play(2000, 1000);      // Play starting sound, 2kHz, 1 second

  while TRUE do                // endless loop
    begin
if (PINB.B7 <> 0) then         // If PORTB.7 is pressed play Tone1
      begin
        Tone1();
      while (PINB.B7 <> 0) do nop; // Wait for button to be released
      end;

if (PINB.B6 <> 0) then            // If PORTB.6 is pressed play Tone2
      begin
 Tone2();
```

```
                       while (PINB.B6 <> 0) do nop; // Wait for but-
ton to be released
       end;

      if (PINB.B5 <> 0) then      // If PORTB.5 is pressed play Tone3
       begin
         Tone3();
       while (PINB.B5 <> 0) do nop; // Wait for button to be released
       end;

      if (PINB.B4 <> 0) then     // If PORTB.4 is pressed play Melody2
        begin
         Melody2();
       while (PINB.B4 <> 0) do nop; // Wait for button to be released
       end;

      if (PINB.B3 <> 0) then     // If PORTB.3 is pressed play Melody
       begin
         Melody();
           while (PINB.B3 <> 0) do nop ; // Wait for button to be
released
       end;
    end;
end.
```

HW Connection



Example of Sound Library sonnection

SPI LIBRARY

mikroPascal PRO for AVR provides a library for comfortable with SPI work in Master mode. The AVR MCU can easily communicate with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

Note: Some AVR MCU's have alternative SPI ports, which SPI signals can be redirected to by setting or clearing SPIPS (SPI Pin Select) bit of the MCUCR register. Please consult the appropriate datasheet.

Library Routines

- SPI1_Init
- SPI1_Init_Advanced
- SPI1_Read
- SPI1_Write

SPI1_Init

Prototype	`procedure SPI1_Init();`
Returns	Nothing.
Description	This routine configures and enables SPI module with the following settings: - master mode - 8 bit data transfer - most significant bit sent first - serial clock low when idle - data sampled on leading edge - serial clock = fosc/4
Requires	MCU must have SPI module.

Example	`// Initialize the SPI1 module with default settings`
	`SPI1_Init();`

SPI1_Init_Advanced

<table>
<tr><td><b>Prototype</b></td><td><code>procedure SPI1_Init_Advanced(mode : byte; fcy_div : byte; and_edge : byte)</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>Configures and initializes SPI. SPI1_Init_Advanced or SPI1_Init needs to be called before using other functions of SPI Library.

Parameters <code>mode, fcy_div</code> and <code>clock_and_edge</code> determine the work mode for SPI, and can have the following values:

<table>
<tr><th>Mask</th><th>Description</th><th>Predefined library const</th></tr>
<tr><td colspan="3"><b>SPI mode constants:</b></td></tr>
<tr><td>0x10</td><td>Master mode</td><td>_SPI_MASTER</td></tr>
<tr><td>0x00</td><td>Slave mode</td><td>_SPI_SLAVE</td></tr>
<tr><td colspan="3"><b>Clock rate select constants:</b></td></tr>
<tr><td>0x00</td><td>Sck = Fosc/4, Master mode</td><td>_SPI_FCY_DIV4</td></tr>
<tr><td>0x01</td><td>Sck = Fosc/16, Master mode</td><td>_SPI_FCY_DIV16</td></tr>
<tr><td>0x02</td><td>Sck = Fosc/64, Master mode</td><td>_SPI_FCY_DIV64</td></tr>
<tr><td>0x03</td><td>Sck = Fosc/128, Master mode</td><td>_SPI_FCY_DIV128</td></tr>
<tr><td>0x04</td><td>Sck = Fosc/2, Master mode</td><td>_SPI_FCY_DIV2</td></tr>
<tr><td>0x05</td><td>Sck = Fosc/8, Master mode</td><td>_SPI_FCY_DIV8</td></tr>
<tr><td>0x06</td><td>Sck = Fosc/32, Master mode</td><td>_SPI_FCY_DIV32</td></tr>
<tr><td colspan="3"><b>SPI clock polarity and phase constants:</b></td></tr>
<tr><td>0x00</td><td>Clock idle level is low, sample on rising edge</td><td>_SPI_CLK_LO_LEADING</td></tr>
<tr><td>0x04</td><td>Clock idle level is low, sample on falling edge</td><td>_SPI_CLK_LO_TRAILING</td></tr>
<tr><td>0x08</td><td>Clock idle level is high, sample on rising edge</td><td>_SPI_CLK_HI_LEADING</td></tr>
<tr><td>0x0C</td><td>Clock idle level is high, sample on falling edge</td><td>_SPI_CLK_HI_TRAILING</td></tr>
</table>

Note: Some SPI clock speeds are not supported by all AVR MCUs and these are: Fosc/2, Fosc/8, Fosc/32. Please consult appropriate datasheet.</td></tr>
</table>

Requires	MCU must have SPI module.

Example	`// Set SPI to the Master Mode, clock = Fosc/32 , clock idle level is high, data sampled on falling edge:`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV32, _SPI_CLK_HI_TRAILING);`

SPI1_Read

Prototype	`function SPI1_Read(buffer: byte): byte;`
Returns	Received data.
Description	Reads one byte from the SPI bus. Parameters : - `buffer`: dummy data for clock generation (see device Datasheet for SPI mod ules implementation details)
Requires	SPI module must be initialized before using this function. See SPI1_Init and SPI1_Init_Advanced routines.

Example	`// read a byte from the SPI bus`
	`var take, dummy1 : byte ;`
	`...`
	`take := SPI1_Read(dummy1);`

SPI1_Write

Prototype	`procedure SPI1_Write(wrdata: byte);`
Returns	Nothing.
Description	Writes byte via the SPI bus. Parameters : - `wrdata`: data to be sent
Requires	SPI module must be initialized before using this function. See SPI1_Init and SPI1_Init_Advanced routines.

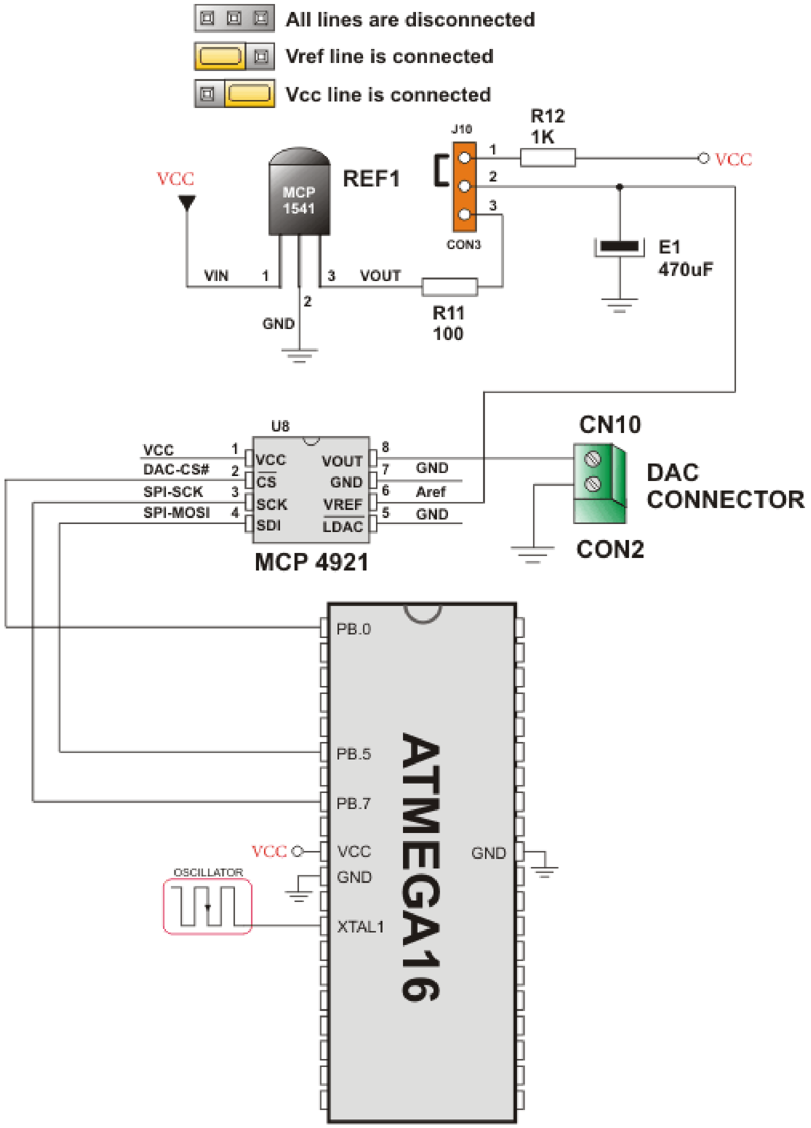
Example	`// write a byte to the SPI bus`
	`var buffer : byte;`
	`...`
	`SPI1_Write(buffer);`

Library Example

The code demonstrates how to use SPI library functions for communication between SPI module of the MCU and Microchip's MCP4921 12-bit D/A converter

```
program SPI;

// DAC module connections
var Chip_Select : sbit at PORTB.B0;
    Chip_Select_Direction : sbit at DDRB.B0;
// End DAC module connections

var value : word;

procedure InitMain();
  begin
    DDA0_bit := 0;                      // Set PA0 pin as input
    DDA1_bit := 0;                      // Set PA1 pin as input
    Chip_Select := 1;                   // Deselect DAC
    Chip_Select_Direction := 1;         // Set CS# pin as Output
    SPI1_Init();                        // Initialize SPI1 module
  end;

// DAC increments (0..4095) --> output voltage (0..Vref)
procedure DAC_Output( valueDAC : word);
var temp : byte;
  begin
    Chip_Select := 0;                           // Select DAC chip

    // Send High Byte
    temp := word(valueDAC shr 8) and 0x0F;  // Store valueDAC[ 11..8]
to temp[ 3..0]
     temp := temp or 0x30;                // Define DAC setting, see
MCP4921 datasheet
    SPI1_Write(temp);                    // Send high byte via SPI

    // Send Low Byte
   temp := valueDAC;           // Store valueDAC[ 7..0] to temp[ 7..0]
    SPI1_Write(temp);          // Send low byte via SPI

    Chip_Select := 1;                          // Deselect DAC chip
  end;

begin

 InitMain();                       // Perform main initialization

  value := 2048;                   // When program starts, DAC gives
  //    the output in the mid-range

  while ( TRUE ) do                          // Endless loop
    begin

       if ((PINA.B0) and (value < 4095)) then   // If PA0 button is
pressed
         Inc(value)                              //    increment value
       else
         begin
           if ((PINA.B1) and (value > 0)) then     // If PA1 button
is pressed
             Dec(value);                  //    decrement value
         end;

       DAC_Output(value);                 // Send value to DAC chip
      Delay_ms(1);                        // Slow down key repeat pace
    end;
end.
```

HW Connection



SPI HW connection

SPI ETHERNET LIBRARY

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

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

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

SPI Ethernet library supports:

- IPv4 protocol.
- ARP requests.
- ICMP echo requests.
- UDP requests.
- TCP requests (no stack, no packet reconstruction).
- packet fragmentation is **NOT** supported.

Note: Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

Note: The appropriate hardware SPI module must be initialized before using any of the SPI Ethernet library routines. Refer to SPI Library.

External dependencies of SPI Ethernet Library

The following variables must be defined in all projects using SPI Ethernet Library:	Description:	Example :
`var SPI_Ethernet_CS : sbit; sfr; external;`	ENC28J60 chip select pin.	`var SPI_Ethernet_CS : sbit at PORTB.B4;`
`var SPI_Ethernet_RST : sbit; sfr; external;`	ENC28J60 reset pin.	`var SPI_Ethernet_RST : sbit at PORTB.B5;`
`var SPI_Ethernet_CS_Direction : sbit; sfr; external;`	Direction of the ENC28J60 chip select pin.	`var SPI_Ethernet_CS_Direction : sbit at DDRB.B4;`
`var SPI_Ethernet_RST_Direction : sbit; sfr; external;`	Direction of the ENC28J60 reset pin.	`var SPI_Ethernet_RST_Direction : sbit at DDRB.B5;`

The following routines must be defined in all project using SPI Ethernet Library:	Description:	Examples :

`function Spi_Ethernet_UserTCP(remoteHost : ^byte,`
`remotePort : word,`
`localPort : word,`
`reqLength : word): word;`
	TCP request handler.	Refer to the library example at the bottom of this page for code implementation.

`function Spi_Ethernet_UserUDP(remoteHost : ^byte,`
`remotePort : word,`
`destPort : word,`
`reqLength : word): word;`
	UDP request handler.	Refer to the library example at the bottom of this page for code implementation.

Library Routines

- Spi_Ethernet_Init
- Spi_Ethernet_Enable
- Spi_Ethernet_Disable
- Spi_Ethernet_doPacket
- Spi_Ethernet_putByte
- Spi_Ethernet_putBytes
- Spi_Ethernet_putString
- Spi_Ethernet_putConstString
- Spi_Ethernet_putConstBytes
- Spi_Ethernet_getByte
- Spi_Ethernet_getBytes
- Spi_Ethernet_UserTCP
- Spi_Ethernet_UserUDP

Spi_Ethernet_Init

Prototype	`procedure Spi_Ethernet_Init(mac: ^byte; ip: ^byte; fullDuplex:byte);`
Returns	Nothing.
Description	This is MAC module routine. It initializes ENC28J60 controller. This function is internaly splited into 2 parts to help linker when coming short of memory. ENC28J60 controller settings (parameters not mentioned here are set to default): - receive buffer start address : 0x0000. - receive buffer end address : 0x19AD. - transmit buffer start address: 0x19AE. - transmit buffer end address : 0x1FFF. - RAM buffer read/write pointers in auto-increment mode. - receive filters set to default: CRC + MAC Unicast + MAC Broadcast in OR mode. - flow control with TX and RX pause frames in full duplex mode. - frames are padded to 60 bytes + CRC. - maximum packet size is set to 1518. - Back-to-Back Inter-Packet Gap: 0x15 in full duplex mode; 0x12 in half duplex mode. - Non-Back-to-Back Inter-Packet Gap: 0x0012 in full duplex mode; 0x0C12 in half duplex mode. - Collision window is set to 63 in half duplex mode to accomodate some - ENC28J60 revisions silicon bugs. - CLKOUT output is disabled to reduce EMI generation. - half duplex loopback disabled. - LED configuration: default (LEDA-link status, LEDB-link activity).

Description	Parameters: - mac: RAM buffer containing valid MAC address. - ip: RAM buffer containing valid IP address. - fullDuplex: ethernet duplex mode switch. Valid values: 0 (half duplex mode) and 1 (full duplex mode).
Requires	The appropriate hardware SPI module must be previously initialized.

Example	`const Spi_Ethernet_HALFDUPLEX = 0;`
	`const Spi_Ethernet_FULLDUPLEX = 1;`
	
	`var`
	`  myMacAddr : array[ 6]  of byte; // my MAC address`
	`  myIpAddr  : array[ 4]  of byte; // my IP addr`
	`  ...`
	`  myMacAddr[ 0]  := 0x00;`
	`  myMacAddr[ 1]  := 0x14;`
	`  myMacAddr[ 2]  := 0xA5;`
	`  myMacAddr[ 3]  := 0x76;`
	`  myMacAddr[ 4]  := 0x19;`
	`  myMacAddr[ 5]  := 0x3F;`
	
	`  myIpAddr[ 0]   := 192;`
	`  myIpAddr[ 1]   := 168;`
	`  myIpAddr[ 2]   := 1;`
	`  myIpAddr[ 3]   := 60;`
	
	`  Spi_Init();`
	`  Spi_Ethernet_Init(PORTC, 0, PORTC, 1, myMacAddr, myIpAddr,`
	`Spi_Ethernet_FULLDUPLEX);`

Spi_Ethernet_Enable

Prototype	`procedure Spi_Ethernet_Enable(enFlt: byte);`
Returns	Nothing.
Description	This is MAC module routine. This routine enables appropriate network traffic on the ENC28J60 module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be enabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be enabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.

<table>
<tr><td><b>Description</b></td><td>Parameters:

- enFlt: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter:

<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th><th>Predefined library const</th></tr>
<tr><td>0</td><td>0x01</td><td>MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be enabled.</td><td>Spi_Ethernet_BROADCAST</td></tr>
<tr><td>1</td><td>0x02</td><td>MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be enabled.</td><td>Spi_Ethernet_MULTICAST</td></tr>
<tr><td>2</td><td>0x04</td><td>not used</td><td>none</td></tr>
<tr><td>3</td><td>0x08</td><td>not used</td><td>none</td></tr>
<tr><td>4</td><td>0x10</td><td>not used</td><td>none</td></tr>
<tr><td>5</td><td>0x20</td><td>CRC check flag. When set, packets with invalid CRC field will be discarded.</td><td>Spi_Ethernet_CRC</td></tr>
<tr><td>6</td><td>0x40</td><td>not used</td><td>none</td></tr>
<tr><td>7</td><td>0x80</td><td>MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be enabled.</td><td>Spi_Ethernet_UNICAST</td></tr>
</table>

<b>Note:</b> Advance filtering available in the ENC28J60 module such as Pattern Match, Magic Packet and Hash Table can not be enabled by this routine. Additionaly, all filters, except CRC, enabled with this routine will work in OR mode, which means that packet will be received if any of the enabled filters accepts it.

<b>Note</b>: This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the ENC28J60 module. The ENC28J60 module should be properly cofigured by the means of Spi_Ethernet_Init routine.</td></tr>
<tr><td><b>Requires</b></td><td>Ethernet module has to be initialized. See Spi_Ethernet_Init.</td></tr>
<tr><td><b>Example</b></td><td><code>Spi_Ethernet_Enable(Spi_Ethernet_CRC or Spi_Ethernet_UNICAST); // enable CRC checking and Unicast traffic</code></td></tr>
</table>

Spi_Ethernet_Disable

<table>
<tr><td><b>Prototype</b></td><td><code><b>procedure</b> Spi_Ethernet_Disable(disFlt: byte);</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>
<p>This is MAC module routine. This routine disables appropriate network traffic on the <code>ENC28J60</code> module by the means of it's receive filters (unicast, multicast, broadcast, crc). Specific type of network traffic will be disabled if a corresponding bit of this routine's input parameter is set. Therefore, more than one type of network traffic can be disabled at the same time. For this purpose, predefined library constants (see the table below) can be ORed to form appropriate input value.</p>
<p>Parameters:</p>
<p>- <code>disFlt</code>: network traffic/receive filter flags. Each bit corresponds to the appropriate network traffic/receive filter:</p>
<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th><th>Predefined library const</th></tr>
<tr><td><code>0</code></td><td><code>0x01</code></td><td>MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be disabled.</td><td><code>Spi_Ethernet_ BROADCAST</code></td></tr>
<tr><td><code>1</code></td><td><code>0x02</code></td><td>MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be disabled.</td><td><code>Spi_Ethernet_ MULTICAST</code></td></tr>
<tr><td><code>2</code></td><td><code>0x04</code></td><td>not used</td><td><code>none</code></td></tr>
<tr><td><code>3</code></td><td><code>0x08</code></td><td>not used</td><td><code>none</code></td></tr>
<tr><td><code>4</code></td><td><code>0x10</code></td><td>not used</td><td><code>none</code></td></tr>
<tr><td><code>5</code></td><td><code>0x20</code></td><td>CRC check flag. When set, CRC check will be disabled and packets with invalid CRC field will be accepted.</td><td><code>Spi_Ethernet_ CRC</code></td></tr>
<tr><td><code>6</code></td><td><code>0x40</code></td><td>not used</td><td><code>none</code></td></tr>
<tr><td><code>7</code></td><td><code>0x80</code></td><td>MAC Unicast traffic/receive filter flag. When set, MAC unicast traffic will be disabled.</td><td><code>Spi_Ethernet_ UNICAST</code></td></tr>
</table>
<p><b>Note:</b> Advance filtering available in the ENC28J60 module such as Pattern Match, Magic Packet and Hash Table can not be disabled by this routine.</p>
<p><b>Note:</b> This routine will change receive filter configuration on-the-fly. It will not, in any way, mess with enabling/disabling receive/transmit logic or any other part of the <code>ENC28J60</code> module. The ENC28J60 module should be properly cofigured by the means of Spi_Ethernet_Init routine.</p>
</td></tr>
</table>

Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	`Spi_Ethernet_Disable(Spi_Ethernet_CRC or Spi_Ethernet_UNICAST);`
	`// disable CRC checking and Unicast traffic`

Spi_Ethernet_doPacket

Prototype	`function Spi_Ethernet_doPacket(): byte;`
Returns	- `0` - upon successful packet processing (zero packets received or received packet processed successfully). - `1` - upon reception error or receive buffer corruption. `ENC28J60` controller needs to be restarted. - `2` - received packet was not sent to us (not our IP, nor IP broadcast address). - `3` - received IP packet was not IPv4. - `4` - received packet was of type unknown to the library.
Description	This is MAC module routine. It processes next received packet if such exists. Packets are processed in the following manner: ARP & ICMP requests are replied automatically. upon TCP request the Spi_Ethernet_UserTCP function is called for further processing. upon UDP request the Spi_Ethernet_UserUDP function is called for further processing. **Note**: Spi_Ethernet_doPacket must be called as often as possible in user's code.
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	`while TRUE do`
	`  begin`
	`    Spi_Ethernet_doPacket(); // process received packets`
	`  end`

Spi_Ethernet_putByte

Prototype	**procedure** Spi_Ethernet_putByte(v: byte);
Returns	Nothing.
Description	This is MAC module routine. It stores one byte to address pointed by the current ENC28J60 write pointer (EWRPT). Parameters: - v: value to store
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.
Example	**var** data as byte; ... Spi_Ethernet_putByte(data); // put an byte into ENC28J60 buffer

Spi_Ethernet_putBytes

Prototype	**procedure** Spi_Ethernet_putBytes(ptr : ^byte; n : byte);
Returns	Nothing.
Description	This is MAC module routine. It stores requested number of bytes into ENC28J60 RAM starting from current ENC28J60 write pointer (EWRPT) location. Parameters: - ptr: RAM buffer containing bytes to be written into ENC28J60 RAM. - n: number of bytes to be written.
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	**var**
	  buffer : **array**[17] **of** byte;
	  ...
	  buffer := 'mikroElektronika';
	  ...
	  Spi_Ethernet_putBytes(buffer, 16); // put an RAM array into ENC28J60 buffer

Spi_Ethernet_putConstBytes

Prototype	`procedure Spi_Ethernet_putConstBytes(const ptr : ^byte; n : byte);`
Returns	Nothing.
Description	This is MAC module routine. It stores requested number of const bytes into `ENC28J60` RAM starting from current ENC28J60 write pointer (EWRPT) location. Parameters: - `ptr`: const buffer containing bytes to be written into ENC28J60 RAM. - `n`: number of bytes to be written.
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	`const`
	`  buffer : array[17] of byte;`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  Spi_Ethernet_putConstBytes(buffer, 16); // put a const array into ENC28J60 buffer`

Spi_Ethernet_putString

Prototype	`function Spi_Ethernet_putString(^ptr : byte) : word;`
Returns	Number of bytes written into ENC28J60 RAM.
Description	This is MAC module routine. It stores whole string (excluding null termination) into `ENC28J60` RAM starting from current `ENC28J60` write pointer (EWRPT) location. Parameters: - `ptr`: string to be written into `ENC28J60` RAM.
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	`var`
	`  buffer : string[16];`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  Spi_Ethernet_putString(buffer); // put a RAM string into ENC28J60 buffer`

Spi_Ethernet_putConstString

Prototype	`function Spi_Ethernet_putConstString(const ptr : ^byte): word;`
Returns	Number of bytes written into `ENC28J60` RAM.
Description	This is MAC module routine. It stores whole const string (excluding null termination) into `ENC28J60` RAM starting from current `ENC28J60` write pointer (EWRPT) location. Parameters: - `ptr`: const string to be written into `ENC28J60` RAM.
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	`const`
	`  buffer : string[16];`
	`  ...`
	`  buffer := 'mikroElektronika';`
	`  ...`
	`  Spi_Ethernet_putConstString(buffer); // put a const string into`
	`ENC28J60 buffer`

Spi_Ethernet_getByte

Prototype	`function Spi_Ethernet_getByte(): byte;`
Returns	Byte read from `ENC28J60` RAM.
Description	This is MAC module routine. It fetches a byte from address pointed to by current `ENC28J60` read pointer (`ERDPT`).
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	`var buffer : byte;`
	`...`
	`buffer := Spi_Ethernet_getByte(); // read a byte from ENC28J60`
	`buffer`

Spi_Ethernet_getBytes

Prototype	`procedure Spi_Ethernet_getBytes(ptr : ^byte; addr : word; n : byte);`
Returns	Nothing.
Description	This is MAC module routine. It fetches equested number of bytes from `ENC28J60` RAM starting from given address. If value of `0xFFFF` is passed as the address parameter, the reading will start from current `ENC28J60` read pointer (`ERDPT`) location. Parameters: - `ptr`: buffer for storing bytes read from ENC28J60 RAM. - `addr`: ENC28J60 RAM start address. Valid values: 0..8192. - `n`: number of bytes to be read.
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.

Example	`var`
	`  buffer : array[ 16] of byte;`
	`  ...`
	`  Spi_Ethernet_getBytes(buffer, 0x100, 16); // read 16 bytes, starting from address 0x100`

Spi_Ethernet_UserTCP

Prototype	`function Spi_Ethernet_UserTCP(remoteHost : ^byte; remotePort : word; localPort : word; reqLength : word) : word;`
Returns	- `0` - there should not be a reply to the request. - Length of TCP/HTTP reply data field - otherwise.
Description	This is TCP module routine. It is internally called by the library. The user accesses to the TCP/HTTP request by using some of the Spi_Ethernet_get routines. The user puts data in the transmit buffer by using some of the Spi_Ethernet_put routines. The function must return the length in bytes of the TCP/HTTP reply, or 0 if there is nothing to transmit. If there is no need to reply to the TCP/HTTP requests, just define this function with return(0) as a single statement. Parameters: - `remoteHost` : client's IP address. - `remotePort` : client's TCP port. - `localPort` : port to which the request is sent. - `reqLength` : TCP/HTTP request data field length. **Note:** The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.

Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.

Spi_Ethernet_UserUDP

Prototype	`function Spi_Ethernet_UserUDP(remoteHost : ^byte; remotePort : word; destPort : word; reqLength : word) : word;`
Returns	- `0` - there should not be a reply to the request. - Length of UDP reply data field - otherwise.
Description	This is UDP module routine. It is internally called by the library. The user accesses to the UDP request by using some of the Spi_Ethernet_get routines. The user puts data in the transmit buffer by using some of the Spi_Ethernet_put routines. 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 the UDP requests, just define this function with a return(0) as single statement. Parameters: - `remoteHost` : client's IP address. - `remotePort` : client's port. - `destPort` : port to which the request is sent. - `reqLength` : UDP request data field length. Note: The function source code is provided with appropriate example projects. The code should be adjusted by the user to achieve desired reply.
Requires	Ethernet module has to be initialized. See Spi_Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.

Library Example

This code shows how to use the AVR mini Ethernet library :

- the board will reply to ARP & ICMP echo requests
- the board will reply to UDP requests on any port :
  returns the request in upper char with a header made of remote host IP & port number
- the board will reply to HTTP requests on port 80, GET method with pathnames :
  / will return the HTML main page
  /s will return board status as text string
  /t0 ... /t7 will toggle P3.b0 to P3.b7 bit and return HTML main page
  all other requests return also HTML main page.

Main program code :

