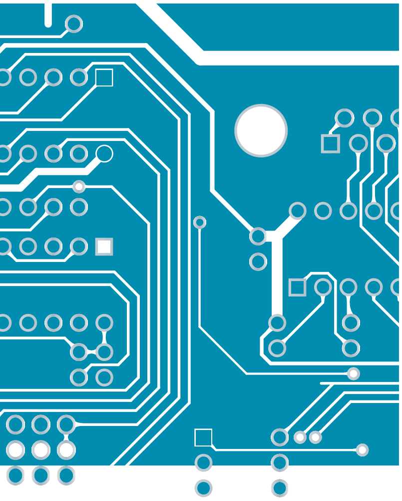
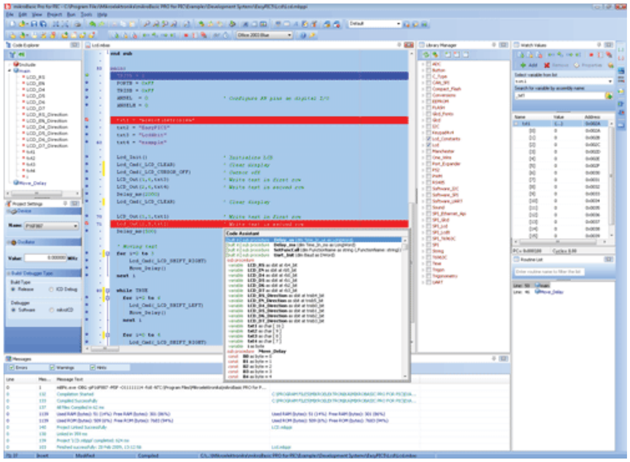
MikroElektronika

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

mikroC PRO for PIC





USER MANUAL

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

April 2009.

Reader's note

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

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

This manual covers *mikroC PRO for PIC* version 1.1 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 *mikroC PRO for PIC*
- 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

Windows is a Registered trademark of Microsoft Corp. All other trade and/or services marks are the property of the respective owners.

Table of Contents

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

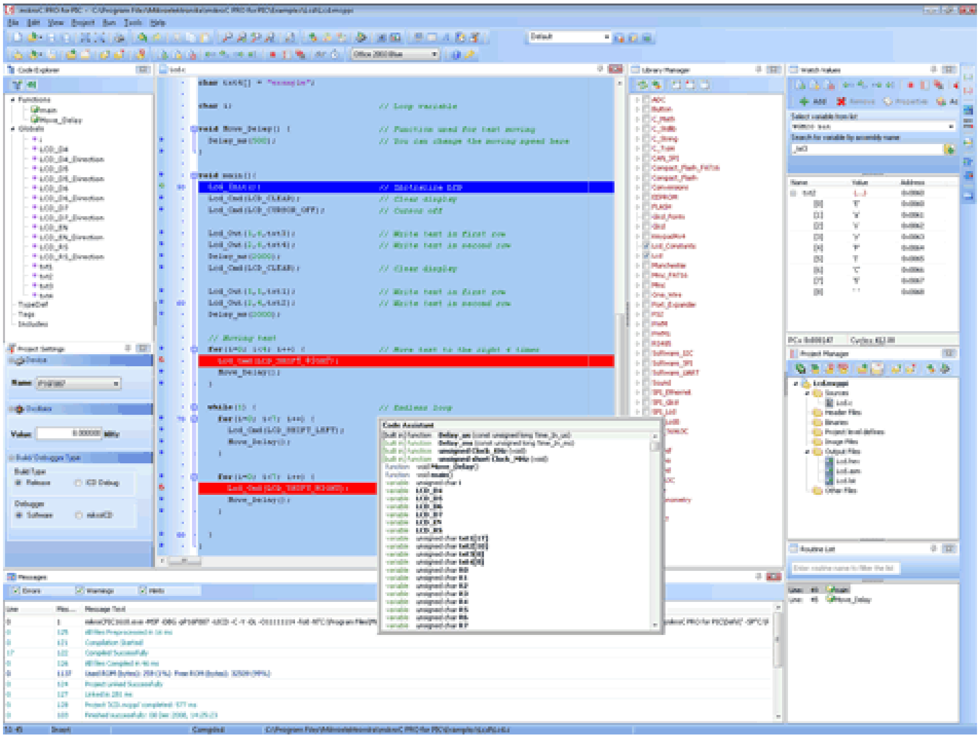
CHAPTER 6

CHAPTER 7

CHAPTER 1

Introduction to *mikroC PRO for PIC*

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



mikroC PRO for PIC IDE

PIC and C fit together well: PIC is the most popular 8-bit chip in the world, used in a wide variety of applications, and C, prized for its efficiency, is the natural choice for developing embedded systems. *mikroC PRO for PIC* provides a successful match featuring highly advanced IDE, ANSI compliant compiler, broad set of hardware libraries, comprehensive documentation, and plenty of ready-to-run examples.