```
program enc_ethernet;

uses eth_enc28j60_utils ; //this is where you should write implemen-
tation for UDP and HTTP
{*********************************
 * RAM variables
 *}
var myMacAddr    : array[ 6]  of byte   ; // my MAC address
    myIpAddr     : array[ 4]  of byte   ; // my IP address
    gwIpAddr     : array[ 4]  of byte   ; // gateway (router) IP address
    ipMask       : array[ 4]  of byte   ; // network mask (for example
: 255.255.255.0)
    dnsIpAddr    : array[ 4]  of byte   ; // DNS server IP address

// mE ehternet NIC pinout
  SPI_Ethernet_Rst : sbit at PORTB.B4;
  SPI_Ethernet_CS  : sbit at PORTB.B5;
  SPI_Ethernet_Rst_Direction : sbit at DDRB.B4;
  SPI_Ethernet_CS_Direction  : sbit at DDRB.B5;
// end ethernet NIC definitions

begin
  // set PORTC as input
  DDRC := 0;
  // set PORTD as output
  DDRD := 0xFF;

  httpCounter := 0;

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

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

  gwIpAddr[ 0]  := 192;
  gwIpAddr[ 1]  := 168;
  gwIpAddr[ 2]  := 20;
  gwIpAddr[ 3]  := 6;

  ipMask[ 0]  := 255;
  ipMask[ 1]  := 255;
  ipMask[ 2]  := 255;
  ipMask[ 3]  := 0;

  dnsIpAddr[ 0]  := 192;
  dnsIpAddr[ 1]  := 168;
  dnsIpAddr[ 2]  := 20;
  dnsIpAddr[ 3]  := 1;

  {*
   * starts ENC28J60 with :
   * reset bit on PORTB.B4
   * CS bit on PORTB.B5
   * my MAC & IP address
   * full duplex
   *}

  SPI1_Init_Advanced(_SPI_MASTER,  _SPI_FCY_DIV2,  _SPI_CLK_LO_LEAD-
ING);
  SPI_Rd_Ptr := @SPI1_Read;
  SPI_Ethernet_UserTCP_Ptr := @SPI_Ethernet_UserTCP;
  SPI_Ethernet_UserUDP_Ptr := @SPI_Ethernet_UserUDP;
  SPI_Ethernet_Init(myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX) ;

  // dhcp will not be used here, so use preconfigured addresses
  SPI_Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr) ;

  while true do                   // do forever
    begin
    SPI_Ethernet_doPacket() ;    // process incoming Ethernet packets

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

Unit eth_enc28j60_utils code :

```
unit eth_enc28j60_utils;

{***********************************************************
 * ROM constant strings
 *}

const httpHeader : string[ 30]  = 'HTTP/1.1 200 OK'+#10+'Content-type:
' ;   // HTTP header
```

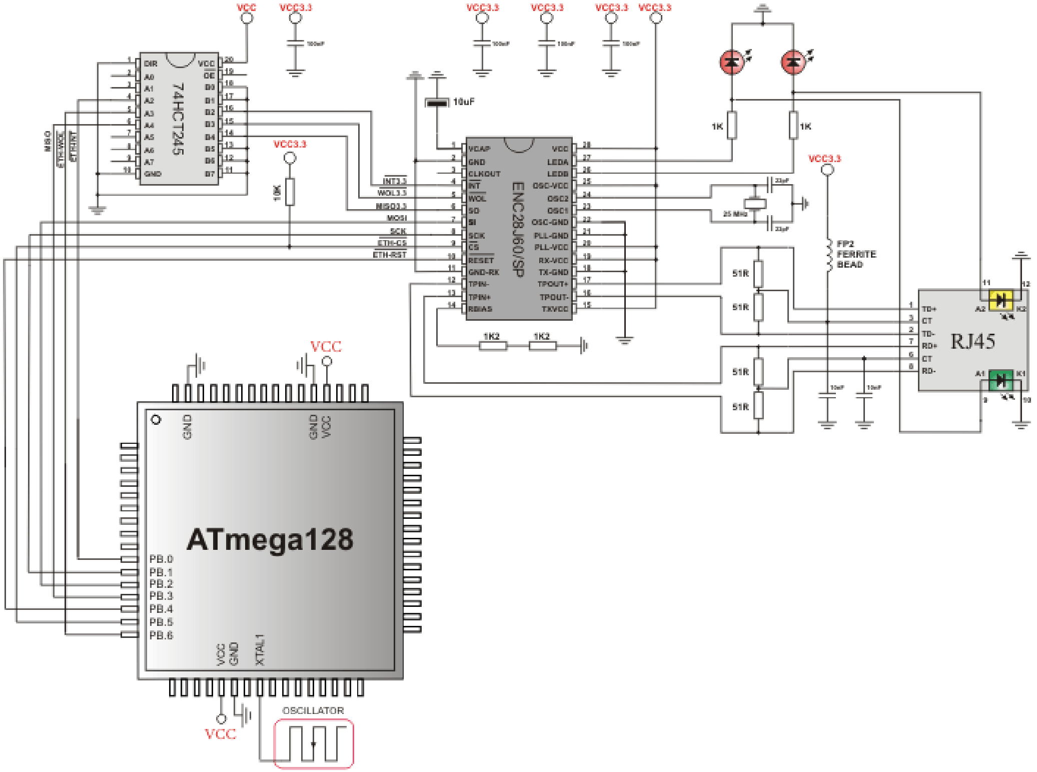
```
const  httpMimeTypeHTML  :  string[13]      =  'text/html'+#10+#10  ;
// HTML MIME type
const  httpMimeTypeScript  :  string[14]  =  'text/plain'+#10+#10  ;
// TEXT MIME type
const httpMethod : string[5] = 'GET /';
{*
 * web page, splited into 2 parts :
 * when coming short of ROM, fragmented data is handled more effi-
ciently by linker
 *
 * this HTML page calls the boards to get its status, and builds
itself with javascript
 *}
const indexPage : string[513] =
                                        '<meta  http-equiv="refresh"
content="3;url=http://192.168.20.60">'+
                      '<HTML><HEAD></HEAD><BODY>'+
                      '<h1>AVR + ENC28J60 Mini Web Server</h1>'+
                      '<a href=/>Reload</a>'+
                      '<script src=/s></script>'+
                          '<table><tr><td valign=top><table border=1
style="font-size:20px ;font-family: terminal ;">'+
                      '<tr><th colspan=2>PINC</th></tr>'+
                      '<script>'+
                      'var str,i;'+
                      'str="";'+
                      'for(i=0;i<8;i++)'+
                 '{ str+="<tr><td bgcolor=pink>BUTTON #"+i+"</td>";'+
                      'if(PINC&(1<<i)){ str+="<td bgcolor=red>ON";}'+
                      'else { str+="<td bgcolor=#cccccc>OFF";}'+
                      'str+="</td></tr>";}'+
                      'document.write(str) ;'+
                      '</script>';

const indexPage2 : string[466] =
                      '</table></td><td>'+
                       '<table border=1 style="font-size:20px ;font-
family: terminal ;">'+
                      '<tr><th colspan=3>PORTD</th></tr>'+
                      '<script>'+
                      'var str,i;'+
                      'str="";'+
                      'for(i=0;i<8;i++)'+
                  '{ str+="<tr><td bgcolor=yellow>LED #"+i+"</td>";'+
                     'if(PORTD&(1<<i)){ str+="<td bgcolor=red>ON";}'+
                      'else { str+="<td bgcolor=#cccccc>OFF";}'+
'str+="</td><td><a href=/t"+i+">Toggle</a></td></tr>";}'+
                      'document.write(str) ;'+
                      '</script>'+
                      '</table></td></tr></table>'+
            'This            is           HTTP          request
#<script>document.write(REQ)</script></BODY></HTML>';

var    getRequest  : array[15] of byte;  // HTTP request buffer
         dyna          : array[31] of byte;   // buffer for dynamic
response
       httpCounter : word ;               // counter of HTTP requests

function  SPI_Ethernet_UserTCP(var  remoteHost  :  array[4]  of  byte;
remotePort, localPort, reqLength : word) : word;
function  SPI_Ethernet_UserUDP(var  remoteHost  :  array[4]  of  byte;
remotePort, destPort, reqLength : word) : word;

implementation
{********************************************
 * user defined functions
 *}

{*
 * put the constant string pointed to by s to the ENC transmit buffer
 *}
function    putConstString(const s: ^byte) : word;
  begin
    result := 0;
    while(s^ <> 0) do
      begin
        SPI_Ethernet_putByte(s^) ;
        s := s + 1;
        result := result + 1 ;
      end;
  end;

{*
 * put the string pointed to by s to the ENC transmit buffer
 *}
function putString(var s : array[100] of byte) : word;
 begin
    result := 0 ;
    while(s[result] <> 0) do
      begin
        SPI_Ethernet_putByte(s[result]) ;
        result := result + 1 ;
      end;
  end;
{*
 * this function is called by the library
  *  the  user  accesses  to  the  HTTP  request  by  successive  calls  to
SPI_Ethernet_getByte()
```

```
 * the user puts data in the transmit buffer by successive calls to
SPI_Ethernet_putByte()
 * the function must return the length in bytes of the HTTP reply,
or 0 if nothing to transmit
 *
 * if you don't need to reply to HTTP requests,
 * just define this function with a return(0) as single statement
 *
 *}
function SPI_Ethernet_UserTCP(var remoteHost : array[4] of byte;
                           remotePort, localPort, reqLength : word)
: word;
  var  len : word ;             // my reply length
       bitMask : byte ;         // for bit mask
       tmp: array[5] of byte; // to copy const array to ram for mem-
cmp

  begin
    len := 0;
 if(localPort <> 80) then // I listen only to web request on port 80
      begin
        result := 0;
        exit;
      end;

    // get 10 first bytes only of the request, the rest does not mat-
ter here
    for len := 0 to 9 do
      begin
        getRequest[len] := SPI_Ethernet_getByte() ;
      end;
    getRequest[len] := 0 ;
    len := 0;

    while (httpMethod[len] <> 0) do
      begin
        tmp[len] := httpMethod[len];
        Inc(len);
      end;
    len := 0;

    if(memcmp(@getRequest, @tmp, 5) <> 0) then  // only GET method
is supported here
      begin
        result := 0 ;
        exit;
      end;

httpCounter := httpCounter + 1 ;   // one more request done
```

```
    if(getRequest[5]  = 's') then                       // if request path
name starts with s, store dynamic data in transmit buffer
      begin
        // the text string replied by this request can be interpret-
ed as javascript statements
         // by browsers
         len := putConstString(@httpHeader) ;          // HTTP header
         len := len + putConstString(@httpMimeTypeScript) ;   // with
text MIME type

         // add PORTC value (buttons) to reply
         len := len + putString('var PINC=  ') ;
         WordToStr(PINC, dyna) ;
         len := len + putString(dyna) ;
         len := len + putString(';') ;

         // add PORTD value (LEDs) to reply
         len := len + putString('var PORTD= ') ;
         WordToStr(PORTD, dyna) ;
         len := len + putString(dyna) ;
         len := len + putString(';') ;

         // add HTTP requests counter to reply
         WordToStr(httpCounter, dyna) ;
         len := len + putString('var REQ=   ') ;
         len := len + putString(dyna) ;
         len := len + putString(';') ;
      end
    else
       if(getRequest[5]  = 't') then                                // if
request path name starts with t, toggle PORTD (LED) bit number that
comes after
         begin
           bitMask := 0;
           if(isdigit(getRequest[6]) <> 0) then               // if 0
<= bit number <= 9, bits 8 & 9 does not exist but does not matter
             begin
               bitMask := getRequest[6] - '0' ;               // con-
vert ASCII to integer
               bitMask := 1 shl bitMask ;          // create bit mask
               PORTD   := PORTD xor bitMask ;                 // tog-
gle PORTD with xor operator
             end;
         end;

    if(len = 0) then                             // what do to by default
begin
  len := putConstString(@httpHeader) ;                 // HTTP header
```

```
          len := len + putConstString(@httpMimeTypeHTML) ;   // with
HTML MIME type
         len := len + putConstString(@indexPage) ;           // HTML
page first part
         len := len + putConstString(@indexPage2) ;          // HTML
page second part
      end;
    result := len ;                        // return to the library with
the number of bytes to transmit
end;
{*
 * this function is called by the library
 * the user accesses to the UDP request by successive calls to
SPI_Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to
SPI_Ethernet_putByte()
 * the function must return the length in bytes of the UDP reply, or
0 if nothing to transmit
 *
 * if you don't need to reply to UDP requests,
 * just define this function with a return(0) as single statement
 *
 *}
function SPI_Ethernet_UserUDP(var remoteHost : array[ 4]  of byte;
                             remotePort, destPort, reqLength : word)
: word;
  var len : word;                          // my reply length
      ptr : ^byte;                       // pointer to the dynamic buffer
      tmp : string[ 5] ;
  begin
    // reply is made of the remote host IP address in human readable
format
    byteToStr(remoteHost[ 0] , dyna) ;   // first IP address byte
    dyna[ 3]  := '.' ;

    byteToStr(remoteHost[ 1] , tmp) ;    // second
    dyna[ 4]  := tmp[ 0] ;
    dyna[ 5]  := tmp[ 1] ;
    dyna[ 6]  := tmp[ 2] ;
    dyna[ 7]  := '.' ;

    byteToStr(remoteHost[ 2] , tmp) ;    // second
    dyna[ 8]  := tmp[ 0] ;
    dyna[ 9]  := tmp[ 1] ;
    dyna[ 10]  := tmp[ 2] ;
    dyna[ 11]  := '.' ;

    byteToStr(remoteHost[ 3] , tmp) ;    // second
    dyna[12]  := tmp[0] ;
    dyna[13]  := tmp[1] ;
    dyna[14]  := tmp[2] ;

    dyna[15]  := ':' ;                  // add separator

    // then remote host port number
    WordToStr(remotePort, tmp) ;
    dyna[16]  := tmp[0] ;
    dyna[17]  := tmp[1] ;
    dyna[18]  := tmp[2] ;
    dyna[19]  := tmp[3] ;
    dyna[20]  := tmp[4] ;
    dyna[21]  := ' ';
    dyna[22]  := '[' ;

    WordToStr(destPort, tmp) ;
    dyna[23]  := tmp[0] ;
    dyna[24]  := tmp[1] ;
    dyna[25]  := tmp[2] ;
    dyna[26]  := tmp[3] ;
    dyna[27]  := tmp[4] ;
    dyna[28]  := ']' ;
    dyna[29]  := ' ';
    dyna[30]  := 0 ;

    // the total length of the request is the length of the dynamic
string plus the text of the request
    len := 30 + reqLength ;

    // puts the dynamic string into the transmit buffer
    ptr := @dyna ;
    while(ptr^ <> 0) do
      begin
        SPI_Ethernet_putByte(ptr^) ;
        ptr := ptr + 1;
      end;

    // then puts the request string converted into upper char into
the transmit buffer
    while(reqLength <> 0) do
      begin
        SPI_Ethernet_putByte(SPI_Ethernet_getByte()) ;
        reqLength := reqLength - 1;
      end;

  result := len ;           // back to the library with the length
of the UDP reply
  end;

end.
```

HW Connection



Spi ethernet HW Conection

SPI Graphic Lcd Library

The mikroPascal PRO for AVR provides a library for operating Graphic Lcd 128x64 (with commonly used Samsung KS108/KS107 controller) via SPI interface.

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

Note: The library uses the SPI module for communication. User must initialize SPI module before using the SPI Graphic Lcd Library.

Note: This Library is designed to work with the mikroElektronika's Serial Lcd/Glcd Adapter Board pinout, see schematic at the bottom of this page for details.

External dependencies of SPI Graphic Lcd Library

The implementation of SPI Graphic Lcd Library routines is based on Port Expander Library routines.

Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

Basic routines:

- SPI_Glcd_Init
- SPI_Glcd_Set_Side
- SPI_Glcd_Set_Page
- SPI_Glcd_Set_X
- SPI_Glcd_Read_Data
- SPI_Glcd_Write_Data

Advanced routines:

- SPI_Glcd_Fill
- SPI_Glcd_Dot
- SPI_Glcd_Line
- SPI_Glcd_V_Line
- SPI_Glcd_H_Line
- SPI_Glcd_Rectangle
- SPI_Glcd_Box
- SPI_Glcd_Circle
- SPI_Glcd_Set_Font
- SPI_Glcd_Write_Char
- SPI_Glcd_Write_Text
- SPI_Glcd_Image

SPI_Glcd_Init

Prototype	`procedure SPI_Glcd_Init(DeviceAddress : byte);`
Returns	Nothing.
Description	Initializes the Glcd module via SPI interface. Parameters : - `DeviceAddress`: spi expander hardware address, see schematic at the bottom of this page
Requires	Global variables : - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.

Example	`// port expander pinout definition`
	`var SPExpanderCS   : sbit at PORTB.B1;`
	`     SPExpanderRST : sbit at PORTB.B0;`
	`     SPExpanderCS_Direction   : sbit at DDRB.B1;`
	`     SPExpanderRST_Direction : sbit at DDRB.B0;`
	`...`
	
	`// If Port Expander Library uses SPI1 module :`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV2, _SPI_CLK_HI_TRAILING);  // Initialize SPI module used with PortExpander`
	`SPI_Rd_Ptr := @SPI1_Read;                    // Pass pointer to SPI Read function of used SPI module`
	`SPI_Glcd_Init(0);`

SPI_Glcd_Set_Side

Prototype	`procedure SPI_Glcd_Set_Side(x_pos : byte);`
Returns	Nothing.
Description	Selects Glcd side. Refer to the Glcd datasheet for detail explanation. Parameters : - `x_pos`: position on x-axis. Valid values: 0..127 The parameter `x_pos` specifies the Glcd side: values from 0 to 63 specify the left side, values from 64 to 127 specify the right side. **Note:** For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	The following two lines are equivalent, and both of them select the left side of Glcd:
	`SPI_Glcd_Set_Side(0);`
	`SPI_Glcd_Set_Side(10);`

SPI_Glcd_Set_Page

Prototype	`procedure SPI_Glcd_Set_Page(page : byte);`
Returns	Nothing.
Description	Selects page of Glcd. Parameters : - `page`: page number. Valid values: 0..7 **Note:** For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.
Example	`SPI_Glcd_Set_Page(5);`

SPI_Glcd_Set_X

Prototype	`procedure SPI_Glcd_Set_X(x_pos : byte);`
Returns	Nothing.
Description	Sets x-axis position to x_pos dots from the left border of Glcd within the selected side. Parameters : - `x_pos`: position on x-axis. Valid values: 0..63 **Note:** For side, x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.
Example	`SPI_Glcd_Set_X(25);`

SPI_Glcd_Read_Data

Prototype	`function SPI_Glcd_Read_Data() : byte;`
Returns	One byte from Glcd memory.
Description	Reads data from the current location of Glcd memory and moves to the next location.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines. Glcd side, x-axis position and page should be set first. See the functions SPI_Glcd_Set_Side, SPI_Glcd_Set_X, and SPI_Glcd_Set_Page.

Example	`var data : byte;`
	`...`
	`data := SPI_Glcd_Read_Data();`

SPI_Glcd_Write_Data

Prototype	`procedure SPI_Glcd_Write_Data(ddata : byte);`
Returns	Nothing.
Description	Writes one byte to the current location in Glcd memory and moves to the next location. Parameters : - `Ddata`: data to be written
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines. Glcd side, x-axis position and page should be set first. See the functions SPI_Glcd_Set_Side, SPI_Glcd_Set_X, and SPI_Glcd_Set_Page.

Example	`var ddata : byte;`
	`...`
	`SPI_Glcd_Write_Data(ddata);`

SPI_Glcd_Fill

Prototype	`procedure SPI_Glcd_Fill(pattern: byte);`
Returns	Nothing.
Description	Fills Glcd memory with byte pattern. Parameters : - `pattern`: byte to fill Glcd memory with To clear the Glcd screen, use SPI_Glcd_Fill(0). To fill the screen completely, use SPI_Glcd_Fill(0xFF).
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Clear screen`
	`SPI_Glcd_Fill(0);`

SPI_Glcd_Dot

Prototype	`procedure SPI_Glcd_Dot(x_pos : byte; y_pos : byte; color : byte)`
Returns	Nothing.
Description	Draws a dot on Glcd at coordinates (`x_pos, y_pos`). Parameters : - `x_pos`: x position. Valid values: 0..127 - `y_pos`: y position. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state. **Note:** For x and y axis layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Invert the dot in the upper left corner`
	`SPI_Glcd_Dot(0, 0, 2);`

SPI_Glcd_Line

Prototype	`procedure SPI_Glcd_Line(x_start : integer; y_start : integer; x_end : integer; y_end : integer; color : byte);`
Returns	Nothing.
Description	Draws a line on Glcd. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `y_start`: y coordinate of the line start. Valid values: 0..63 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 Parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Draw a line between dots (0,0) and (20,30)`
	`SPI_Glcd_Line(0, 0, 20, 30, 1);`

SPI_Glcd_V_Line

Prototype	`procedure SPI_Glcd_V_Line(y_start: byte; y_end: byte; x_pos: byte; color: byte);`
Returns	Nothing.
Description	Draws a vertical line on Glcd. Parameters : - `y_start`: y coordinate of the line start. Valid values: 0..63 - `y_end`: y coordinate of the line end. Valid values: 0..63 - `x_pos`: x coordinate of vertical line. Valid values: 0..127 - `color`: color parameter. Valid values: 0..2 Parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Draw a vertical line between dots (10,5) and (10,25)`
	`SPI_Glcd_V_Line(5, 25, 10, 1);`

SPI_Glcd_H_Line

Prototype	`procedure SPI_Glcd_V_Line(x_start : byte; x_end : byte; y_pos : yte; color : byte);`
Returns	Nothing.
Description	Draws a horizontal line on Glcd. Parameters : - `x_start`: x coordinate of the line start. Valid values: 0..127 - `x_end`: x coordinate of the line end. Valid values: 0..127 - `y_pos`: y coordinate of horizontal line. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the line color: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Draw a horizontal line between dots (10,20) and (50,20)`
	`SPI_Glcd_H_Line(10, 50, 20, 1);`

SPI_Glcd_Rectangle

Prototype	`procedure SPI_Glcd_Rectangle(x_upper_left : byte; y_upper_left : byte; x_bottom_right : byte; y_bottom_right : byte; color : byte);`
Returns	Nothing.
Description	Draws a rectangle on Glcd. Parameters : - `x_upper_left`: x coordinate of the upper left rectangle corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left rectangle corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right rectangle corner. Valid val ues: 0..127 - `y_bottom_right`: y coordinate of the lower right rectangle corner. Valid val ues: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the rectangle border: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Draw a rectangle between dots (5,5) and (40,40)`
	`SPI_Glcd_Rectangle(5, 5, 40, 40, 1);`

SPI_Glcd_Box

Prototype	`procedure SPI_Glcd_Box(x_upper_left : byte; y_upper_left : byte; x_bottom_right : byte; y_bottom_right : byte; color : byte);`
Returns	Nothing.
Description	Draws a box on Glcd. Parameters : - `x_upper_left`: x coordinate of the upper left box corner. Valid values: 0..127 - `y_upper_left`: y coordinate of the upper left box corner. Valid values: 0..63 - `x_bottom_right`: x coordinate of the lower right box corner. Valid values: 0..127 - `y_bottom_right`: y coordinate of the lower right box corner. Valid values: 0..63 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the box fill: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Draw a box between dots (5,15) and (20,40)`
	`SPI_Glcd_Box(5, 15, 20, 40, 1);`

SPI_Glcd_Circle

Prototype	`procedure SPI_Glcd_Circle(x_center : integer; y_center : integer; radius : integer; color : byte);`
Returns	Nothing.
Description	Draws a circle on Glcd. Parameters : - `x_center`: x coordinate of the circle center. Valid values: 0..127 - `y_center`: y coordinate of the circle center. Valid values: 0..63 - `radius`: radius size - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the circle line: 0 white, 1 black, and 2 inverts each dot.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routine.

Example	`// Draw a circle with center in (50,50) and radius=10`
	`SPI_Glcd_Circle(50, 50, 10, 1);`

SPI_Glcd_Set_Font

Prototype	`procedure SPI_Glcd_Set_Font(activeFont : longint; aFontWidth : byte; aFontHeight : byte; aFontOffs : word);`
Returns	Nothing.
Description	Sets font that will be used with SPI_Glcd_Write_Char and SPI_Glcd_Write_Text routines. Parameters : - activeFont: font to be set. Needs to be formatted as an array of char - aFontWidth: width of the font characters in dots. - aFontHeight: height of the font characters in dots. - aFontOffs: number that represents difference between the mikroPascal PRO for AVR character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroPascal PRO for AVR character set, aFontOffs is 20). Demo fonts supplied with the library have an offset of 32, which means that they start with space. The user can use fonts given in the file “__Lib_GLCD_fonts.mpas” file located in the Uses folder or create his own fonts.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Use the custom 5x7 font "myfont" which starts with space (32):`
	`SPI_Glcd_Set_Font(@myfont, 5, 7, 32);`

SPI_Glcd_Write_Char

Prototype	`procedure SPI_Glcd_Write_Char(chr1 : byte; x_pos : byte; page_num : byte; color : byte);`
Returns	Nothing.
Description	Prints character on Glcd. Parameters : - `chr1`: character to be written - `x_pos`: character starting position on x-axis. Valid values: 0..(127-FontWidth) - `page_num`: the number of the page on which character will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the character: 0 white, 1 black, and 2 inverts each dot. **Note:** For x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines. Use the SPI_Glcd_Set_Font to specify the font for display; if no font is specified, then the default 5x8 font supplied with the library will be used.

Example	`// Write character 'C' on the position 10 inside the page 2:`
	`SPI_Glcd_Write_Char("C", 10, 2, 1);`

SPI_Glcd_Write_Text

Prototype	`procedure SPI_Glcd_Write_Text(var text : array[ 40] of byte; x_pos : byte; page_numb : byte; color : byte);`
Returns	Nothing.
Description	Prints text on Glcd. Parameters : - `text`: text to be written - `x_pos`: text starting position on x-axis. - `page_num`: the number of the page on which text will be written. Valid values: 0..7 - `color`: color parameter. Valid values: 0..2 The parameter color determines the color of the text: 0 white, 1 black, and 2 inverts each dot. **Note:** For x axis and page layout explanation see schematic at the bottom of this page.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines. Use the SPI_Glcd_Set_Font to specify the font for display; if no font is specified, then the default 5x8 font supplied with the library will be used.
Example	`// Write text "Hello world!" on the position 10 inside the page 2:SPI_Glcd_Write_Text("Hello world!", 10, 2, 1);`

SPI_Glcd_Image

Prototype	`procedure SPI_Glcd_Image(const image : ^byte);`
Returns	Nothing.
Description	Displays bitmap on Glcd. Parameters : - `image`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for AVR pointer to const and pointer to RAM equivalency). Use the mikroPascal PRO for AVR integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd.
Requires	Glcd needs to be initialized for SPI communication, see SPI_Glcd_Init routines.

Example	`// Draw image my_image on Glcd`
	`SPI_Glcd_Image(my_image);`

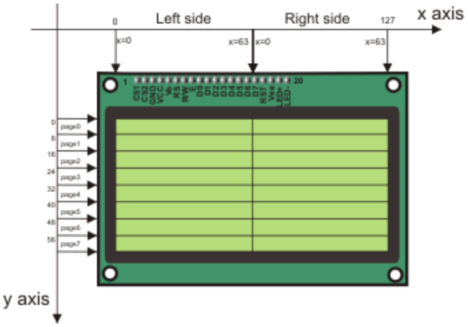
Library Example

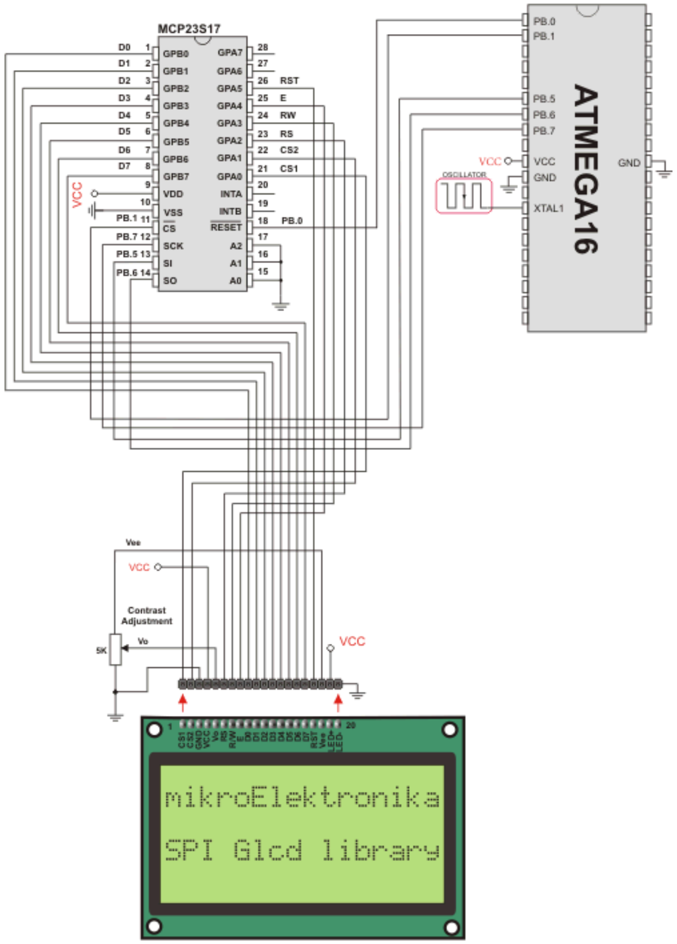
The example demonstrates how to communicate to KS0108 Glcd via the SPI module, using serial to parallel convertor MCP23S17.

```
program SPI_Glcd;

uses bitmap;

// Port Expander module connections
var SPExpanderRST : sbit at PORTB.0;
    SPExpanderCS  : sbit at PORTB.1;
    SPExpanderRST_Direction : sbit at DDRB.0;
    SPExpanderCS_Direction  : sbit at DDRB.1;
// End Port Expander module connections

var someText : array[ 20] of char;
    counter : byte;

procedure Delay2S;
  begin
    delay_ms(2000);
  end;

begin

// If Port Expander Library uses SPI1 module
   SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV2, _SPI_CLK_HI_TRAIL-
ING);  // Initialize SPI module used with PortExpander
  Spi_Rd_Ptr := @SPI1_Read;                                // Pass pointer
to SPI Read function of used SPI module
  // // If Port Expander Library uses SPI2 module
          //     SPI2_Init_Advanced(_SPI_MASTER,     _SPI_FCY_DIV2,
_SPI_CLK_HI_TRAILING);     //   Initialize   SPI   module   used   with
PortExpander
  // Spi_Rd_Ptr = @SPI2_Read;                              // Pass pointer to
SPI Read function of used SPI module

  SPI_Glcd_Init(0);                            // Initialize Glcd via SPI
  SPI_Glcd_Fill(0x00);                         // Clear Glcd

  while (TRUE) do
    begin

SPI_Glcd_Image(@truck_bmp);                    // Draw image
      Delay2s(); Delay2s();

SPI_Glcd_Fill(0x00);                           // Clear Glcd
      Delay2s;
```

```
         SPI_Glcd_Box(62,40,124,56,1);                    // Draw box
        SPI_Glcd_Rectangle(5,5,84,35,1);            // Draw rectangle
        SPI_Glcd_Line(0, 63, 127, 0,1);                // Draw line
        Delay2s();
        counter := 5;
        while (counter < 60) do  // Draw horizontal and vertical line
          begin
            Delay_ms(250);
            SPI_Glcd_V_Line(2, 54, counter, 1);
            SPI_Glcd_H_Line(2, 120, counter, 1);
            counter := counter + 5;
          end;
        Delay2s();

        SPI_Glcd_Fill(0x00);                              // Clear Glcd
         SPI_Glcd_Set_Font(@Character8x7, 8, 8, 32);   // Choose font
"Character8x7"
        SPI_Glcd_Write_Text('mikroE', 5, 7, 2);       // Write string

        for counter := 1 to 10 do// Draw circles
          SPI_Glcd_Circle(63,32, 3*counter, 1);
        Delay2s();

        SPI_Glcd_Box(12,20, 70,63, 2);                       // Draw box
        Delay2s();

        SPI_Glcd_Fill(0xFF);                                // Fill Glcd

        SPI_Glcd_Set_Font(@Character8x7, 8, 7, 32);     // Change font
        someText := '8x7 Font';
        SPI_Glcd_Write_Text(someText, 5, 1, 2);        // Write string
        Delay2s();

        SPI_Glcd_Set_Font(@System3x6, 3, 5, 32);        // Change font
        someText := '3X5 CAPITALS ONLY';
        SPI_Glcd_Write_Text(someText, 5, 3, 2);        // Write string
        Delay2s();

        SPI_Glcd_Set_Font(@font5x7, 5, 7, 32);          // Change font
        someText := '5x7 Font';
        SPI_Glcd_Write_Text(someText, 5, 5, 2);        // Write string
        Delay2s();

       SPI_Glcd_Set_Font(@FontSystem5x7_v2, 5, 7, 32);  // Change font
        someText := '5x7 Font (v2)';
  SPI_Glcd_Write_Text(someText, 5, 7, 2);              // Write string
        Delay2s();

    end;
  end.
```

HW Connection





SPI Glcd HW connection

SPI LCD LIBRARY

The mikroPascal PRO for AVR provides a library for communication with Lcd (with HD44780 compliant controllers) in 4-bit mode via SPI interface.

For creating a custom set of Lcd characters use Lcd Custom Character Tool.

Note: The library uses the SPI module for communication. The user must initialize the SPI module before using the SPI Lcd Library.

Note: This Library is designed to work with the mikroElektronika's Serial Lcd Adapter Board pinout. See schematic at the bottom of this page for details.

External dependencies of SPI Lcd Library

The implementation of SPI Lcd Library routines is based on Port Expander Library routines.

Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

- SPI_Lcd_Config
- SPI_Lcd_Out
- SPI_Lcd_Out_Cp
- SPI_Lcd_Chr
- SPI_Lcd_Chr_Cp
- SPI_Lcd_Cmd

SPI_Lcd_Config

Prototype	`procedure SPI_Lcd_Config(DeviceAddress: byte);`
Returns	Nothing.
Description	Initializes the Lcd module via SPI interface. Parameters : - `DeviceAddress`: spi expander hardware address, see schematic at the bottom of this page
Requires	Global variables : - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.

Example	`// port expander pinout definition`
	`var SPExpanderCS   : sbit at PORTB.B1;`
	`    SPExpanderRST : sbit at PORTB.B0;`
	`    SPExpanderCS_Direction   : sbit at DDRB.B1;`
	`    SPExpanderRST_Direction : sbit at DDRB.B0;`
	
	`// If Port Expander Library uses SPI1 module`
	`SPI1_Init();                              // Initialize SPI module used with PortExpander`
	`Spi_Rd_Ptr := @SPI1_Read;                 // Pass pointer to SPI Read function of used SPI module`
	`SPI_Lcd_Config(0);             // initialize lcd over spi interface`

SPI_Lcd_Out

Prototype	`procedure SPI_Lcd_Out(row: byte; column: byte; var text: array[20] of byte);`
Returns	Nothing.
Description	Prints text on the Lcd starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routines.

Example	`// Write text "Hello!" on Lcd starting from row 1, column 3:`
	`SPI_Lcd_Out(1, 3, "Hello!");`

SPI_Lcd_Out_Cp

Prototype	`procedure SPI_Lcd_Out_CP(var text : array[20] of byte);`
Returns	Nothing.
Description	Prints text on the Lcd at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routines.