Features

mikroC PRO for PIC allows you to quickly develop and deploy complex applications:

- Write your C source code using the built-in Code Editor (Code and Parameter Assistants, Code Folding, Syntax Highlighting, Auto Correct, Code Templates, and more.)
- Use included *mikroC PRO for PIC* libraries to dramatically speed up the devel opment: 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 compati ble with all programmers.
- Use the integrated mikroICD (In-Circuit Debugger) Real-Time debugging tool to

monitor program execution on the hardware level.

- 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.
- *mikroC PRO for PIC* 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 PIC microcontrollers, read carefully the PIC Specifics chapter. It might give you some useful pointers on PIC constraints, code portability, and good programming practices.

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

- If you are not very experienced in C programming, don't panic! *mikroC PRO for PIC* 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 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 *mikroC PRO for PIC* 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 *mikroC PRO for PIC* is always available for downloading from our website. It is a fully functional software libraries, examples, and comprehensive help included.

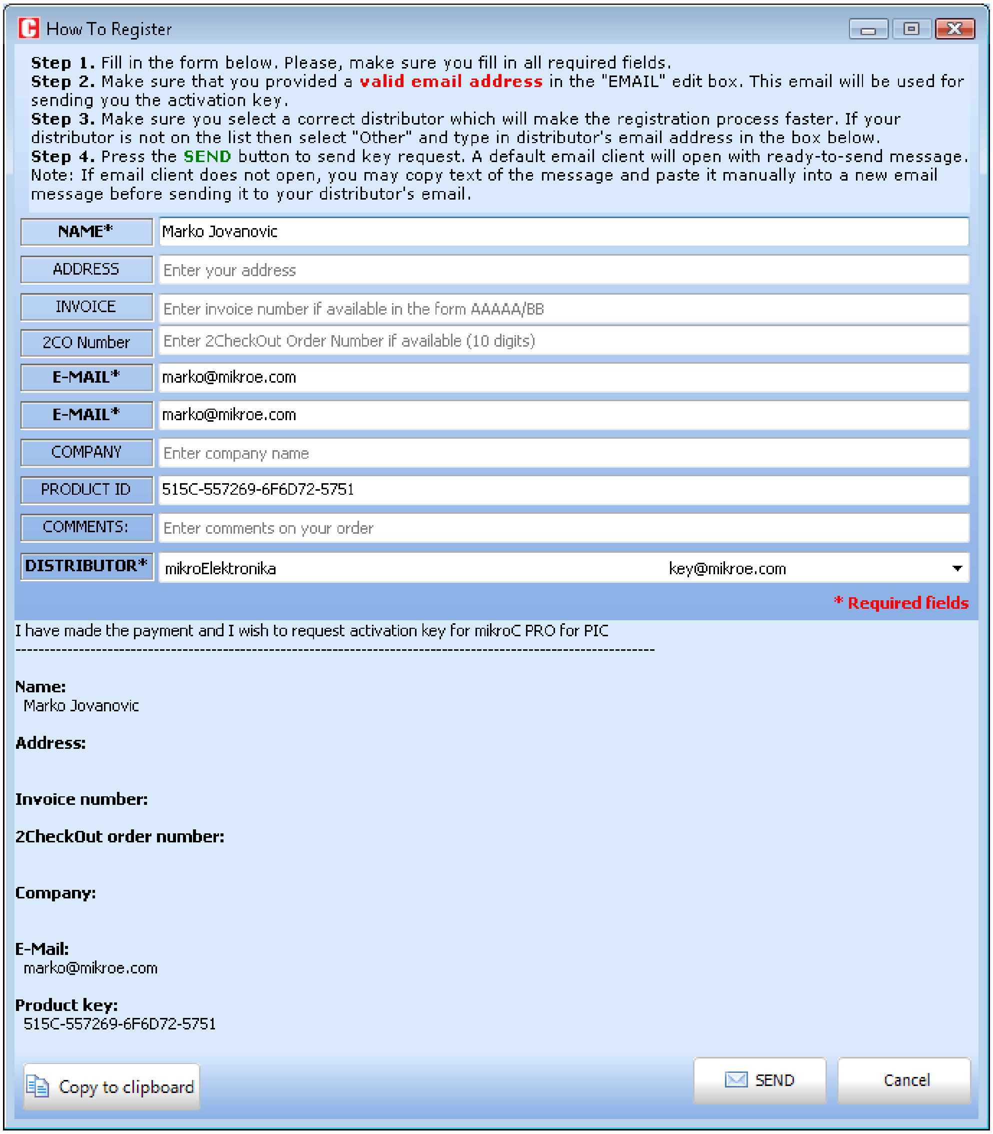
The only limitation of the free version is that it cannot generate hex output over 2 KB. Although it might sound restrictive, this margin allows you to develop practical, working applications with no thinking of demo limit. If you intend to develop really complex projects in the *mikroC PRO for PIC*,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

Who Gets the License Key

Buyers of the *mikroC PRO for PIC* are entitled to the license key. After you have completed the payment procedure, you have an option of registering your mikroC PRO. 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.