Example	`// Write text "Here!" at current cursor position:`
	`SPI_Lcd_Out_CP("Here!");`

SPI_Lcd_Chr

Prototype	`procedure SPI_Lcd_Chr(Row : byte; Column : byte; Out_Char : byte);`
Returns	Nothing.
Description	Prints character on Lcd at specified position. Both variables and literals can be passed as character. Parameters : - `Row`: writing position row number - `Column`: writing position column number - `Out_Char`: character to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routines.

Example	`// Write character "i" at row 2, column 3:`
	`SPI_Lcd_Chr(2, 3, 'i');`

SPI_Lcd_Chr_Cp

Prototype	`procedure SPI_Lcd_Chr_CP(Out_Char : byte);`
Returns	Nothing.
Description	Prints character on Lcd at current cursor position. Both variables and literals can be passed as character. Parameters : - `Out_Char`: character to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routines.

Example	`// Write character "e" at current cursor position:`
	`SPI_Lcd_Chr_Cp('e');`

SPI_Lcd_Cmd

Prototype	`procedure SPI_Lcd_Cmd(out_char : byte);`
Returns	Nothing.
Description	Sends command to Lcd. Parameters : - `out_char`: command to be sent **Note:** Predefined constants can be passed to the function, see Available Lcd Commands.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd_Config routines.

Example	`// Clear Lcd display:`
	`SPI_Lcd_Cmd(LCD_CLEAR);`

Available SPI Lcd Commands

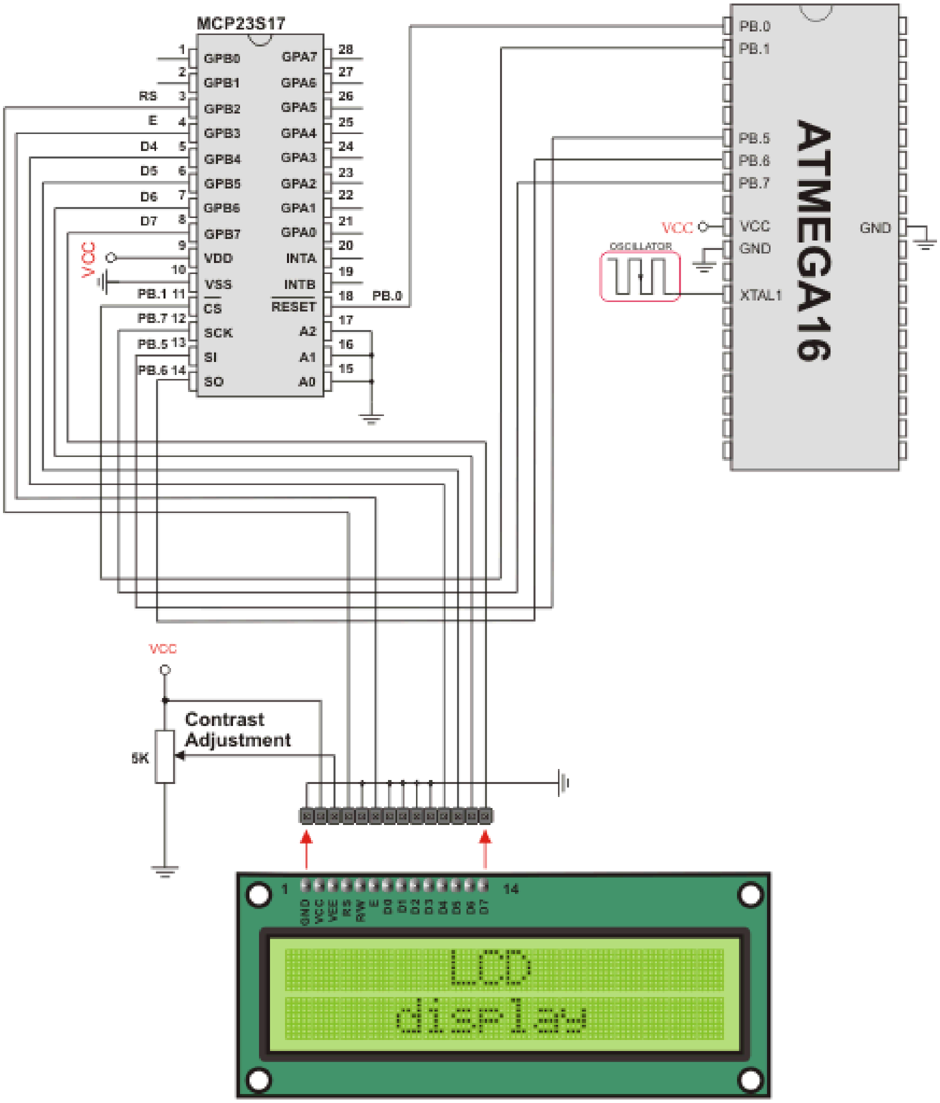
Lcd Command	Purpose
`LCD_FIRST_ROW`	Move cursor to the 1st row
`LCD_SECOND_ROW`	Move cursor to the 2nd row
`LCD_THIRD_ROW`	Move cursor to the 3rd row
`LCD_FOURTH_ROW`	Move cursor to the 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to its 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

This example demonstrates how to communicate Lcd via the SPI module, using serial to parallel convertor MCP23S17.

```
program SPI_Lcd;

var text : array[16] of char;

// Port Expander module connections
var SPExpanderRST : sbit at PORTB.B0;
    SPExpanderCS  : sbit at PORTB.B1;
    SPExpanderRST_Direction : sbit at DDRB.B0;
    SPExpanderCS_Direction  : sbit at DDRB.B1;
// End Port Expander module connections

begin
  text := 'mikroElektronika';
  // If Port Expander Library uses SPI1 module
  SPI1_Init();                                 // Initialize SPI module
used with PortExpander
  Spi_Rd_Ptr := @SPI1_Read;                    // Pass pointer to SPI Read
function of used SPI module

  // If Port Expander Library uses SPI2 module
  // SPI2_Init();                              // Initialize SPI module
used with PortExpander
  // Spi_Rd_Ptr = @SPI2_Read;                  // Pass pointer to SPI Read
function of used SPI module
  SPI_Lcd_Config(0);          // Initialize Lcd over SPI interface
  SPI_Lcd_Cmd(LCD_CLEAR);                     // Clear display
  SPI_Lcd_Cmd(LCD_CURSOR_OFF);                // Turn cursor off
  SPI_Lcd_Out(1,6, 'mikroE');                 // Print text to Lcd, 1st
row, 6th column
  SPI_Lcd_Chr_CP('!');                        // Append '!'
  SPI_Lcd_Out(2,1, text); // Print text to Lcd, 2nd row, 1st column
 end.
```

HW Connection



SPI Lcd HW connection

SPI LCD8 (8-BIT INTERFACE) LIBRARY

The mikroPascal PRO for AVR provides a library for communication with Lcd (with HD44780 compliant controllers) in 8-bit mode via SPI interface.

For creating a custom set of Lcd characters use Lcd Custom Character Tool.

Note: Library uses the SPI module for communication. The user must initialize the SPI module before using the SPI Lcd Library.

Note: This Library is designed to work with mikroElektronika's Serial Lcd/Glcd Adapter Board pinout, see schematic at the bottom of this page for details.

External dependencies of SPI Lcd Library

The implementation of SPI Lcd Library routines is based on Port Expander Library routines.

Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

- SPI_Lcd8_Config
- SPI_Lcd8_Out
- SPI_Lcd8_Out_Cp
- SPI_Lcd8_Chr
- SPI_Lcd8_Chr_Cp
- SPI_Lcd8_Cmd

SPI_Lcd8_Config

Prototype	`procedure SPI_Lcd8_Config(DeviceAddress : byte);`
Returns	Nothing.
Description	Initializes the Lcd module via SPI interface. Parameters : - `DeviceAddress`: spi expander hardware address, see schematic at the bottom of this page
Requires	Global variables : - `SPExpanderCS`: Chip Select line - `SPExpanderRST`: Reset line - `SPExpanderCS_Direction`: Direction of the Chip Select pin - `SPExpanderRST_Direction`: Direction of the Reset pin must be defined before using this function. SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.

Example	`// port expander pinout definition`
	`var SPExpanderCS   : sbit at PORTB.B1;`
	`    SPExpanderRST : sbit at PORTB.B0;`
	`    SPExpanderCS_Direction   : sbit at DDRB.B1;`
	`    SPExpanderRST_Direction : sbit at DDRB.B0;`
	`SPI1_Init();                                  // Initialize SPI module used with PortExpander`
	`Spi_Rd_Ptr := @SPI1_Read;                     // Pass pointer to SPI Read function of used SPI module`
	`...`
	`SPI_Lcd8_Config(0);                          // intialize lcd in 8bit mode via spi`

SPI_Lcd8_Out

Prototype	`procedure SPI_Lcd8_Out(row: byte; column: byte; var text: array[20] of byte);`
Returns	Nothing.
Description	Prints text on Lcd starting from specified position. Both string variables and literals can be passed as a text. Parameters : - `row`: starting position row number - `column`: starting position column number - `text`: text to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routines.

Example	`// Write text "Hello!" on Lcd starting from row 1, column 3:`
	`SPI_Lcd8_Out(1, 3, "Hello!");`

SPI_Lcd8_Out_Cp

Prototype	`procedure SPI_Lcd8_Out_CP(var text : array[20] of byte);`
Returns	Nothing.
Description	Prints text on Lcd at current cursor position. Both string variables and literals can be passed as a text. Parameters : - `text`: text to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routines.

Example	`// Write text "Here!" at current cursor position:`
	`SPI_Lcd8_Out_Cp("Here!");`

SPI_Lcd8_Chr

Prototype	`procedure SPI_Lcd8_Chr(Row : byte; Column : byte; Out_Char : byte);`
Returns	Nothing.
Description	Prints character on Lcd at specified position. Both variables and literals can be passed as character. Parameters : - `row`: writing position row number - `column`: writing position column number - `out_char`: character to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routines.

Example	`// Write character "i" at row 2, column 3:`
	`SPI_Lcd8_Chr(2, 3, 'i');`

SPI_Lcd8_Chr_Cp

Prototype	`procedure SPI_Lcd8_Chr_CP(Out_Char : byte);`
Returns	Nothing.
Description	Prints character on Lcd at current cursor position. Both variables and literals can be passed as character. Parameters : - `out_char` : character to be written
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routines.

Example	Print “e” at current cursor position:
	`// Write character "e" at current cursor position:`
	`SPI_Lcd8_Chr_Cp('e');`

SPI_Lcd8_Cmd

Prototype	`procedure SPI_Lcd8_Cmd(out_char : byte);`
Returns	Nothing.
Description	Sends command to Lcd. Parameters : - `out_char`: command to be sent **Note:** Predefined constants can be passed to the function, see Available Lcd Commands.
Requires	Lcd needs to be initialized for SPI communication, see SPI_Lcd8_Config routines.

Example	`// Clear Lcd display:`
	`SPI_Lcd8_Cmd(LCD_CLEAR);`

Available SPI Lcd8 Commands

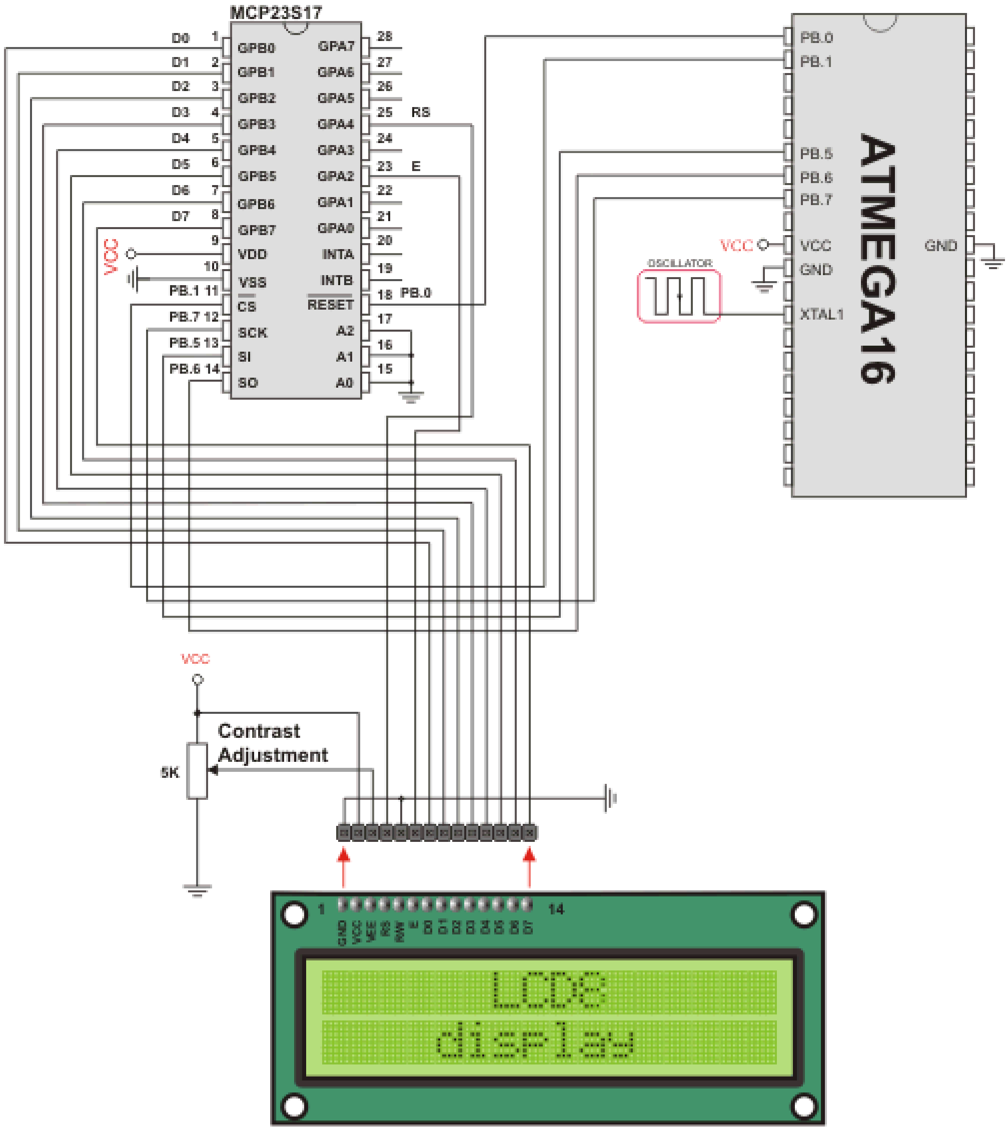
Lcd Command	Purpose
`LCD_FIRST_ROW`	Move cursor to the 1st row
`LCD_SECOND_ROW`	Move cursor to the 2nd row
`LCD_THIRD_ROW`	Move cursor to the 3rd row
`LCD_FOURTH_ROW`	Move cursor to the 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to its 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

This example demonstrates how to communicate Lcd in 8-bit mode via the SPI module, using serial to parallel convertor MCP23S17.

```
program SPI_Lcd8_Test;

var text : array[ 16]  of char;

// Port Expander module connections
var SPExpanderRST : sbit at PORTB.B0;
    SPExpanderCS   : sbit at PORTB.B1;
    SPExpanderRST_Direction : sbit at DDRB.B0;
    SPExpanderCS_Direction  : sbit at DDRB.B1;
// End Port Expander module connections

begin

  text := 'mikroE';
  // If Port Expander Library uses SPI1 module
  SPI1_Init();                                 // Initialize SPI mod-
ule used with PortExpander
  Spi_Rd_Ptr := @SPI1_Read;                    // Pass pointer to SPI
Read function of used SPI module

  // // If Port Expander Library uses SPI2 module
  // SPI2_Init();                              // Initialize SPI mod-
ule used with PortExpander
  // Spi_Rd_Ptr = @SPI2_Read;                  // Pass pointer to SPI
Read function of used SPI module

  SPI_Lcd8_Config(0);                        // Intialize Lcd in 8bit
mode via SPI
  SPI_Lcd8_Cmd(LCD_CLEAR);                   // Clear display
  SPI_Lcd8_Cmd(LCD_CURSOR_OFF);              // Turn cursor off
  SPI_Lcd8_Out(1,6, text);                   // Print text to Lcd, 1st
row, 6th column...
  SPI_Lcd8_Chr_CP('!');                      // Append '!'
  SPI_Lcd8_Out(2,1, 'mikroelektronika');    // Print text to Lcd, 2nd
row, 1st column...
  SPI_Lcd8_Out(3,1, text); // For Lcd modules with more than two rows
  SPI_Lcd8_Out(4,15, text)// For Lcd modules with more than two rows
end.
```

HW Connection



SPI Lcd8 HW connection

SPI T6963C Graphic Lcd Library

The mikroPascal PRO for AVR provides a library for working with Glcds based on TOSHIBA T6963C controller via SPI interface. The Toshiba T6963C is a very popular Lcd controller for the use in small graphics modules. It is capable of controlling displays with a resolution up to 240x128. Because of its low power and small outline it is most suitable for mobile applications such as PDAs, MP3 players or mobile measurement equipment. Although this controller is small, it has a capability of displaying and merging text and graphics and it manages all interfacing signals to the displays Row and Column drivers.

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

Note: The library uses the SPI module for communication. The user must initialize SPI module before using the SPI T6963C Glcd Library.

Note: This Library is designed to work with mikroElektronika's Serial Glcd 240x128 and 240x64 Adapter Boards pinout, see schematic at the bottom of this page for details.

Note: Some mikroElektronika's adapter boards have pinout different from T6369C datasheets. Appropriate relations between these labels are given in the table below:

Adapter Board	T6369C datasheet
RS	C/D
R/W	/RD
E	/WR

External dependencies of SPI T6963C Graphic Lcd Library

The implementation of SPI T6963C Graphic Lcd Library routines is based on Port Expander Library routines.

Prior to calling any of this library routines, Spi_Rd_Ptr needs to be initialized with the appropriate SPI_Read routine.

External dependencies are the same as Port Expander Library external dependencies.

Library Routines

- SPI_T6963C_Config
- SPI_T6963C_WriteData
- SPI_T6963C_WriteCommand
- SPI_T6963C_SetPtr
- SPI_T6963C_WaitReady
- SPI_T6963C_Fill
- SPI_T6963C_Dot
- SPI_T6963C_Write_Char
- SPI_T6963C_Write_Text
- SPI_T6963C_Line
- SPI_T6963C_Rectangle
- SPI_T6963C_Box
- SPI_T6963C_Circle
- SPI_T6963C_Image
- SPI_T6963C_Sprite
- SPI_T6963C_Set_Cursor
- SPI_T6963C_ClearBit
- SPI_T6963C_SetBit
- SPI_T6963C_NegBit
- SPI_T6963C_DisplayGrPanel
- SPI_T6963C_DisplayTxtPanel
- SPI_T6963C_SetGrPanel
- SPI_T6963C_SetTxtPanel
- SPI_T6963C_PanelFill
- SPI_T6963C_GrFill
- SPI_T6963C_TxtFill
- SPI_T6963C_Cursor_Height
- SPI_T6963C_Graphics
- SPI_T6963C_Text
- SPI_T6963C_Cursor
- SPI_T6963C_Cursor_Blink

SPI_T6963C_Config

Prototype	`procedure SPI_T6963C_Config(width : word; height : word; fntW :word; DeviceAddress : byte; wr : byte; rd : byte; cd : byte; rst : byte);`
Returns	Nothing.

Description	Initalizes the Graphic Lcd controller.
	Parameters :
	- `width`: width of the Glcd panel
	- `height`: height of the Glcd panel
	- `fntW`: font width
	- `DeviceAddress`: SPI expander hardware address, see schematic at the bottom of this page
	- `wr`: write signal pin on Glcd control port
	- `rd`: read signal pin on Glcd control port
	- `cd`: command/data signal pin on Glcd control port
	- `rst`: reset signal pin on Glcd control port
	Display RAM organization:
	The library cuts RAM into panels : a complete panel is one graphics panel followed by a text panel (see schematic below).
	`schematic:`
	`+---------------------+ /\`
	`+ GRAPHICS PANEL #0    +  |`
	`+                       +   |`
	`+                       +   |`
	`+                       +   |`
	`+---------------------+   | PANEL 0`
	`+ TEXT PANEL #0         +  |`
	`+                       + \/`
	`+---------------------+ /\`
	`+ GRAPHICS PANEL #1    +  |`
	`+                       +   |`
	`+                       +   |`
	`+                       +   |`
	`+---------------------+   | PANEL 1`
	`+ TEXT PANEL #2         +  |`
	`+                       +   |`
	`+---------------------+ \/`

Requires	Global variables : - SPExpanderCS: Chip Select line - SPExpanderRST: Reset line - SPExpanderCS_Direction: Direction of the Chip Select pin - SPExpanderRST_Direction: Direction of the Reset pin must be defined before using this function. SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.