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 in the form AAAAA/BB
2CO Number	Enter 2CheckOut Order Number if available (10 digits)
E-MAIL*	marko@mikroe.com
E-MAIL*	marko@mikroe.com
COMPANY	Enter company name
PRODUCT ID	515C-557269-6F6D72-5751
COMMENTS:	Enter comments on your order
DISTRIBUTOR*	mikroElektronika key@mikroe.com

*** Required fields**

I have made the payment and I wish to request activation key for mikroC PRO for PIC

Name:
Marko Jovanovic

Address:

Invoice number:

2CheckOut order number:

Company:

E-Mail:
marko@mikroe.com

Product key:
515C-557269-6F6D72-5751

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 Receiving 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 *mikroC PRO for PIC* at the time of activation.

Notes:

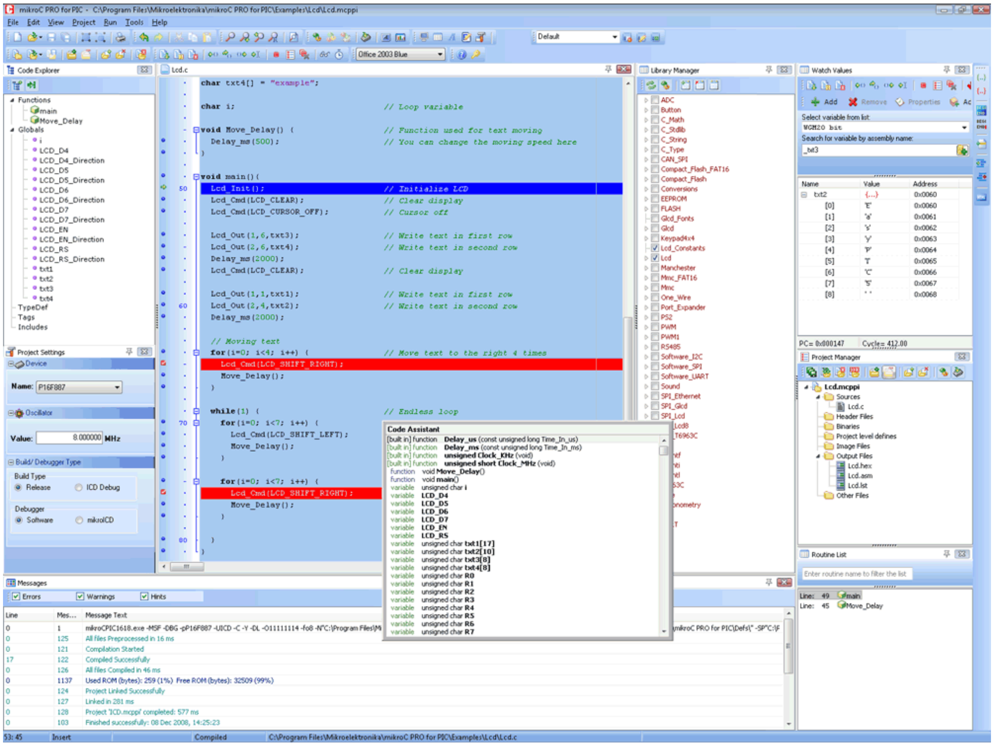
- The license key is valid until you format your hard disk. In case you need to for mat the hard disk, you should request a new activation key.
- Please keep the activation program in a safe place. Every time you upgrade the compiler you should start this program again in order to reactivate the license.

CHAPTER 2

mikroC PRO for PIC Environment

The mikroC PRO for PIC is an user-friendly and intuitive environment.