Example	`' port expander pinout definition`
	`var SPExpanderCS as sbit at PORTB.B1`
	`    SPExpanderRST as sbit at PORTB.B0`
	`    SPExpanderCS_Direction as sbit at DDRB.B1`
	`    SPExpanderRST_Direction as sbit at DDRB.B0`
	`...`
	`// Initialize SPI module`
	`SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV32, _SPI_CLK_HI_TRAILING);`
	`SPI_Rd_Ptr := @SPI1_Read;                    // Pass pointer to SPI Read function of used SPI module`
	`SPI_T6963C_Config(240, 64, 8, 0, 0, 1, 3, 4);`

SPI_T6963C_WriteData

Prototype	`procedure SPI_T6963C_WriteData(Ddata : byte);`
Returns	Nothing.
Description	Writes data to T6963C controller via SPI interface. Parameters : - Ddata: data to be written
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_WriteData(AddrL);`

SPI_T6963C_WriteCommand

Prototype	`procedure SPI_T6963C_WriteCommand(Ddata : byte);`
Returns	Nothing.
Description	Writes command to T6963C controller via SPI interface. Parameters : - `Ddata`: command to be written
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_WriteCommand(SPI_T6963C_CURSOR_POINTER_SET);`

SPI_T6963C_SetPtr

Prototype	`procedure SPI_T6963C_SetPtr(p : word; c : byte);`
Returns	Nothing.
Description	Sets the memory pointer p for command c. Parameters : - `p`: address where command should be written - `c`: command to be written
Requires	SToshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`SPI_T6963C_SetPtr(T6963C_grHomeAddr + start,`
	`T6963C_ADDRESS_POINTER_SET);`

SPI_T6963C_WaitReady

Prototype	`procedure SPI_T6963C_WaitReady();`
Returns	Nothing.
Description	Pools the status byte, and loops until Toshiba Glcd module is ready.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_WaitReady();`

SPI_T6963C_Fill

Prototype	`procedure SPI_T6963C_Fill(v : byte; start : word; len : word);`
Returns	Nothing.
Description	Fills controller memory block with given byte. Parameters : - `v`: byte to be written - `start`: starting address of the memory block - `len`: length of the memory block in bytes
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Fill(0x33; 0x00FF; 0x000F);`

SPI_T6963C_Dot

Prototype	`procedure SPI_T6963C_Dot(x : integer; y : integer; color : byte)`
Returns	Nothing.
Description	Draws a dot in the current graphic panel of Glcd at coordinates (x, y). Parameters : - `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Dot(x0, y0, pcolor);`

SPI_T6963C_Write_Char

Prototype	`procedure SPI_T6963C_Write_Char(c : byte; x : byte; y : byte; mode : byte);`
Returns	Nothing.
Description	Writes a char in the current text panel of Glcd at coordinates (x, y). Parameters : - `c`: char to be written - `x`: char position on x-axis - `y`: char position on y-axis - `mode`: mode parameter. Valid values: SPI_T6963C_ROM_MODE_OR, SPI_T6963C_ROM_MODE_XOR, SPI_T6963C_ROM_MODE_AND and SPI_T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Write_Char("A",22,23,AND);`

SPI_T6963C_Write_Text

Prototype	`procedure SPI_T6963C_write_text(var str : array[10] of byte; x, y, mode : byte);`
Returns	Nothing.
Description	Writes text in the current text panel of Glcd at coordinates (x, y). Parameters : - `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: SPI_T6963C_ROM_MODE_OR, SPI_T6963C_ROM_MODE_XOR, SPI_T6963C_ROM_MODE_AND and SPI_T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the logical “exclusive OR”. This can be useful to display text in negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on the display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Write_Text('GLCD LIBRARY DEMO, WELCOME !', 0, 0, T6963C_ROM_MODE_EXOR);`

SPI_T6963C_Line

Prototype	`procedure SPI_T6963C_Line(x0 : integer; y0 : integer; x1 : integer; y1 : integer; pcolor : byte);`
Returns	Nothing.
Description	Draws a line from (x0, y0) to (x1, y1). Parameters : - `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Line(0, 0, 239, 127, T6963C_WHITE);`

SPI_T6963C_Rectangle

Prototype	`procedure SPI_T6963C_Rectangle(x0 : integer; y0 : integer; x1 : integer; y1 : integer; pcolor : byte);`
Returns	Nothing.
Description	Draws a rectangle on Glcd. Parameters : - `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Rectangle(20, 20, 219, 107, T6963C_WHITE);`

SPI_T6963C_Box

Prototype	`procedure SPI_T6963C_Box(x0 : integer; y0 : integer; x1 : integer; y1 : integer; pcolor : byte);`
Returns	Nothing.
Description	Draws a box on the Glcd Parameters : - `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Box(0, 119, 239, 127, T6963C_WHITE);`

SPI_T6963C_Circle

Prototype	`procedure SPI_T6963C_Circle(x : integer; y : integer; r : longint; pcolor : word);`
Returns	Nothing.
Description	Draws a circle on the Glcd. Parameters : - `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: SPI_T6963C_BLACK and SPI_T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Circle(120, 64, 110, T6963C_WHITE);`

SPI_T6963C_Image

Prototype	`procedure SPI_T6963C_image(const pic : ^byte);`
Returns	Nothing.
Description	Displays bitmap on Glcd. Parameters : - `pic`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for AVR pointer to const and pointer to RAM equivalency). Use the mikroPascal's integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Image(my_image);`

SPI_T6963C_Sprite

Prototype	`procedure SPI_T6963C_sprite(px, py : byte; const pic : ^byte; sx, sy : byte);`
Returns	Nothing.
Description	Fills graphic rectangle area (px, py) to (px+sx, py+sy) with custom size picture. Parameters : - `px`: x coordinate of the upper left picture corner. Valid values: multiples of the font width - `py`: y coordinate of the upper left picture corner - `pic`: picture to be displayed - `sx`: picture width. Valid values: multiples of the font width - `sy`: picture height **Note:** If px and sx parameters are not multiples of the font width they will be scaled to the nearest lower number that is a multiple of the font width.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Sprite(76, 4, einstein, 88, 119); // draw a sprite`

SPI_T6963C_Set_Cursor

Prototype	`procedure SPI_T6963C_set_cursor(x, y : byte);`
Returns	Nothing.
Description	Sets cursor to row x and column y. Parameters : - `x`: cursor position row number - `y`: cursor position column number
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Set_Cursor(cposx, cposy);`

SPI_T6963C_ClearBit

Prototype	`procedure SPI_T6963C_clearBit(b : byte);`
Returns	Nothing.
Description	Clears control port bit(s). Parameters : - `b`: bit mask. The function will clear bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// clear bits 0 and 1 on control port`
	`SPI_T6963C_ClearBit(0x03);`

SPI_T6963C_SetBit

Prototype	`procedure SPI_T6963C_setBit(b : byte);`
Returns	Nothing.
Description	Sets control port bit(s). Parameters : - `b`: bit mask. The function will set bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// set bits 0 and 1 on control port`
	`SPI_T6963C_SetBit(0x03);`

SPI_T6963C_NegBit

Prototype	`procedure SPI_T6963C_negBit(b : byte);`
Returns	Nothing.
Description	Negates control port bit(s). Parameters : - `b`: bit mask. The function will negate bit x on control port if bit x in bit mask is set to 1.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// negate bits 0 and 1 on control port`
	`SPI_T6963C_NegBit(0x03);`

SPI_T6963C_DisplayGrPanel

Prototype	`procedure SPI_T6963C_DisplayGrPanel(n : byte);`
Returns	Nothing.
Description	Display selected graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// display graphic panel 1`
	`SPI_T6963C_DisplayGrPanel(1);`

SPI_T6963C_DisplayTxtPanel

Prototype	`procedure SPI_T6963C_DisplayTxtPanel(n : byte);`
Returns	Nothing.
Description	Display selected text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// display text panel 1`
	`SPI_T6963C_DisplayTxtPanel(1);`

SPI_T6963C_SetGrPanel

Prototype	`procedure SPI_T6963C_SetGrPanel(n : byte);`
Returns	Nothing.
Description	Compute start address for selected graphic panel and set appropriate internal pointers. All subsequent graphic operations will be preformed at this graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// set graphic panel 1 as current graphic panel.`
	`SPI_T6963C_SetGrPanel(1);`

SPI_T6963C_SetTxtPanel

Prototype	`procedure SPI_T6963C_SetTxtPanel(n : byte);`
Returns	Nothing.
Description	Compute start address for selected text panel and set appropriate internal pointers. All subsequent text operations will be preformed at this text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// set text panel 1 as current text panel.`
	`SPI_T6963C_SetTxtPanel(1);`

SPI_T6963C_PanelFill

Prototype	`procedure SPI_T6963C_PanelFill(v : byte);`
Returns	Nothing.
Description	Fill current panel in full (graphic+text) with appropriate value (0 to clear). Parameters : - `v`: value to fill panel with.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`clear current panel`
	`SPI_T6963C_PanelFill(0);`

SPI_T6963C_GrFill

Prototype	`procedure SPI_T6963C_GrFill(v : byte);`
Returns	Nothing.
Description	Fill current graphic panel with appropriate value (0 to clear). Parameters : - `v`: value to fill graphic panel with.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// clear current graphic panel`
	`SPI_T6963C_GrFill(0);`

SPI_T6963C_TxtFill

Prototype	`procedure SPI_T6963C_TxtFill(v : byte);`
Returns	Nothing.
Description	Fill current text panel with appropriate value (0 to clear). Parameters : - `v`: this value increased by 32 will be used to fill text panel.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// clear current text panel`
	`SPI_T6963C_TxtFill(0);`

SPI_T6963C_Cursor_Height

Prototype	`procedure SPI_T6963C_Cursor_Height(n : byte);`
Returns	Nothing.
Description	Set cursor size. Parameters : - `n`: cursor height. Valid values: 0..7.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_Cursor_Height(7);`

SPI_T6963C_Graphics

Prototype	`procedure SPI_T6963C_Graphics(n : byte);`
Returns	Nothing.
Description	Enable/disable graphic displaying. Parameters : - `n`: graphic enable/disable parameter. Valid values: 0 (disable graphic dispaying) and 1 (enable graphic displaying).
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// enable graphic displaying`
	`SPI_T6963C_Graphics(1);`

SPI_T6963C_Text

Prototype	`procedure SPI_T6963C_Text(n : byte);`
Returns	Nothing.
Description	Enable/disable text displaying. Parameters : - `n`: text enable/disable parameter. Valid values: 0 (disable text dispaying) and 1 (enable text displaying).
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// enable text displaying`
	`SPI_T6963C_Text(1);`

SPI_T6963C_Cursor

Prototype	`procedure SPI_T6963C_Cursor(n : byte);`
Returns	Nothing.
Description	Set cursor on/off. Parameters : - `n`: on/off parameter. Valid values: 0 (set cursor off) and 1 (set cursor on).
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// set cursor on`
	`SPI_T6963C_Cursor(1);`

SPI_T6963C_Cursor_Blink

Prototype	`procedure SPI_T6963C_Cursor_Blink(n : byte);`
Returns	Nothing.
Description	Enable/disable cursor blinking. Parameters : - `n`: cursor blinking enable/disable parameter. Valid values: 0 (disable cursor blinking) and 1 (enable cursor blinking).
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.

Example	`// enable cursor blinking`
	`SPI_T6963C_Cursor_Blink(1);`

Library Example

The following drawing demo tests advanced routines of the SPI T6963C Glcd library. Hardware configurations in this example are made for the T6963C 240x128 display, EasyAVR5A board and ATmega16.

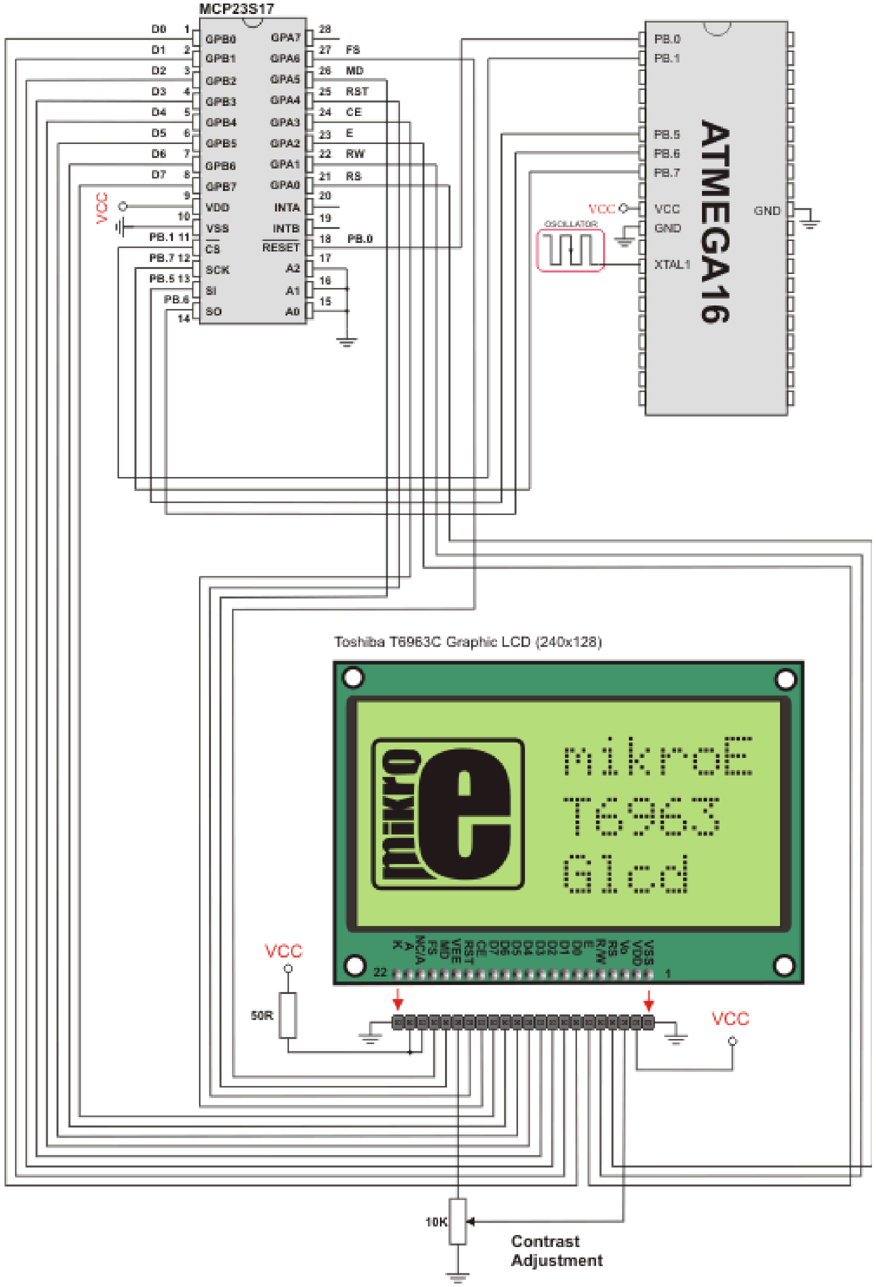
```
program SPI_T6963C_240x128;

uses __Lib_SPIT6963C_Const, bitmap, bitmap2;

var
// Port Expander module connections
    SPExpanderRST : sbit at PORTB.B0;
    SPExpanderCS  : sbit at PORTB.B1;
    SPExpanderRST_Direction : sbit at DDRB.B0;
    SPExpanderCS_Direction  : sbit at DDRB.B1;
// End Port Expander module connections

var   panel : byte;            // current panel
          i : word;           // general purpose register
       curs : byte;           // cursor visibility
      cposx,
      cposy : word;           // cursor x-y position
      txt, txt1 : string[29];

begin

  txt1 := ' EINSTEIN WOULD HAVE LIKED mE';
  txt  := ' GLCD LIBRARY DEMO, WELCOME !';
```

```
DDRA := 0x00;                       // configure PORTA as input

  {*
  * init display for 240 pixel width and 128 pixel height
  * 8 bits character width
  * data bus on MCP23S17 portB
  * control bus on MCP23S17 portA
  * bit 2 is !WR
  * bit 1 is !RD
  * bit 0 is !CD
  * bit 4 is RST
  * chip enable, reverse on, 8x8 font internaly set in library
  *}

  // Pass pointer to SPI Read function of used SPI module
  Spi_Rd_Ptr := @SPI1_Read;

  // Initialize SPI module
   SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV2, _SPI_CLK_HI_TRAIL-
ING);

  // // If Port Expander Library uses SPI2 module
  // Pass pointer to SPI Read function of used SPI module
  // Spi_Rd_Ptr = @SPI2_Read;            // Pass pointer to SPI Read
function of used SPI module

  // Initialize SPI module used with PortExpander
         //     SPI2_Init_Advanced(_SPI_MASTER,     _SPI_FCY_DIV2,
_SPI_CLK_HI_TRAILING);


  // Initialize SPI Toshiba 240x128
  SPI_T6963C_Config(240, 128, 8, 0, 2, 1, 0, 4) ;
  //Delay_ms(1000);

  {*
   * Enable both graphics and text display at the same time
   *}

  SPI_T6963C_graphics(1) ;

  SPI_T6963C_text(1) ;

  panel := 0 ;
  i := 0 ;
  curs := 0 ;
  cposx := 0;
  cposy := 0 ;
```

```
{*
   * Text messages
   *}
  SPI_T6963C_write_text(txt, 0, 0, SPI_T6963C_ROM_MODE_XOR) ;
  SPI_T6963C_write_text(txt1, 0, 15, SPI_T6963C_ROM_MODE_XOR) ;

  {*
   * Cursor
   *}
  SPI_T6963C_cursor_height(8) ;         // 8 pixel height
  SPI_T6963C_set_cursor(0, 0) ;         // move cursor to top left
  SPI_T6963C_cursor(0) ;                // cursor off

  {*
   * Draw rectangles
   *}
  SPI_T6963C_rectangle(0, 0, 239, 127, SPI_T6963C_WHITE) ;
  SPI_T6963C_rectangle(20, 20, 219, 107, SPI_T6963C_WHITE) ;
  SPI_T6963C_rectangle(40, 40, 199, 87, SPI_T6963C_WHITE) ;
  SPI_T6963C_rectangle(60, 60, 179, 67, SPI_T6963C_WHITE) ;


  {*
   * Draw a cross
   *}
  SPI_T6963C_line(0, 0, 239, 127, SPI_T6963C_WHITE) ;
  SPI_T6963C_line(0, 127, 239, 0, SPI_T6963C_WHITE) ;

  {*
   * Draw solid boxes
   *}
  SPI_T6963C_box(0, 0, 239, 8, SPI_T6963C_WHITE) ;
  SPI_T6963C_box(0, 119, 239, 127, SPI_T6963C_WHITE) ;

  {*
   * Draw circles
   *}
  SPI_T6963C_circle(120, 64, 10,  SPI_T6963C_WHITE) ;
  SPI_T6963C_circle(120, 64, 30,  SPI_T6963C_WHITE) ;
  SPI_T6963C_circle(120, 64, 50,  SPI_T6963C_WHITE) ;
  SPI_T6963C_circle(120, 64, 70,  SPI_T6963C_WHITE) ;
  SPI_T6963C_circle(120, 64, 90,  SPI_T6963C_WHITE) ;
  SPI_T6963C_circle(120, 64, 110, SPI_T6963C_WHITE) ;
  SPI_T6963C_circle(120, 64, 130, SPI_T6963C_WHITE) ;

  SPI_T6963C_sprite(76, 4, @einstein, 88, 119) ;   // Draw a sprite
  SPI_T6963C_setGrPanel(1) ;           // Select other graphic panel
  SPI_T6963C_image(@me) ;  // Fill the graphic screen with a picture
```

```
 while (TRUE) do                                // Endless loop
    begin

      {*
       * If PORTA_0 is pressed, toggle the display between graphic
panel 0 and graphic 1
       *}
      if( PINA0_bit = 0)  then
        begin
          Inc(panel) ;
          panel := panel and 1;
                       SPI_T6963C_setPtr((SPI_T6963C_grMemSize   +
SPI_T6963C_txtMemSize) * panel, SPI_T6963C_GRAPHIC_HOME_ADDRESS_SET)
;
          Delay_ms(300) ;
        end

      {*
       * If PORTA_1 is pressed, display only graphic panel
       *}
      else
        if ( PINA1_bit = 0)  then
          begin
            SPI_T6963C_graphics(1) ;
            SPI_T6963C_text(0) ;
            Delay_ms(300) ;
          end

      {*
       * If PORTA_2 is pressed, display only text panel
       *}
        else
          if ( PINA2_bit = 0)  then
            begin
              SPI_T6963C_graphics(0) ;
              SPI_T6963C_text(1) ;
              Delay_ms(300) ;
            end

      {*
       * If PORTA_3 is pressed, display text and graphic panels
       *}
          else
            if ( PINA3_bit = 0)  then
              begin
                SPI_T6963C_graphics(1) ;
          SPI_T6963C_text(1) ;
                Delay_ms(300) ;
              end
 {*
       * If PORTA_4 is pressed, change cursor
       *}
             else
               if( PINA4_bits = 0) then
                 begin
                   Inc(curs);
                   if (curs = 3) then
                     curs := 0;
                   case curs of
                     0:
                        // no cursor
                        SPI_T6963C_cursor(0) ;

                     1: begin
                          // blinking cursor
                          SPI_T6963C_cursor(1) ;
                          SPI_T6963C_cursor_blink(1) ;
                        end;
                     2: begin
                          // non blinking cursor
                          SPI_T6963C_cursor(1) ;
                          SPI_T6963C_cursor_blink(0) ;
                        end;
                   end;
                   Delay_ms(300) ;
                 end;

      {*
       * Move cursor, even if not visible
       *}
      Inc(cposx);
      if (cposx = SPI_T6963C_txtCols) then
        begin
          cposx := 0 ;
          Inc(cposy);
                           if    (cposy    =    SPI_T6963C_grHeight    /
SPI_T6963C_CHARACTER_HEIGHT) then
            cposy := 0 ;
        end;
      SPI_T6963C_set_cursor(cposx, cposy) ;

      Delay_ms(100) ;
    end;
end.
```

HW Connection



SPI T6963C Glcd HW connection

T6963C GRAPHIC LCD LIBRARY

The mikroPascal PRO for AVR provides a library for working with Glcds based on TOSHIBA T6963C controller. The Toshiba T6963C is a very popular Lcd controller for the use in small graphics modules. It is capable of controlling displays with a resolution up to 240x128. Because of its low power and small outline it is most suitable for mobile applications such as PDAs, MP3 players or mobile measurement equipment. Although small, this contoller has a capability of displaying and merging text and graphics and it manages all the interfacing signals to the displays Row and Column drivers.