IDE Overview



```
char txt4[] = "example";

char i;                              // Loop variable

void Move_Delay() {                  // Function used for text moving
  Delay_ms(500);                     // You can change the moving speed here
}

void main(){
  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(2000);

  // Moving text
  for(i=0; i<4; i++) {               // Move text to the right 4 times
    Lcd_Cmd(LCD_SHIFT_RIGHT);
    Move_Delay();
  }

  while(1) {                         // Endless loop
    for(i=0; i<7; i++) {
      Lcd_Cmd(LCD_SHIFT_LEFT);
      Move_Delay();
    }

    for(i=0; i<7; i++) {
      Lcd_Cmd(LCD_SHIFT_RIGHT);
      Move_Delay();
    }

  }
}
```

```
Code Assistant
[built in] function   Delay_us (const unsigned long Time_In_us)
[built in] function   Delay_ms (const unsigned long Time_In_ms)
[built in] function   unsigned Clock_KHz (void)
[built in] function   unsigned short Clock_MHz (void)
function   void Move_Delay()
function   void main()
variable   unsigned char i
variable   LCD_D4
variable   LCD_D5
variable   LCD_D6
variable   LCD_D7
variable   LCD_EN
variable   LCD_RS
variable   unsigned char txt1[17]
variable   unsigned char txt2[10]
variable   unsigned char txt3[8]
variable   unsigned char txt4[8]
variable   unsigned char R0
variable   unsigned char R1
variable   unsigned char R2
variable   unsigned char R3
variable   unsigned char R4
variable   unsigned char R5
variable   unsigned char R6
variable   unsigned char R7
```

```
Line   Mes...   Message Text
0      1        mikroCPIC1618.exe -MSF -DBG -pP16F887 -UICD -C -Y -DL -O11111114 -fo8 -N"C:\Program Files\M   mikroC PRO for PIC\Defs\" -SP"C:\F
0      125      All files Preprocessed in 16 ms
0      121      Compilation Started
17     122      Compiled Successfully
0      126      All files Compiled in 46 ms
0      1137     Used ROM (bytes): 259 (1%)  Free ROM (bytes): 32509 (99%)
0      124      Project Linked Successfully
0      127      Linked in 281 ms
0      128      Project 'ICD.mcppi' completed: 577 ms
0      103      Finished successfully: 08 Dec 2008, 14:25:23
```

- The Code Editor features adjustable Syntax Highlighting, Code Folding, Code Assistant, Parameters Assistant, Auto Correct for common typos and Code Templates (Auto Complete).
- The Code Explorer 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 mikroC PRO for PIC 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

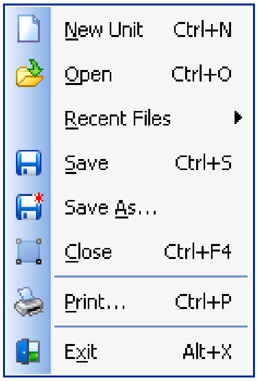
Available Main Menu options are:

- File
- Edit
- View
- Project
- Run
- Tools
- Help

Related topics: Keyboard shortcuts

FILE MENU OPTIONS

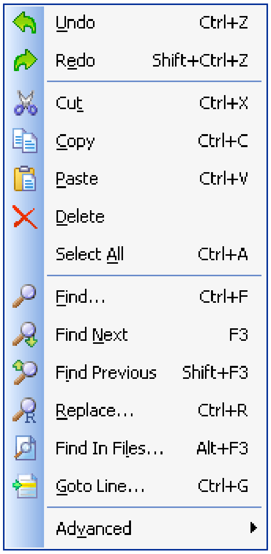
The File menu is the main entry point for manipulation with the source files.



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

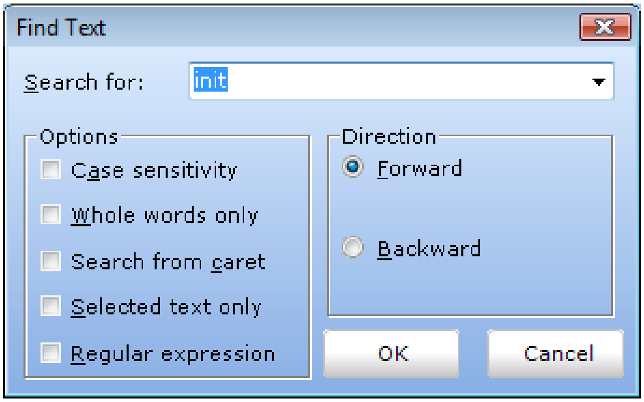


Edit	Description
Undo Ctrl+Z	Undo last change.
Redo Shift+Ctrl+Z	Redo last change.
Cut Ctrl+X	Cut selected text to clipboard.
Copy Ctrl+C	Copy selected text to clipboard.
Paste Ctrl+V	Paste text from clipboard.
Delete	Delete selected text.
Select All Ctrl+A	Select all text in active editor.
Find... Ctrl+F	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