For creating a custom set of Glcd images use Glcd Bitmap Editor Tool.

Note: ChipEnable(CE), FontSelect(FS) and Reverse(MD) have to be set to appropriate levels by the user outside of the `T6963C_Init` function. See the Library Example code at the bottom of this page.

Note: Some mikroElektronika's adapter boards have pinout different from T6369C datasheets. Appropriate relations between these labels are given in the table below:

Adapter Board	T6369C datasheet
RS	C/D
R/W	/RD
E	/WR

External dependencies of T6963C Graphic Lcd Library

The following variables must be defined in all projects using T6963C Graphic Lcd library:	Description:	Example :
`var T6963C_dataPort : byte; sfr; external;`	T6963C Data Port.	`var T6963C_dataPort : byte at PORTD;`
`var T6963C_ctrlPort : byte; sfr; external;`	T6963C Control Port.	`var T6963C_ctrlPort : byte at PORTC;`
`var T6963C_ctrlwr : sbit; sfr; external;`	Write signal.	`var T6963C_ctrlwr : sbit at PORTC.B2;`
`var T6963C_ctrlrd : sbit; sfr; external;`	Read signal.	`var T6963C_ctrlrd : sbit at PORTC.B1;`
`var T6963C_ctrlcd : sbit; sfr; external;`	Command/Data signal.	`var T6963C_ctrlcd : sbit at PORTC.B0;`
`var T6963C_ctrlrst : sbit; sfr; external;`	Reset signal.	`var T6963C_ctrlrst : sbit at PORTC.B4;`
`var T6963C_dataPort_Direction : byte; sfr; external;`	Direction of the T6963C Data Port.	`var T6963C_dataPort_Direction : byte at DDRD;`
`var T6963C_ctrlPort_Direction : byte; sfr; external;`	Direction of the T6963C Control Port.	`var T6963C_ctrlPort_Direction : byte at DDRC;`
`var T6963C_ctrlwr_Direction : sbit; sfr; external;`	Direction of the Write pin.	`var T6963C_ctrlwr_Direction : sbit at DDRC.B2;`
`var T6963C_ctrlrd_Direction : sbit; sfr; external;`	Direction of the Read pin.	`var T6963C_ctrlrd_Direction : sbit at DDRC.B1;`
`var T6963C_ctrlcd_Direction : sbit; sfr; external;`	Direction of the Command/Data pin.	`var T6963C_ctrlcd_Direction : sbit at DDRC.B0;`
`var T6963C_ctrlrst_Direction : sbit; sfr; external;`	Direction of the Reset pin.	`var T6963C_ctrlrst_Direction : sbit at DDRC.B4;`

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_DisplayGrPanel
- T6963C_DisplayTxtPanel
- T6963C_SetGrPanel
- T6963C_SetTxtPanel
- T6963C_PanelFill
- T6963C_GrFill
- T6963C_TxtFill
- T6963C_Cursor_Height
- T6963C_Graphics
- T6963C_Text
- T6963C_Cursor
- T6963C_Cursor_Blink

T6963C_Init

<table>
<tr><td><b>Prototype</b></td><td><code>procedure T6963C_init(width, height, fntW : byte);</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>

Initalizes the Graphic Lcd controller.

Parameters :

- width: width of the Glcd panel
- height: height of the Glcd panel
- fntW: font width

Display RAM organization:

The library cuts the RAM into panels : a complete panel is one graphics panel followed by a text panel (see schematic below).

```
schematic:
+---------------------+ /\
+ GRAPHICS PANEL #0    +  |
+                       +   |
+                       +   |
+                       +   |
+---------------------+  | PANEL 0
+ TEXT PANEL #0        +  |
+                       + \/
+---------------------+ /\
+ GRAPHICS PANEL #1    +  |
+                       +   |
+                       +   |
+                       +   |
+---------------------+  | PANEL 1
+ TEXT PANEL #2        +  |
+                       +   |
+---------------------+ \/
```

</td></tr>
</table>

Requires	Global variables : - T6963C_dataPort: Data Port - T6963C_ctrlPort: Control Port - T6963C_ctrlwr: Write signal pin - T6963C_ctrlrd: Read signal pin - T6963C_ctrlcd: Command/Data signal pin - T6963C_ctrlrst: Reset signal pin - T6963C_dataPort_Direction: Direction of Data Port - T6963C_ctrlPort_Direction: Direction of Control Port - T6963C_ctrlwr_Direction: Direction of Write signal pin - T6963C_ctrlrd_Direction: Direction of Read signal pin - T6963C_ctrlcd_Direction: Direction of Command/Data signal pin - T6963C_ctrlrst_Direction: Direction of Reset signal pin must be defined before using this function.

Example	`// T6963C module connections`
	`var T6963C_ctrlPort : byte at PORTC;`
	`var T6963C_dataPort : byte at PORTD;`
	`var T6963C_ctrlPort_Direction : byte at DDRC;`
	`var T6963C_dataPort_Direction : byte at DDRD;`
	
	`var T6963C_ctrlwr : sbit at PORTC.B2;`
	`var T6963C_ctrlrd : sbit at PORTC.B1;`
	`var T6963C_ctrlcd : sbit at PORTC.B0;`
	`var T6963C_ctrlrst : sbit at PORTC.B4;`
	`var T6963C_ctrlwr_Direction : sbit at DDRC.B2;`
	`var T6963C_ctrlrd_Direction : sbit at DDRC.B1;`
	`var T6963C_ctrlcd_Direction : sbit at DDRC.B0;`
	`var T6963C_ctrlrst_Direction : sbit at DDRC.B4;`
	`// End of T6963C module connections`
	
	`...`
	`// init display for 240 pixel width, 128 pixel height and 8 bits character width`
	`T6963C_init(240, 128, 8);`

T6963C_WriteData

Prototype	`procedure T6963C_WriteData(mydata : byte);`
Returns	Nothing.
Description	Writes data to T6963C controller. Parameters : - `mydata`: data to be written
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_WriteData(AddrL);`

T6963C_WriteCommand

Prototype	`procedure T6963C_WriteCommand(mydata : byte);`
Returns	Nothing.
Description	Writes command to T6963C controller. Parameters : - `mydata`: command to be written
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_WriteCommand(T6963C_CURSOR_POINTER_SET);`

T6963C_SetPtr

Prototype	`procedure T6963C_SetPtr(p : word; c : byte);`
Returns	Nothing.
Description	Sets the memory pointer p for command c. Parameters : - `p`: address where command should be written - `c`: command to be written
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`T6963C_SetPtr(T6963C_grHomeAddr + start,`
	`T6963C_ADDRESS_POINTER_SET);`

T6963C_WaitReady

Prototype	`procedure T6963C_WaitReady();`
Returns	Nothing.
Description	Pools the status byte, and loops until Toshiba Glcd module is ready.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_WaitReady();`

T6963C_Fill

Prototype	`procedure T6963C_Fill(v : byte; start, len : word);`
Returns	Nothing.
Description	Fills controller memory block with given byte. Parameters : - `v`: byte to be written - `start`: starting address of the memory block - `len`: length of the memory block in bytes
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Fill(0x33,0x00FF,0x000F);`

T6963C_Dot

Prototype	`procedure T6963C_Dot(x, y : integer; color : byte);`
Returns	Nothing.
Description	Draws a dot in the current graphic panel of Glcd at coordinates (x, y). Parameters : - `x`: dot position on x-axis - `y`: dot position on y-axis - `color`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Dot(x0, y0, pcolor);`

T6963C_Write_Char

Prototype	`procedure T6963C_Write_Char(c, x, y, mode : byte);`
Returns	Nothing.
Description	Writes a char in the current text panel of Glcd at coordinates (x, y). Parameters : - `c`: char to be written - `x`: char position on x-axis - `y`: char position on y-axis - `mode`: mode parameter. Valid values: T6963C_ROM_MODE_OR, T6963C_ROM_MODE_XOR, T6963C_ROM_MODE_AND and T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the log ical “exclusive OR”. This can be useful to display text in the negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
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);`
Returns	Nothing.
Description	Writes text in the current text panel of Glcd at coordinates (x, y). Parameters : - `str`: text to be written - `x`: text position on x-axis - `y`: text position on y-axis - `mode`: mode parameter. Valid values: T6963C_ROM_MODE_OR, T6963C_ROM_MODE_XOR, T6963C_ROM_MODE_AND and T6963C_ROM_MODE_TEXT Mode parameter explanation: - OR Mode: In the OR-Mode, text and graphics can be displayed and the data is logically “OR-ed”. This is the most common way of combining text and graphics for example labels on buttons. - XOR-Mode: In this mode, the text and graphics data are combined via the log ical “exclusive OR”. This can be useful to display text in the negative mode, i.e. white text on black background. - AND-Mode: The text and graphic data shown on display are combined via the logical “AND function”. - TEXT-Mode: This option is only available when displaying just a text. The Text Attribute values are stored in the graphic area of display memory. For more details see the T6963C datasheet.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
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);`
Returns	Nothing.
Description	Draws a line from (x0, y0) to (x1, y1). Parameters : - `x0`: x coordinate of the line start - `y0`: y coordinate of the line end - `x1`: x coordinate of the line start - `y1`: y coordinate of the line end - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Line(0, 0, 239, 127, T6963C_WHITE);`

T6963C_Rectangle

Prototype	`procedure T6963C_Rectangle(x0, y0, x1, y1 : integer; pcolor : byte);`
Returns	Nothing.
Description	Draws a rectangle on Glcd. Parameters : - `x0`: x coordinate of the upper left rectangle corner - `y0`: y coordinate of the upper left rectangle corner - `x1`: x coordinate of the lower right rectangle corner - `y1`: y coordinate of the lower right rectangle corner - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Rectangle(20, 20, 219, 107, T6963C_WHITE);`

T6963C_Box

Prototype	`procedure T6963C_Box(x0, y0, x1, y1 : integer; pcolor : byte);`
Returns	Nothing.
Description	Draws a box on Glcd Parameters : - `x0`: x coordinate of the upper left box corner - `y0`: y coordinate of the upper left box corner - `x1`: x coordinate of the lower right box corner - `y1`: y coordinate of the lower right box corner - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Box(0, 119, 239, 127, T6963C_WHITE);`

T6963C_Circle

Prototype	`procedure T6963C_Circle(x, y : integer; r : longint; pcolor : word);`
Returns	Nothing.
Description	Draws a circle on Glcd. Parameters : - `x`: x coordinate of the circle center - `y`: y coordinate of the circle center - `r`: radius size - `pcolor`: color parameter. Valid values: T6963C_BLACK and T6963C_WHITE
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Circle(120, 64, 110, T6963C_WHITE);`

T6963C_Image

Prototype	`procedure T6963C_Image(const code pic : ^byte);`
Returns	Nothing.
Description	Displays bitmap on Glcd. Parameters : - `pic`: image to be displayed. Bitmap array can be located in both code and RAM memory (due to the mikroPascal PRO for AVR pointer to const and pointer to RAM equivalency). Use the mikroPascal's integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Image(mc);`

T6963C_Sprite

Prototype	`procedure T6963C_Sprite(px, py : byte; const pic : ^byte; sx, sy : byte);`
Returns	Nothing.
Description	Fills graphic rectangle area (px, py) to (px+sx, py+sy) with custom size picture. Parameters : - `px`: x coordinate of the upper left picture corner. Valid values: multiples of the font width - `py`: y coordinate of the upper left picture corner - `pic`: picture to be displayed - `sx`: picture width. Valid values: multiples of the font width - `sy`: picture height **Note:** If px and sx parameters are not multiples of the font width they will be scaled to the nearest lower number that is a multiple of the font width.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Sprite(76, 4, einstein, 88, 119); // draw a sprite`

T6963C_Set_Cursor

Prototype	`procedure T6963C_Set_Cursor(x, y : byte);`
Returns	Nothing.
Description	Sets cursor to row x and column y. Parameters : - `x`: cursor position row number - `y`: cursor position column number
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Set_Cursor(cposx, cposy);`

T6963C_DisplayGrPanel

Prototype	`procedure T6963C_DisplayGrPanel(n : byte);`
Returns	Nothing.
Description	Display selected graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// display graphic panel 1`
	`T6963C_DisplayGrPanel(1);`

T6963C_DisplayTxtPanel

Prototype	`procedure T6963C_DisplayTxtPanel(n : byte);`
Returns	Nothing.
Description	Display selected text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// display text panel 1`
	`T6963C_DisplayTxtPanel(1);`

T6963C_SetGrPanel

Prototype	`procedure T6963C_SetGrPanel(n : byte);`
Returns	Nothing.
Description	Compute start address for selected graphic panel and set appropriate internal pointers. All subsequent graphic operations will be preformed at this graphic panel. Parameters : - `n`: graphic panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// set graphic panel 1 as current graphic panel.`
	`T6963C_SetGrPanel(1);`

T6963C_SetTxtPanel

Prototype	`procedure T6963C_SetTxtPanel(n : byte);`
Returns	Nothing.
Description	Compute start address for selected text panel and set appropriate internal pointers. All subsequent text operations will be preformed at this text panel. Parameters : - `n`: text panel number. Valid values: 0 and 1.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// set text panel 1 as current text panel.`
	`T6963C_SetTxtPanel(1);`

T6963C_PanelFill

Prototype	`procedure T6963C_PanelFill(v : byte);`
Returns	Nothing.
Description	Fill current panel in full (graphic+text) with appropriate value (0 to clear). Parameters : - `v`: value to fill panel with.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`clear current panel`
	`T6963C_PanelFill(0);`

T6963C_GrFill

Prototype	`procedure T6963C_GrFill(v : byte);`
Returns	Nothing.
Description	Fill current graphic panel with appropriate value (0 to clear). Parameters : - `v`: value to fill graphic panel with.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// clear current graphic panel`
	`T6963C_GrFill(0);`

T6963C_TxtFill

Prototype	`procedure T6963C_TxtFill(v : byte);`
Returns	Nothing.
Description	Fill current text panel with appropriate value (0 to clear). Parameters : - `v`: this value increased by 32 will be used to fill text panel.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// clear current text panel`
	`T6963C_TxtFill(0);`

T6963C_Cursor_Height

Prototype	`procedure T6963C_Cursor_Height(n : byte);`
Returns	Nothing.
Description	Set cursor size. Parameters : - `n`: cursor height. Valid values: 0..7.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.
Example	`T6963C_Cursor_Height(7);`

T6963C_Graphics

Prototype	`procedure T6963C_Graphics(n : byte);`
Returns	Nothing.
Description	Enable/disable graphic displaying. Parameters : - `n`: on/off parameter. Valid values: 0 (disable graphic dispaying) and 1 (enable graphic displaying).
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// enable graphic displaying`
	`T6963C_Graphics(1);`

T6963C_Text

Prototype	`procedure T6963C_Text(n : byte);`
Returns	Nothing.
Description	Enable/disable text displaying. Parameters : - `n`: on/off parameter. Valid values: 0 (disable text dispaying) and 1 (enable text displaying).
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// enable text displaying`
	`T6963C_Text(1);`

T6963C_Cursor

Prototype	`procedure T6963C_Cursor(n : byte);`
Returns	Nothing.
Description	Set cursor on/off. Parameters : - `n`: on/off parameter. Valid values: 0 (set cursor off) and 1 (set cursor on).
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// set cursor on`
	`T6963C_Cursor(1);`

T6963C_Cursor_Blink

Prototype	`procedure T6963C_Cursor_Blink(n : byte);`
Returns	Nothing.
Description	Enable/disable cursor blinking. Parameters : - `n`: on/off parameter. Valid values: 0 (disable cursor blinking) and 1 (enable cursor blinking).
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_Init routine.

Example	`// enable cursor blinking`
	`T6963C_Cursor_Blink(1);`

Library Example

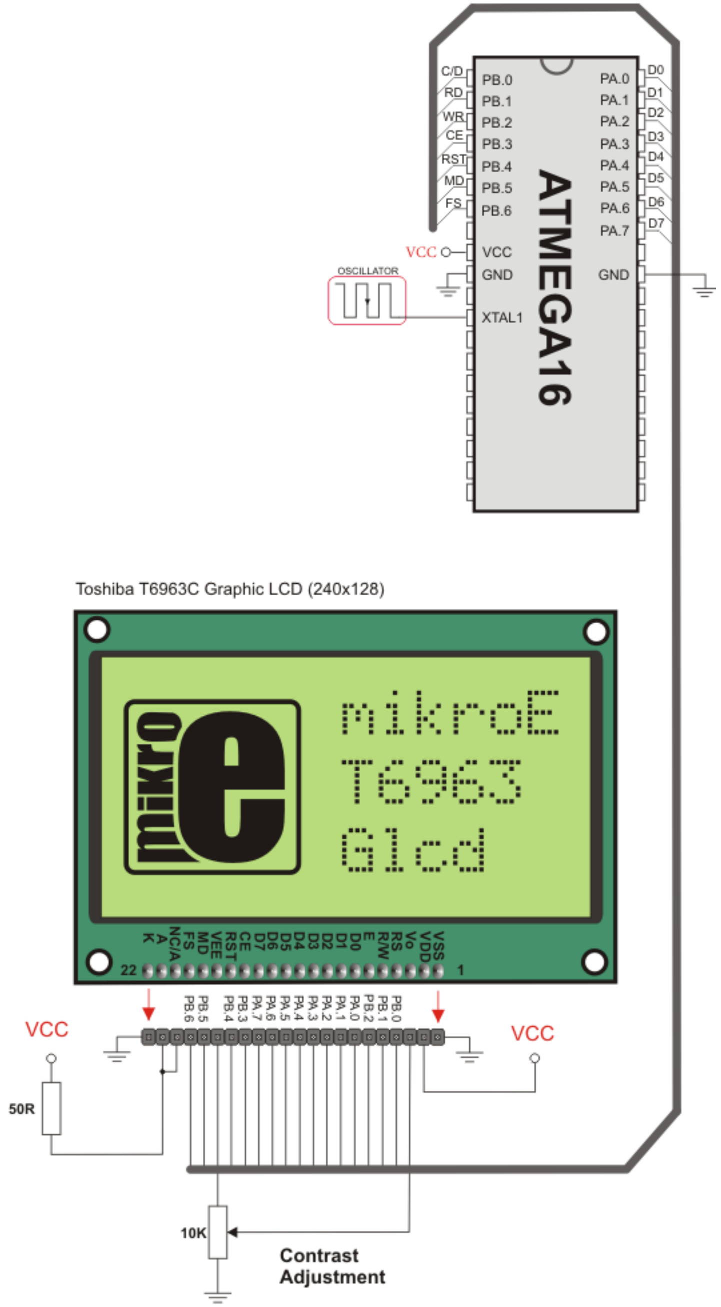
The following drawing demo tests advanced routines of the T6963C Glcd library. Hardware configurations in this example are made for the T6963C 240x128 display, EasyAVR5A board and ATmega16.

```
program T6963C_240x128;

uses __Lib_T6963C_Consts, __Lib_T6963C, bitmap, bitmap2;

// T6963C module connections
var T6963C_ctrlPort : byte at PORTC;              // CONTROL port
var T6963C_dataPort : byte at PORTD;              // DATA port
var T6963C_ctrlPort_Direction : byte at DDRC;     // CONTROL direction register
var T6963C_dataPort_Direction : byte at DDRD;     // DATA direction register

var T6963C_ctrlwr : sbit at PORTC.B2;             // WR write signal
var T6963C_ctrlrd : sbit at PORTC.B1;             // RD read signal
var T6963C_ctrlcd : sbit at PORTC.B0;             // CD command/data signal
var T6963C_ctrlrst : sbit at PORTC.B4;            // RST reset signal
var T6963C_ctrlwr_Direction : sbit at DDRC.B2;    // WR write signal direction
var T6963C_ctrlrd_Direction : sbit at DDRC.B1;    // RD read signal direction
var T6963C_ctrlcd_Direction : sbit at DDRC.B0;    // CD command/data signal
direction
var T6963C_ctrlrst_Direction : sbit at DDRC.B4;   // RST reset signal direction

// Signals not used by library, they are set in main function
var T6963C_ctrlce : sbit at PORTC.B3;                    // CE signal
var T6963C_ctrlfs : sbit at PORTC.B6;                    // FS signal
var T6963C_ctrlmd : sbit at PORTC.B5;                    // MD signal
var T6963C_ctrlce_Direction : sbit at DDRC.B3;           // CE signal direction
           var T6963C_ctrlfs_Direction : sbit at DDRC.B6;// FS signal direction
           var T6963C_ctrlmd_Direction : sbit at DDRC.B5;// MD signal direction
           // End T6963C module connections

           var   panel : byte;           // current panel
                      i : word;          // general purpose register
                   curs : byte;          // cursor visibility
                  cposx,
                  cposy : word;          // cursor x-y position
                  txtcols : byte;        // number of text coloms
                  txt, txt1 : string[ 29] ;

           begin

             txt1 := ' EINSTEIN WOULD HAVE LIKED mE';
             txt  := ' GLCD LIBRARY DEMO, WELCOME !';

             DDRA := 0x00;                           // configure PORTA as input

             DDB0 := 0;                             // Set PB0 as input
             DDB1 := 0;                             // Set PB1 as input
             DDB2 := 0;                             // Set PB2 as input
             DDB3 := 0;                             // Set PB3 as input
             DDB4 := 0;                             // Set PB4 as input

             T6963C_ctrlce_Direction := 1;
             T6963C_ctrlce := 0;                   // Enable T6963C
             T6963C_ctrlfs_Direction := 1;
             T6963C_ctrlfs := 0;                   // Font Select 8x8
             T6963C_ctrlmd_Direction := 1;
             T6963C_ctrlmd := 0;                   // Column number select

             panel := 0;
             i := 0;
             curs := 0;
             cposx := 0;
             cposy := 0;

             // Initialize T6369C
             T6963C_init(240, 128, 8);

             {*
              * Enable both graphics and text display at the same time
              *}
             T6963C_graphics(1);
             T6963C_text(1);

           {*
              * Text messages
            *}
             T6963C_write_text(txt, 0, 0, T6963C_ROM_MODE_XOR);
             T6963C_write_text(txt1, 0, 15, T6963C_ROM_MODE_XOR);

             {*
              * Cursor
              *}
             T6963C_cursor_height(8);       // 8 pixel height
             T6963C_set_cursor(0, 0);       // Move cursor to top left
             T6963C_cursor(0);              // Cursor off

             {*
              * Draw rectangles
              *}
             T6963C_rectangle(0, 0, 239, 127, T6963C_WHITE);
             T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE);
             T6963C_rectangle(40, 40, 199, 87, T6963C_WHITE);
             T6963C_rectangle(60, 60, 179, 67, T6963C_WHITE);

             {*
              * Draw a cross
              *}
             T6963C_line(0, 0, 239, 127, T6963C_WHITE);
             T6963C_line(0, 127, 239, 0, T6963C_WHITE);

             {*
              * Draw solid boxes
              *}
             T6963C_box(0, 0, 239, 8, T6963C_WHITE);
             T6963C_box(0, 119, 239, 127, T6963C_WHITE);
              //while true do nop;
             {*
              * Draw circles
              *}
           T6963C_circle(120, 64, 10, T6963C_WHITE);
             T6963C_circle(120, 64, 30, T6963C_WHITE);
             T6963C_circle(120, 64, 50, T6963C_WHITE);
             T6963C_circle(120, 64, 70, T6963C_WHITE);
             T6963C_circle(120, 64, 90, T6963C_WHITE);
             T6963C_circle(120, 64, 110, T6963C_WHITE);
             T6963C_circle(120, 64, 130, T6963C_WHITE);

           T6963C_sprite(76, 4, @einstein, 88, 119);      // Draw a sprite

             T6963C_setGrPanel(1);                 // Select other graphic panel

             T6963C_image(@me);

           while (TRUE) do                               // Endless loop
              begin
```

```
{*
       * If PORTA_0 is pressed, toggle the display between graphic
panel 0 and graphic 1
       *}
      if( PINA0_bit = 0) then
        begin
          Inc(panel) ;
          panel := panel and 1;
            T6963C_setPtr((T6963C_grMemSize + T6963C_txtMemSize) *
panel, T6963C_GRAPHIC_HOME_ADDRESS_SET) ;
          Delay_ms(300) ;
        end

      {*
       * If PORTA_1 is pressed, display only graphic panel
       *}
      else
        if ( PINA1_bit = 0) then
          begin
            T6963C_graphics(1) ;
            T6963C_text(0) ;
            Delay_ms(300) ;
          end

      {*
       * If PORTA_2 is pressed, display only text panel
       *}
        else
          if ( PINA2_bit = 0) then
            begin
              T6963C_graphics(0) ;
              T6963C_text(1) ;
              Delay_ms(300) ;
            end

{*
       * If PORTA_3 is pressed, display text and graphic panels
       *}
          else
            if ( PINA3_bit = 0) then
              begin
                T6963C_graphics(1) ;
                T6963C_text(1) ;
                Delay_ms(300) ;
              end

      {*
       * If PORTA_4 is pressed, change cursor
       *}
```

```
 else
                if( PINA4_bit = 0) then
                  begin
                    Inc(curs);
                    if (curs = 3) then
                      curs := 0;
                    case curs of
                      0:
                         // no cursor
                         T6963C_cursor(0) ;

                      1: begin
                           // blinking cursor
                           T6963C_cursor(1) ;
                           T6963C_cursor_blink(1) ;
                         end;
                      2: begin
                           // non blinking cursor
                           T6963C_cursor(1) ;
                           T6963C_cursor_blink(0) ;
                         end;
                    end;
                    Delay_ms(300) ;
                  end;

       {*
        * Move cursor, even if not visible
        *}
       Inc(cposx);
       if (cposx = T6963C_txtCols) then
         begin
           cposx := 0 ;
           Inc(cposy);
          if (cposy = T6963C_grHeight / T6963C_CHARACTER_HEIGHT) then
             cposy := 0 ;
         end;
       T6963C_set_cursor(cposx, cposy) ;

       Delay_ms(100) ;
     end;
end.
```

HW Connection



T6963C Glcd HW connection

TWI LIBRARY

TWI full master MSSP module is available with a number of AVR MCU models. mikroPascal PRO for AVR provides library which supports the master TWI mode.

Library Routines

- TWI_Init
- TWI_Busy
- TWI_Start
- TWI_Stop
- TWI_Read
- TWI_Write
- TWI_Status
- TWI_Close

TWI_Init

Prototype	`procedure TWI_Init(clock : dword);`
Returns	Nothing.
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. You don't need to configure ports manually for using the module; library will take care of the initialization.
Requires	Library requires MSSP module on PORTB or PORTC.
Example	`TWI_Init(100000);`

TWI_Busy

Prototype	`function TWI_Busy() : byte;`
Returns	Returns 0 if TWI start sequnce is finished, 1 if TWI start sequnce is not finished.
Description	Signalizes the status of TWI bus.
Requires	TWI must be configured before using this function. See TWI_Init.

Example	`if (TWI_Busy = 1)`
	`  begin`
	`    ...`

TWI_Start

Prototype	`function TWI_Start() : char;`
Returns	If there is no error function returns 0, otherwise returns 1.
Description	Determines if TWI bus is free and issues START signal.
Requires	TWI must be configured before using this function. See TWI_Init.

Example	`if (TWI_Start = 1)`
	`  begin`
	`    ...`

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	TWI must be configured before using this function. See TWI_Init. Also, START signal needs to be issued in order to use this function. See TWI_Start.
Example	Read data and send not *acknowledge* signal: `tmp := TWI_Read(0);`

TWI_Write

Prototype	`procedure TWI_Write(data_ : byte);`
Returns	Nothing.
Description	Sends data byte (parameter data_) via TWI bus.
Requires	TWI must be configured before using this function. See TWI_Init. Also, START signal needs to be issued in order to use this function. See TWI_Start.
Example	`TWI_Write(0xA3);`

TWI_Stop

Prototype	`procedure TWI_Stop();`
Returns	Nothing.
Description	Issues STOP signal to TWI operation.
Requires	TWI must be configured before using this function. See TWI_Init.
Example	`TWI_Stop();`

TWI_Status

Prototype	`function TWI_Status() : byte;`
Returns	Returns value of status register (TWSR), the highest 5 bits.
Description	Returns status of TWI.
Requires	TWI must be configured before using this function. See TWI_Init.
Example	`status := TWI_Status();`

TWI_Close

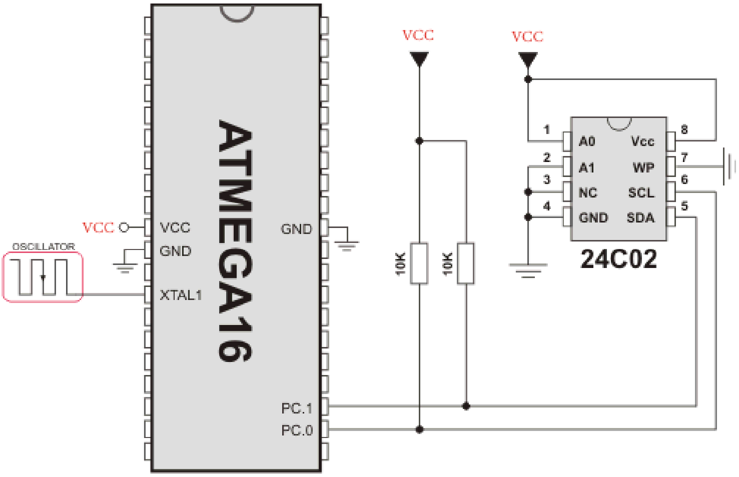
Prototype	`procedure TWI_Close();`
Returns	Nothing.
Description	Closes TWI connection.
Requires	TWI must be configured before using this function. See TWI_Init.
Example	`TWI_Close();`

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 PORTA, to check if the cycle was successful. Check the figure below.

```
program TWI_Simple;

begin
  DDRA := 0xFF;             // configure PORTA as output

  TWI_Init(100000);         // initialize TWI communication
  TWI_Start();              // issue TWI start signal
  TWI_Write(0xA2);          // send byte via TWI (device address + W)
  TWI_Write(2);             // send byte (address of EEPROM location)
  TWI_Write(0xAA);          // send data (data to be written)
  TWI_Stop();               // issue TWI stop signal

  Delay_100ms();

  TWI_Start();              // issue TWI start signal
  TWI_Write(0xA2);          // send byte via TWI (device address + W)
  TWI_Write(2);             // send byte (data address)
  TWI_Start();              // issue TWI signal repeated start
  TWI_Write(0xA3);          // send byte (device address + R)
  PORTA := TWI_Read(0);     // read data (NO acknowledge)
  TWI_Stop();               // issue TWI stop signal}
end.
```

HW Connection



Interfacing 24c02 to AVR via TWI

UART LIBRARY

UART hardware module is available with a number of AVR MCUs. mikroPascal PRO for AVR UART Library provides comfortable work with the Asynchronous (full duplex) mode.

You can easily communicate with other devices via RS-232 protocol (for example with PC, see the figure at the end of the topic – RS-232 HW connection). You need a AVR MCU with hardware integrated UART, for example ATmega16. Then, simply use the functions listed below.

Library Routines

- UARTx_Init
- UARTx_Init_Advanced
- UARTx_Data_Ready
- UARTx_Read
- UARTx_Read_Text
- UARTx_Write
- UARTx_Write_Text

The following routine is for the internal use by compiler only:

- `UARTx_TX_Idle`

Note: AVR MCUs require you to specify the module you want to use. To select the desired UART, simply change the letter x in the prototype for a number from 1 to 4. Number of UART modules per MCU differs from chip to chip. Please, read the appropriate datasheet before utilizing this library.

Example: `UART2_Init();` initializes UART 2 module.

Note: Some of the AVR MCUs do not support UARTx_Init_Advanced routine. Please, refer to the appropriate datasheet.

UARTx_Init

Prototype	`procedure UARTx_Init(baud_rate: longint);`
Returns	Nothing.
Description	Configures and initializes the UART module. The internal UART module module is set to: - receiver enabled - transmitter enabled - frame size 8 bits - 1 STOP bit - parity mode disabled - asynchronous operation Parameters : - `baud_rate`: requested baud rate Refer to the device data sheet for baud rates allowed for specific Fosc.
Requires	You'll need AVR MCU with hardware UART. UARTx_Init needs to be called before using other functions from UART Library.
Example	This will initialize hardware UART1 module and establish the communication at 2400 bps: `UART1_Init(2400);`

UARTx_Init_Advanced

<table>
<tr><td><b>Prototype</b></td><td><code>procedure UARTx_Init_Advanced(baud_rate : dword; parity : byte; byte);</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>Configures and initializes UART module.

Parameter baud_rate configures UART module to work on a requested baud rate. Parameters parity and stop_bits determine the work mode for UART, and can have the following values:
<table>
<tr><th>Mask</th><th>Description</th><th>Predefined library const</th></tr>
<tr><th colspan="3">Parity constants:</th></tr>
<tr><td>0x00</td><td>Parity mode disabled</td><td>_UART_NOPARITY</td></tr>
<tr><td>0x20</td><td>Even parity</td><td>_UART_EVENPARITY</td></tr>
<tr><td>0x30</td><td>Odd parity</td><td>_UART_ODDPARITY</td></tr>
<tr><th colspan="3">Stop bit constants:</th></tr>
<tr><td>0x00</td><td>1 stop bit</td><td>_UART_ONE_STOPBIT</td></tr>
<tr><td>0x01</td><td>2 stop bits</td><td>_UART_TWO_STOPBITS</td></tr>
</table>
<b>Note:</b> Some MCUs do not support advanced configuration of the UART module. Please consult appropriate daatsheet.</td></tr>
<tr><td><b>Requires</b></td><td>MCU must have UART module.</td></tr>
<tr><td><b>Example</b></td><td><code>// Initialize hardware UART1 module and establish communication at 9600 bps, 8-bit data, even parity and 2 STOP bits
UART1_Init_Advanced(9600, _UART_EVENPARITY, _UART_TWO_STOPBITS);</code></td></tr>
</table>

UARTx_Data_Ready

Prototype	`function UARTx_Data_Ready(): byte;`
Returns	Function returns 1 if data is ready or 0 if there is no data.
Description	The function tests if data in receive buffer is ready for reading.
Requires	MCU with the UART module. The UART module must be initialized before using this routine. See the UARTx_Init routine.

Example	`var receive: byte;`
	`...`
	`// read data if ready`
	`if (UART1_Data_Ready() = 1) then`
	`   receive := UART1_Read();`

UARTx_Read

Prototype	`function UARTx_Read(): byte;`
Returns	Received byte.
Description	The function receives a byte via UART. Use the Uart_Data_Ready function to test if data is ready first.
Requires	MCU with the UART module. The UART module must be initialized before using this routine. See UARTx_Init routine.

Example	`var receive: byte;`
	`...`
	`// read data if ready`
	`if (UART1_Data_Ready() = 1) then`
	`   receive := UART1_Read();`

UARTx_Read_Text

Prototype	`procedure UARTx_Read_Text(var Output : string[255]; var Delimiter : sting[10]; Attempts : byte);`
Returns	Nothing.
Description	Reads characters received via UART until the delimiter sequence is detected. The read sequence is stored in the parameter output; delimiter sequence is stored in the parameter `delimiter.` This is a blocking call: the delimiter sequence is expected, otherwise the procedure exits(if the delimiter is not found). Parameter Attempts defines number of received characters in which Delimiter sequence is expected. If Attempts is set to 255, this routine will continuously try to detect the Delimiter sequence.
Requires	UART HW module must be initialized and communication established before using this function. See UARTx_Init.

Example	Read text until the sequence “OK” is received, and send back what’s been received:
	`UART1_Init(4800);                        // initialize UART module`
	`Delay_ms(100);`
	
	` while TRUE do`
	`   begin`
	`     if (UART1_Data_Ready() = 1)          // if data is received`
	`       begin`
	`         UART1_Read_Text(output, 'delim', 10); // reads text`
	`until 'delim' is found`
	`         UART1_Write_Text(output);            // sends back text`
	`       end;`
	`   end.`

UARTx_Write

<table>
<tr><td><b>Prototype</b></td><td><code>procedure UARTx_Write(TxData: byte);</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>The function transmits a byte via the UART module.

Parameters :

- <code>TxData</code>: data to be sent</td></tr>
<tr><td><b>Requires</b></td><td>MCU with the UART module.

The UART module must be initialized before using this routine. See UARTx_Init routine.</td></tr>
<tr><td><b>Example</b></td><td><pre>var data_: byte;
...
data := 0x1E
UART1_Write(data_);</pre></td></tr>
</table>

UARTx_Write_Text

<table>
<tr><td><b>Prototype</b></td><td><code>procedure UARTx_Write_Text(var uart_text : string[255]);</code></td></tr>
<tr><td><b>Returns</b></td><td>Nothing.</td></tr>
<tr><td><b>Description</b></td><td>Sends text (parameter <code>uart_text</code>) via UART. Text should be zero terminated.</td></tr>
<tr><td><b>Requires</b></td><td>UART HW module must be initialized and communication established before using this function. See UARTx_Init.</td></tr>
<tr><td><b>Example</b></td><td>Read text until the sequence “OK” is received, and send back what’s been received:<pre>UART1_Init(4800);   // initialize UART module
Delay_ms(100);

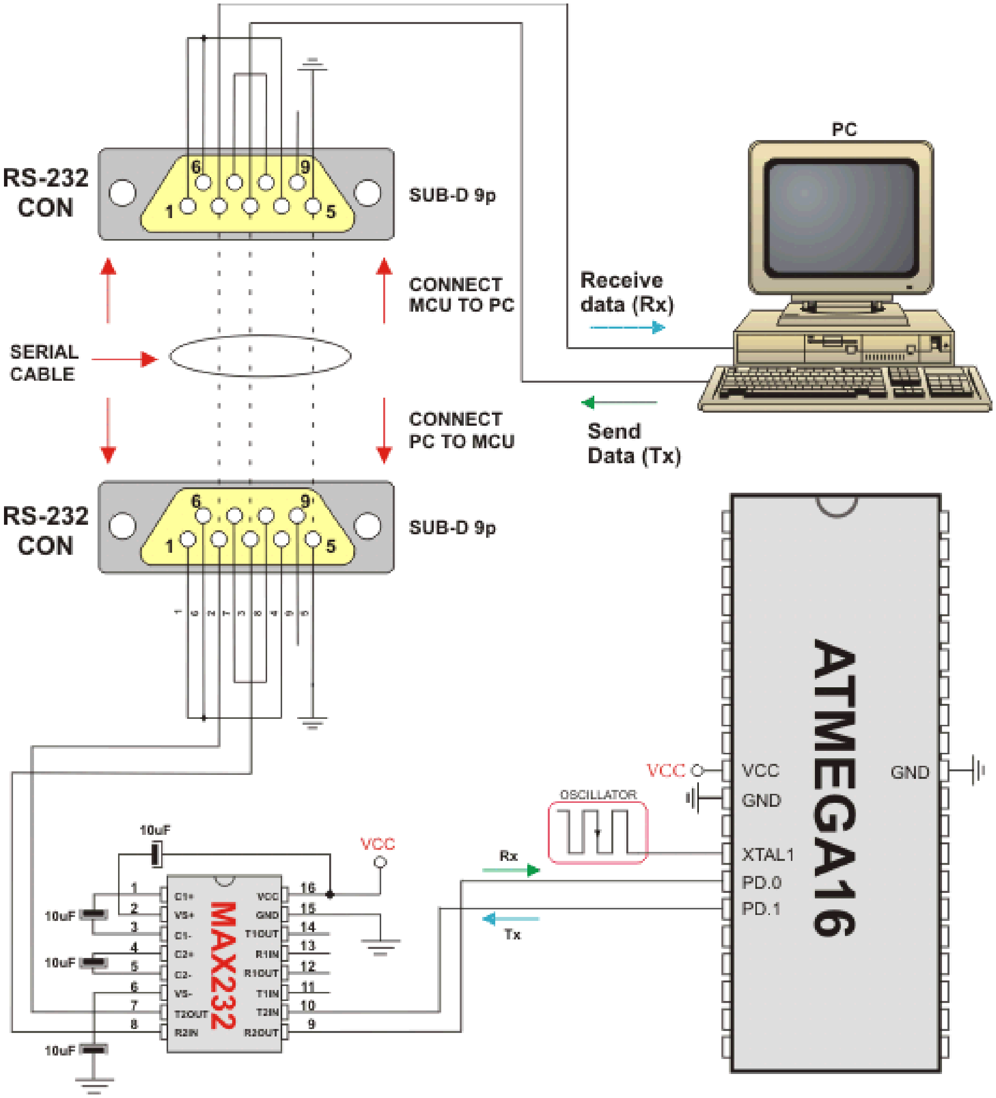
 while TRUE do
   begin
     if (UART1_Data_Ready() = 1)   // if data is received
       begin
         UART1_Read_Text(output, 'delim', 10); // reads text
until 'delim' is found
         UART1_Write_Text(output);   // sends back text
       end;
   end.</pre></td></tr>
</table>

Library Example

This example demonstrates simple data exchange via UART. If MCU is connected to the PC, you can test the example from the mikroPascal PRO for AVR USART Terminal.

```
program UART;
var uart_rd : byte;

begin
  UART1_Init(9600);            // Initialize UART module at 9600 bps
  Delay_ms(100);                // Wait for UART module to stabilize

  while (TRUE) do                            // Endless loop
    begin
       if (UART1_Data_Ready() <> 0) then     // If data is received,
         begin
          uart_rd := UART1_Read();      //   read the received data,
           UART1_Write(uart_rd);        //   and send data via UART
         end;
    end;
end.
```

HW Connection



UART HW connection

BUTTON LIBRARY

The Button library contains miscellaneous routines useful for a project development.