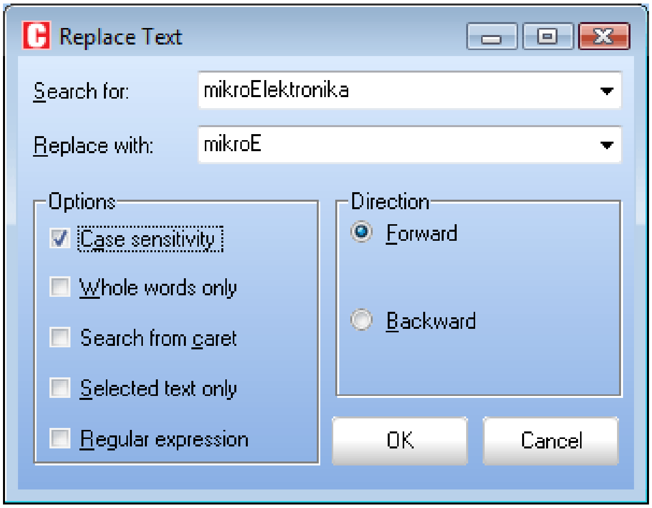
Dialog box for searching the document for the specified text. The search is performed in the direction specified. If the string is not found a message is displayed.



Search for: init

Replace Text

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



Search for: mikroElektronika

Replace with: mikroE

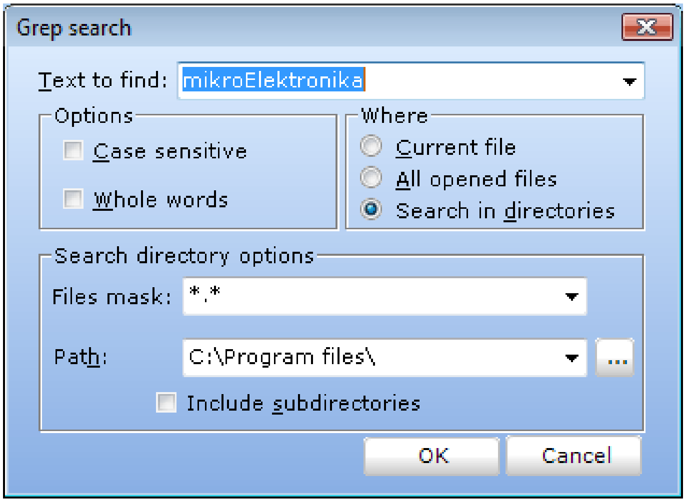
☑ Case sensitivity
☐ Whole words only
☐ Search from caret
☐ Selected text only
☐ Regular expression

● 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.



Text to find: mikroElektronika

☐ Case sensitive
☐ Whole words

○ Current file
○ All opened files
● Search in directories

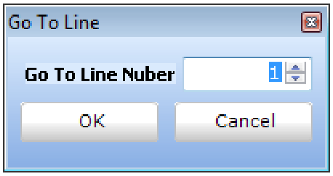
Files mask: *.*

Path: C:\Program files\

☐ Include subdirectories

Go To Line

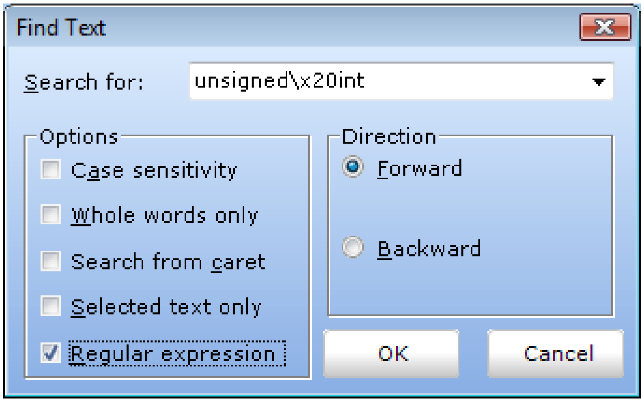
Dialog box that allows the user to specify the line number at which the cursor should be positioned.



Go To Line Nuber 1

Regular expressions option

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



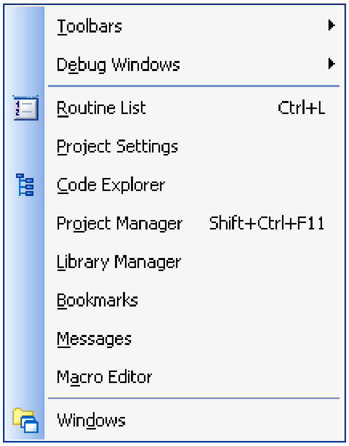
Search for: unsigned\x20int

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

● 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+	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
	New project
	Open Project
	Save Project
	Close current project.
	Edit project settings.
	Add existing project to project group.
	Remove existing project from project group.
	Add File To Project
	Remove File From Project

Build Toolbar

Build Toolbar comes with the following options:

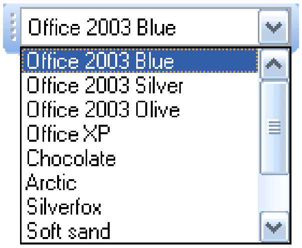
Icon	Description
	Build current project.
	Build all opened projects.
	Build and program active project.
	Start programmer and load current HEX file.
	Open assembly code in editor.
	Open lisitng file in editor.
	View statistics for current project.

Debugger

Debugger Toolbar comes with following options:

Icon	Description
	Start Software Simulator or mikroICD (In-Circuit Debugger).
	Run/Pause debugger.
	Stop debugger.
	Step into.
	Step over.
	Step out.
	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

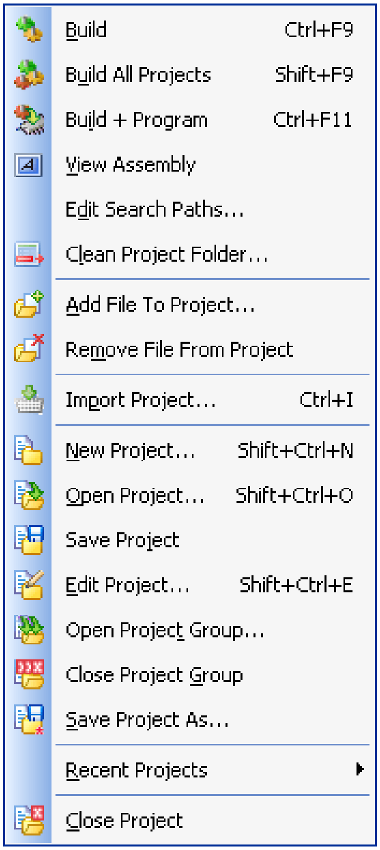
Tools Toolbar comes with following default options:

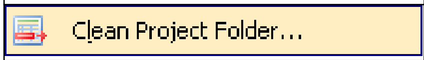
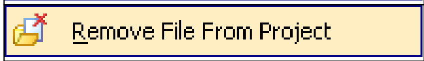
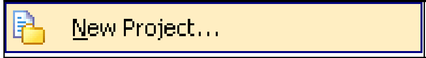
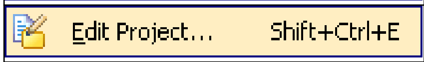
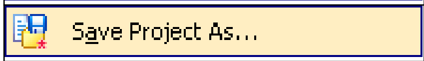
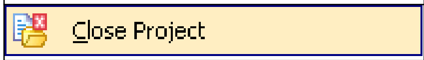
Icon	Description
🖥	Run USART Terminal
▦	EEPROM
A	ASCII Chart
📝	Seven segment decoder tool.
🛠	Options menu

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

Related topics: Keyboard shortcuts, Integrated Tools, Debugger Windows

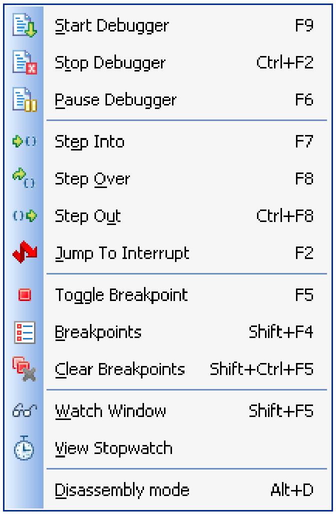
PROJECT MENU OPTIONS



Project	Description
 Build Ctrl+F9	Build active project.
 Build All Projects 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.
 Import Project... Ctrl+I	Import projects from previous versions of mikroC.
 New Project...	Open New Project Wizard
 Open Project... Shift+Ctrl+O	Open existing project.
 Save Project	Save current project.
 Edit Project... Shift+Ctrl+E	Edit project settings
 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

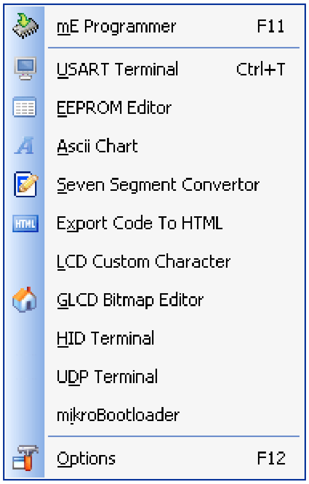
RUN MENU OPTIONS



Run	Description
Start Debugger F9	Start Software Simulator or mikroICD (In-Circuit Debugger).
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