External dependencies of Button Library

The following variable must be defined in all projects using Button library:	Description:	Example :
`var Button_Pin : sbit; sfr; external;`	Declares button pins.	`var Button_Pin : sbit at PINB.B0;`
`var Button_Pin_Direction : sbit; sfr; external;`	Declares direction of the button pin.	`var Button_Pin_Direction : sbit at DDRB.B0;`

Library Routines

- Button

Button

Prototype	`function Button(time_ms : byte; active_state : byte) : byte;`
Returns	- `255` if the pin was in the active state for given period. - `0` otherwise
Description	The function eliminates the influence of contact flickering upon pressing a button (debouncing). The Button pin is tested just after the function call and then again after the debouncing period has expired. If the pin was in the active state in both cases then the function returns 255 (true). Parameters : - `time_ms` : debouncing period in milliseconds - `active_state`: determines what is considered as active state. Valid values: 0 (logical zero) and 1 (logical one)
Requires	Global variables : - `Button_Pin`: Button pin line - `Button_Pin_Direction`: Direction of the button pin must be defined before using this function.

Example

On every PORTB0 one-to-zero transition PORTC is inverted :

```
program Button_Test;

// Button connections
var Button_Pin : sbit at PINB.B0;    // Input pin, PINx register
is used
var Button_Pin_Direction : sbit at DDRB.B0;
// End Button connections

var oldstate : bit;

begin
  Button_Pin_Direction := 0;     // Set Button pin as input

  DDRC  := 0xFF;                 // Configure PORTC as output
  PORTC := 0xAA;                // Initial PORTC value

  oldstate := 0;                // oldstate initial value

  while TRUE do
    begin
      if (Button(1, 1) <> 0) then          // Detect logical one
        oldstate := 1;                     // Update flag

      if (oldstate and Button(1, 0)) then // Detect one-to-zero
transition
        begin
          PORTC := not PORTC;           // Invert PORTC
          oldstate := 0;                // Update flag
        end;
    end;                                // Endless loop
end.
```

Conversions Library

mikroPascal PRO for AVR Conversions Library provides routines for numerals to strings and 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
- LongWordToStr
- FloatToStr

The following functions convert decimal values to BCD and vice versa:

- Dec2Bcd
- Bcd2Dec16
- Dec2Bcd16

ByteToStr

Prototype	`procedure ByteToStr(input : word; var output : array[3] of char)`
Returns	Nothing.
Description	Converts input byte to a string. The output string is right justified and remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: byte to be converted - `output`: destination string
Requires	Nothing.

Example	`var t : word;`
	`    txt : array[3] of char;`
	`...`
	`t := 24;`
	`ByteToStr(t, txt);  // txt is " 24" (one blank here)`

ShortToStr

Prototype	`procedure ShortToStr(input : short; var output : array[ 4] of char);`
Returns	Nothing.
Description	Converts input short (signed byte) number to a string. The output string is right justified and remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: short number to be converted - `output`: destination string
Requires	Nothing.

Example	`var t : short;`
	`    txt : array[ 4] of char;`
	`...`
	`t := -24;`
	`ByteToStr(t, txt);  // txt is " -24" (one blank here)`

WordToStr

Prototype	`procedure WordToStr(input : word; var output : array[ 5] of char)`
Returns	Nothing.
Description	Converts input word to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: word to be converted - `output`: destination string
Requires	Nothing.

Example	`var t : word;`
	`    array[ 5] of char;`
	`...`
	`t := 437;`
	`WordToStr(t, txt);  // txt is "  437" (two blanks here)`

IntToStr

Prototype	`procedure IntToStr(input : integer; var output : array[6] of char);`
Returns	Nothing.
Description	Converts input integer number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: integer number to be converted - `output`: destination string
Requires	Nothing.

Example	`var input : integer;`
	`    txt : string[5];`
	`//...`
	`begin`
	`input := -4220;`
	`IntToStr(input, txt);   // txt is ' -4220'`

LongintToStr

Prototype	`procedure LongintToStr(input : longint; var output : array[11] of char);`
Returns	Nothing.
Description	Converts input longint number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: longint number to be converted - `output`: destination string
Requires	Nothing.

Example	`var input : longint;`
	`    txt : array[11] of char;`
	`//...`
	`begin`
	`input := -12345678;`
	`IntToStr(input, txt);    // txt is '  -12345678'`

LongWordToStr

Prototype	`procedure LongWordToStr(input : dword; var output : array[10] of char);`
Returns	Nothing.
Description	Converts input double word number to a string. The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters : - `input`: double word number to be converted - `output`: destination string
Requires	Nothing.

Example	`var input : longint;`
	`    txt : array[10] of char;`
	`//...`
	`begin`
	`input := 12345678;`
	`IntToStr(input, txt);    // txt is '  12345678'`

FloatToStr

Prototype	`procedure FloatToStr(input : real; var output : array[23] of char);`
Returns	- `3` if input number is NaN - `2` if input number is -INF - `1` if input number is +INF - `0` if conversion was successful
Description	Converts a floating point number to a string. Parameters : - `input`: floating point number to be converted - `output`: destination string The output string is left justified and null terminated after the last digit. **Note:** Given floating point number will be truncated to 7 most significant digits before conversion.
Requires	Nothing.

Example	`var ff1, ff2, ff3 : real;`
	`    txt : array[23] of char;`
	`  ...`
	`  ff1 := -374.2;`
	`  ff2 := 123.456789;`
	`  ff3 := 0.000001234;`
	
	`  FloatToStr(ff1, txt);   // txt is "-374.2"`
	`  FloatToStr(ff2, txt);   // txt is "123.4567"`
	`  FloatToStr(ff3, txt);   // txt is "1.234e-6"`

Dec2Bcd

Prototype	`function Dec2Bcd(decnum : byte) : byte;`
Returns	Converted BCD value.
Description	Converts input number to its appropriate BCD representation. Parameters : - `decnum`: number to be converted
Requires	Nothing.

Example	`var a, b : byte;`
	`...`
	`a := 22;`
	`b := Dec2Bcd(a); // b equals 34`

Bcd2Dec16

Prototype	`function Bcd2Dec16(bcdnum : word) : word;`
Returns	Converted decimal value.
Description	Converts 16-bit BCD numeral to its decimal equivalent. Parameters : - `bcdnum`: 16-bit BCD numeral to be converted
Requires	Nothing.

Example	`var a, b : word;`
	`...`
	`a := 0x1234;          // a equals 4660`
	`b := Bcd2Dec16(a);   // b equals 1234`

Dec2Bcd16

Prototype	`function Dec2Bcd16(decnum : word) : word;`
Returns	Converted BCD value.
Description	Converts decimal value to its BCD equivalent. Parameters : - `decnum` decimal number to be converted
Requires	Nothing.

Example	`var a, b : word;`
	`...`
	`a := 2345;`
	`b := Dec2Bcd16(a);   // b equals 9029`

MATH LIBRARY

The mikroPascal PRO for AVR provides a set of library functions for floating point math handling. See also Predefined Globals and Constants for the list of predefined math constants.

Library Functions

- acos
- asin
- atan
- atan2
- ceil
- cos
- cosh
- eval_poly
- exp
- fabs
- floor
- frexp
- ldexp
- log
- log10
- modf
- pow
- sin
- sinh
- sqrt
- tan
- tanh

acos

Prototype	`function acos(x : real) : real;`
Description	The function returns the arc cosine of parameter x; that is, the value whose cosine is x. The input parameter x must be between -1 and 1 (inclusive). The return value is in radians, between 0 and đ (inclusive).

asin

Prototype	`function asin(x : real) : real;`
Description	The function returns the arc sine of parameter x; that is, the value whose sine is x. The input parameter x must be between -1 and 1 (inclusive). The return value is in radians, between -đ/2 and đ/2 (inclusive).

atan

Prototype	`function atan(arg : real) : real;`
Description	The function computes the arc tangent of parameter arg; that is, the value whose tangent is arg. The return value is in radians, between -đ/2 and đ/2 (inclusive).

atan2

Prototype	`function atan2(y : real; x : real) : real;`
Description	This is the two-argument arc tangent function. It is similar to computing the arc tangent of y/x, except that the signs of both arguments are used to determine the quadrant of the result and x is permitted to be zero. The return value is in radians, between -đ and đ (inclusive).

ceil

Prototype	`function ceil(x : real) : real;`
Description	The function returns value of parameter x rounded up to the next whole number.

cos

Prototype	`function cos(arg : real) : real;`
Description	The function returns the cosine of arg in radians. The return value is from -1 to 1.

cosh

Prototype	`function cosh(x : real) : real;`
Description	The function returns the hyperbolic cosine of x, defined mathematically as (ex+e-x)/2. If the value of x is too large (if overflow occurs), the function fails.

eval_poly

Prototype	`function eval_poly(x : real; var d : array[10] of real; n : integer) : real;`
Description	Function Calculates polynom for number x, with coefficients stored in d[], for degree `n`.

exp

Prototype	`function exp(x : real) : real;`
Description	The function returns the value of e — the base of natural logarithms — raised to the power x (i.e. ex).

fabs

Prototype	`function fabs(d : real) : real;`
Description	The function returns the value of parameter x rounded down to the nearest integer.

frexp

Prototype	`function frexp(value : real; var eptr : integer) : real;`
Description	The function splits a floating-point value value into a normalized fraction and an integral power of 2. The return value is a normalized fraction and the integer exponent is stored in the object pointed to by `eptr`.

ldexp

Prototype	`function ldexp(value : real; newexp : integer) : real;`
Description	The function returns the result of multiplying the floating-point number value by 2 raised to the power newexp (i.e. returns $\texttt{value * 2}^{\texttt{newexp}}$).

log

Prototype	`function log(x : real) : real;`
Description	The function returns the natural logarithm of x (i.e. `loge(x)`).

log10

Prototype	`function log10(x : real) : real;`
Description	The function returns the base-10 logarithm of x (i.e. $\texttt{log}_{10}\texttt{(x)}$).

modf

Prototype	`function modf(val : real; var iptr : real) : real;`
Description	The function returns the signed fractional component of `val`, placing its whole number component into the variable pointed to by iptr.

pow

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

sin

Prototype	`function sin(arg : real) : real;`
Description	The function returns the sine of `arg` in radians. The return value is from -1 to 1.

sinh

Prototype	`function sinh(x : real) : real;`
Description	The function returns the hyperbolic sine of x, defined mathematically as (ex-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	The function returns the non negative square root of x.

tan

Prototype	`function tan(x : real) : real;`
Description	The function returns the tangent of `x` in radians. The return value spans the allowed range of floating point in mikroPascal PRO for AVR.

tanh

Prototype	`function tanh(x : real) : real;`
Description	The function returns the hyperbolic tangent of x, defined mathematically as sinh(x)/cosh(x).

STRING LIBRARY

The mikroPascal PRO for AVR includes a library which automatizes string related tasks.

Library Functions

- memchr
- memcmp
- memcpy
- memmove
- memset
- strcat
- strchr
- strcmp
- strcpy
- strlen
- strncat
- strncpy
- strspn
- strcspn
- strncmp
- strpbrk
- strrchr
- strstr

memchr

Prototype	`function memchr(p : ^byte; ch : byte; n : word) : word;`
Description	The function locates the first occurrence of the word ch in the initial n words of memory area starting at the address p. The function returns the offset of this occurrence from the memory address p or 0xFFFF if ch was not found. For the parameter p you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring or @PORTB`.

memcmp

Prototype	`function memcmp(p1, p2 : ^byte; n : word) : short;`

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

memcpy

Prototype	`procedure memcpy(p1, p2 : ^byte; nn : word);`
Description	The function copies nn words 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 words are copied before being overwritten. If these buffers do overlap, use the memmove function. For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring or @PORTB`.

memmove

Prototype	`procedure memmove(p1, p2 : ^byte; nn : word);`
Description	The function copies nn words 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 the words in p2 are copied to p1 before being overwritten. For parameters p1 and p2 you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring or @PORTB`.

memset

Prototype	`procedure memset(p : ^byte; character : byte; n : word);`
Description	The function fills the first n words in the memory area starting at the address p with the value of word `character`. For parameter p you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `@mystring or @PORTB`.

strcat

Prototype	`procedure strcat(var s1, s2 : string[100]);`
Description	The 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 : byte) : word;`
Description	The function searches the string s for the first occurrence of the character ch. The null character terminating s is not included in the search. The function returns the position (index) of the first character ch found in s; if no matching character was found, the function returns `0xFFFF`.

strcmp

Prototype	`function strcmp(var s1, s2 : string[100]) : short;`

Description	The function lexicographically compares the contents of the strings s1 and s2 and returns a value indicating their relationship:
	
	Value **Meaning**
	`< 0` `s1 "less than" s2`
	`= 0` `s1 "equal to" s2`
	`> 0` `s1 "greater than" s2`
	
	The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared.

strcpy

Prototype	`procedure strcpy(var s1, s2 : string[100]);`
Description	The function copies the value of the 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]) : word;`
Description	The function searches the string s1 for any of the characters in the string s2. The function returns the index of the first character located in s1 that matches any character in s2. If the first character in s1 matches a character in s2, a value of 0 is returned. If there are no matching characters in s1, the length of the string is returned (not including the terminating null character).

strlen

Prototype	`function strlen(var s : string[100]) : word;`
Description	The function returns the length, in words, of the string s. The length does not include the null terminating character.

strncat

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

strncmp

Prototype	`function strncmp(var s1, s2 : string[100]; len : byte) : short;`

Description	The function lexicographically compares the first len words of the strings s1 and s2 and returns a value indicating their relationship:
	
	Value **Meaning**
	`< 0      s1 "less than" s2`
	`= 0      s1 "equal to" s2`
	`> 0      s1 "greater than" s2`
	
	The value returned by the function is determined by the difference between the values of the first pair of words that differ in the strings being compared (within first len words).

strncpy

Prototype	`procedure strncpy(var s1, s2 : string[100]; size : word);`
Description	The function copies at most size characters from the string s2 to the string s1. If s2 contains fewer characters than size, s1 is padded out with null characters up to the total length of the size characters.

strpbrk

Prototype	`function strpbrk(var s1, s2 : string[100]) : word;`
Description	The function searches s1 for the first occurrence of any character from the string s2. The null terminator is not included in the search. The function returns an index of the matching character in s1. If s1 contains no characters from s2, the function returns 0xFFFF.

strrchr

Prototype	`function strrchr(var s : string[100]; ch : byte) : word;`
Description	The function searches the string s for the last occurrence of the character ch. The null character terminating s is not included in the search. The function returns an index of the last ch found in s; if no matching character was found, the function returns 0xFFFF.

strspn

Prototype	`function strspn(var s1, s2 : string[100]) : byte;`
Description	The function searches the string s1 for characters not found in the s2 string. The function returns the index of first character located in s1 that does not match a character in s2. If the first character in s1 does not match a character in s2, a value of 0 is returned. If all characters in s1 are found in s2, the length of s1 is returned (not including the terminating null character).

strstr

Prototype	`function strstr(var s1, s2 : string[100]) : word;`
Description	The function locates the first occurrence of the string s2 in the string s1 (excluding the terminating null character). The function returns a number indicating the position of the first occurrence of s2 in s1; if no string was found, the function returns 0xFFFF. If s2 is a null string, the function returns 0.

TIME LIBRARY

The Time Library contains functions and type definitions for time calculations in the UNIX time format which counts the number of seconds since the "epoch". This is very convenient for programs that work with time intervals: the difference between two UNIX time values is a real-time difference measured in seconds.

What is the epoch?
Originally it was defined as the beginning of 1970 GMT. (January 1, 1970 Julian day) GMT, Greenwich Mean Time, is a traditional term for the time zone in England.

The TimeStruct type is a structure type suitable for time and date storage.

Library Routines

- Time_dateToEpoch
- Time_epochToDate
- Time_datediff

Time_dateToEpoch

Prototype	`function Time_dateToEpoch(var ts : TimeStruct) : longint;`
Returns	Number of seconds since January 1, 1970 0h00mn00s.
Description	This function returns the UNIX time : number of seconds since January 1, 1970 0h00mn00s. Parameters : - `ts`: time and date value for calculating UNIX time.
Requires	Nothing.

Example	`var ts1 : TimeStruct;`
	`    Epoch : longint;`
	`...`
	`// what is the epoch of the date in ts ?`
	`epoch := Time_dateToEpoch(ts1) ;`

Time_epochToDate

Prototype	procedure Time_epochToDate(e: longint; var ts : TimeStruct);
Returns	Nothing.
Description	Converts the UNIX time to time and date. Parameters : - e: UNIX time (seconds since UNIX epoch) - ts: time and date structure for storing conversion output
Requires	Nothing.

Example	var ts2 : TimeStruct;
	    epoch : longint;
	...
	//what date is epoch 1234567890 ?
	epoch := 1234567890 ;
	Time_epochToDate(epoch,ts2);

Time_dateDiff

Prototype	function Time_dateDiff(t1 : ^TimeStruct; t2 : ^TimeStruct) : longint ;
Returns	Time difference in seconds as a signed long.
Description	This function compares two dates and returns time difference in seconds as a signed long. The result is positive if t1 is before t2, null if t1 is the same as t2 and negative if t1 is after t2. Parameters : - t1: time and date structure (the first comparison parameter) - t2: time and date structure (the second comparison parameter)
Requires	Nothing.

Example	var ts1, ts2 : TimeStruct;
	    diff : longint;
	...
	//how many seconds between these two dates contained in ts1 and ts2 buffers?
	 diff := Time_dateDiff(ts1, ts2);

Library Example

Demonstration of Time library routines usage for time calculations in UNIX time format.

```
program Time_Demo;

program Time_Demo;

var epoch, diff : longint;

    ts1, ts2 : TimeStruct;

  begin
    ts1.ss := 0 ;
    ts1.mn := 7 ;
    ts1.hh := 17 ;
    ts1.md := 23 ;
    ts1.mo := 5 ;
    ts1.yy := 2006 ;

    {*
     * What is the epoch of the date in ts ?
     *}
    epoch := Time_dateToEpoch(ts1) ;


    {*
     * What date is epoch 1234567890 ?
     *}
    epoch := 1234567890 ;
    Time_epochToDate(epoch, ts2) ;

    {*
     * How much seconds between this two dates ?
     *}
    diff := Time_dateDiff(ts1, ts2) ;
  end.
```

TimeStruct type definition

```
type TimeStruct = record

        ss : byte ;       // seconds
        mn : byte ;       // minutes
        hh : byte ;       // hours
        md : byte ;       // day in month, from 1 to 31
         wd : byte ;      // day in week, monday=0, tuesday=1, ....
sunday=6
        mo : byte ;       // month number, from 1 to 12 (and not
from 0 to 11 as with unix C time !)
     yy : word ;          // year Y2K compliant, from 1892 to 2038
        end;
```

TRIGONOMETRY LIBRARY

The mikroPascal PRO for AVR implements fundamental trigonometry functions. These functions are implemented as look-up tables. Trigonometry functions are implemented in integer format in order to save memory.

Library Routines

- sinE3
- cosE3

sinE3

Prototype	`function sinE3(angle_deg : word): integer;`
Returns	The function returns the sine of input parameter.
Description	The function calculates sine multiplied by 1000 and rounded to the nearest integer: `result := round(sin(angle_deg)*1000)` Parameters: - `angle_deg`: input angle in degrees **Note**: Return value range: -`1000..1000`.
Requires	Nothing.

Example	`var res : integer;`
	`...`
	`res := sinE3(45);  // result is 707`

cosE3

Prototype	`function cosE3(angle_deg : word): integer;`
Returns	The function returns the cosine of input parameter.
Description	The function calculates cosine multiplied by 1000 and rounded to the nearest integer: - `result := round(cos(angle_deg)*1000)` Parameters: - `angle_deg`: input angle in degrees **Note:** Return value range: -`1000..1000`.
Requires	Nothing.

Example	`var res: integer;`
	`...`
	`res := cosE3(196);   // result is -193`

MikroElektronika

SOFTWARE AND HARDWARE SOLUTIONS FOR EMBEDDED WORLD ...making it simple

If you have any other question, comment or a business proposal, please contact us: web: www.mikroe.com
e-mail: office@mikroe.com

If you are experiencing problems with any of our products
or you just want additional information, please let us know. TECHNICAL SUPPORT: www.mikroe.com/en/support