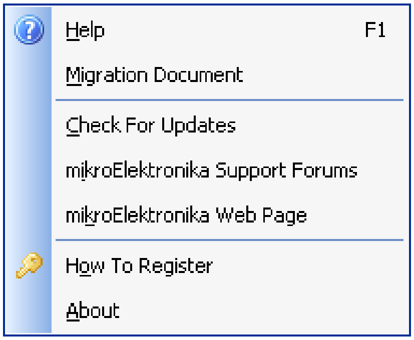
TOOLS MENU OPTIONS



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 Converter
Export Code To HTML	Generate HTML code suitable for publishing source code on the web.
LCD Custom Character	Run Lcd custom character.
GLCD Bitmap Editor	Run Glcd bitmap editor.
HID Terminal	Run HID Terminal.
UDP Terminal	Run UDP communication terminal.
mikroBootloader	Run mikroBootloader.
Options F12	Open Options window.

Related topics: Keyboard shortcuts, Tools Toolbar

HELP MENU OPTIONS



Help	Description
Help F1	Open Help File.
Migration Document	Open Code Migration Document.
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

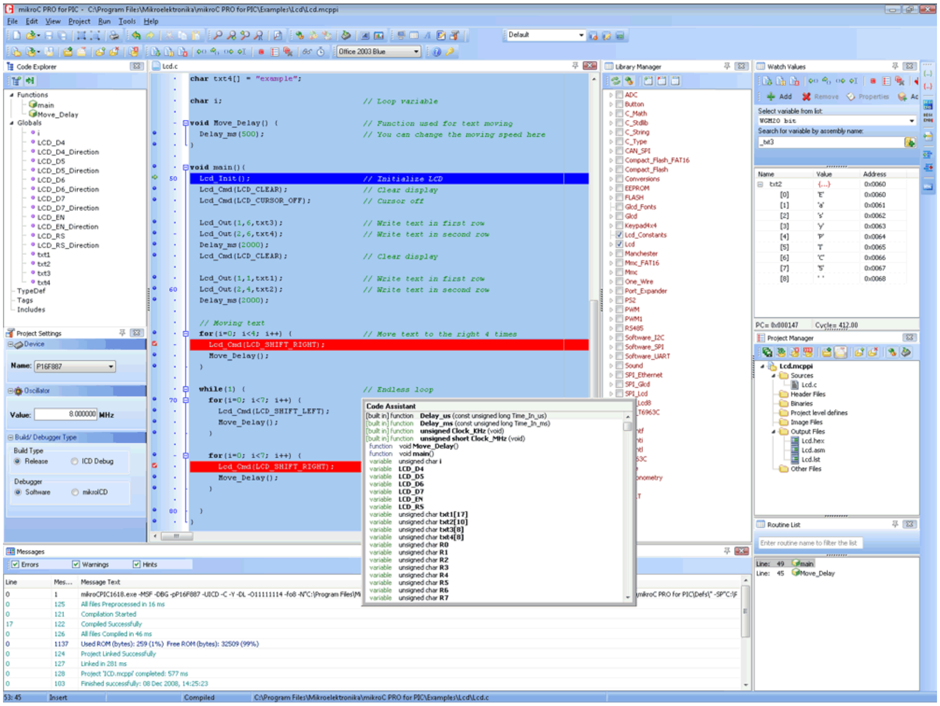
Below is a complete list of keyboard shortcuts available in mikroC PRO for PIC IDE.

IDE Shortcuts	
F1	Help
Ctrl+N	New Unit
Ctrl+O	Open
Ctrl+Shift+O	Open Project
Ctrl+Shift+N	New Project
Ctrl+K	Close Project
Ctrl+F4	Close Unit
Ctr+Shift+E	Edit Project
Ctrl+F9	Build
Shift+F9	Build All
Ctrl+F11	Build And Program
Shift+F4	View Breakpoints
Ctrl+Shift+F5	Clear Breakpoints
F11	Start mE Programmer
Ctrl+Shift+F1	Project Manager
F12	Options
Alt+X	Close mikroC PRO for PIC
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+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

mikroICD Debugger and 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	Stop Debugger
Ctrl+F5	Add to Watch List
Ctrl+F8	Step out
Alt+D	Dissassembly View
Shift+F5	Open Watch Window
Ctrl+Shift+A	Show Advanced Breakpoints

IDE OVERVIEW

The *mikroC PRO for PIC* is an user-friendly and intuitive environment:



```
char txt4[] = "example";

char i;                              // Loop variable

void Move_Delay() {                  // Function used for text moving
  Delay_ms(500);                     // You can change the moving speed here
}

void main(){
  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(2000);

  // Moving text
  for(i=0; i<4; i++) {               // Move text to the right 4 times
    Lcd_Cmd(LCD_SHIFT_RIGHT);
    Move_Delay();
  }

  while(1) {                         // Endless loop
    for(i=0; i<7; i++) {
      Lcd_Cmd(LCD_SHIFT_LEFT);
      Move_Delay();
    }

    for(i=0; i<7; i++) {
      Lcd_Cmd(LCD_SHIFT_RIGHT);
      Move_Delay();
    }

  }
}
```

Line	Mes...	Message Text
0	1	mikroCPIC1618.exe -MSF -DBG -pP16F887 -UICD -C -Y -DL -O11111114 -fo8 -N"C:\Program Files\Mi … mikroC PRO for PIC\Defs\" -SP"C:\F
0	125	All files Preprocessed in 16 ms
0	121	Compilation Started
17	122	Compiled Successfully
0	126	All files Compiled in 46 ms
0	1137	Used ROM (bytes): 259 (1%) Free ROM (bytes): 32509 (99%)
0	124	Project Linked Successfully
0	127	Linked in 281 ms
0	128	Project 'ICD.mcppi' completed: 577 ms
0	103	Finished successfully: 08 Dec 2008, 14:25:23

```
53: 45    Insert    Compiled    C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Lcd\Lcd.c
```

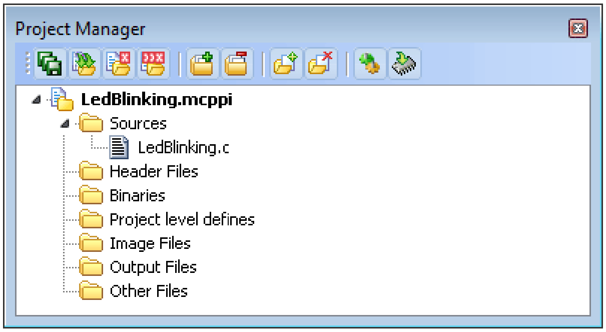
- The Code Editor features adjustable Syntax Highlighting, Code Folding, Code Assistant, Parameters Assistant, Auto Correct for common typos and Code Templates (Auto Complete).
- The Code Explorer is at your disposal for easier project management.
- The Project Manager allows multiple project management
- 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 *mikroC PRO for PIC* 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

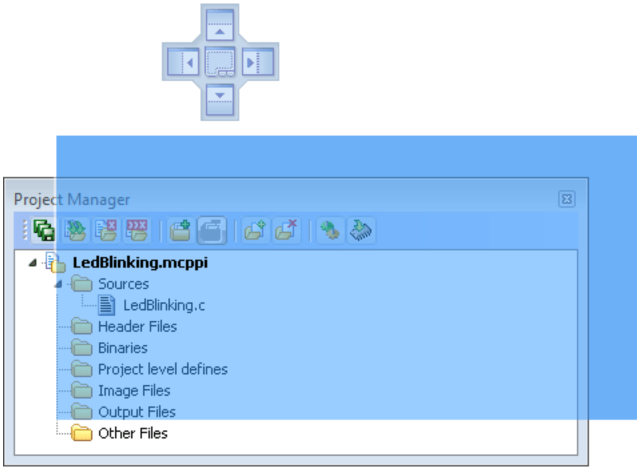
Docking Windows

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

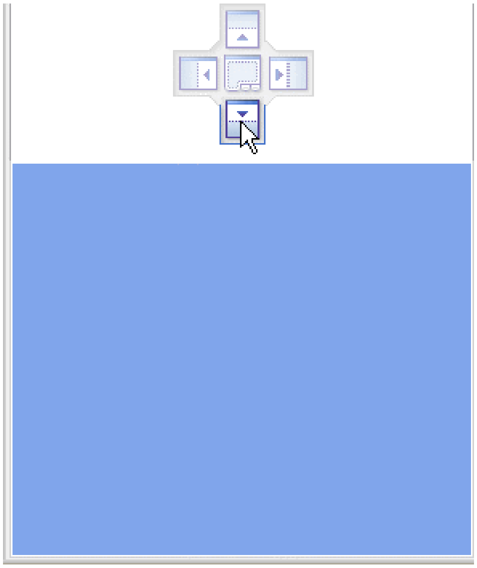
Step 1: Click the window you want to dock, to give it focus.



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.

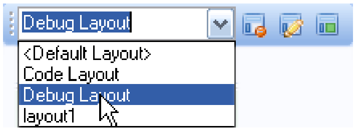


Step 4: To dock the window in the position indicated, release the mouse button.

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 .



Debug Layout

- <Default Layout>
- Code Layout
- Debug Layout
- layout1

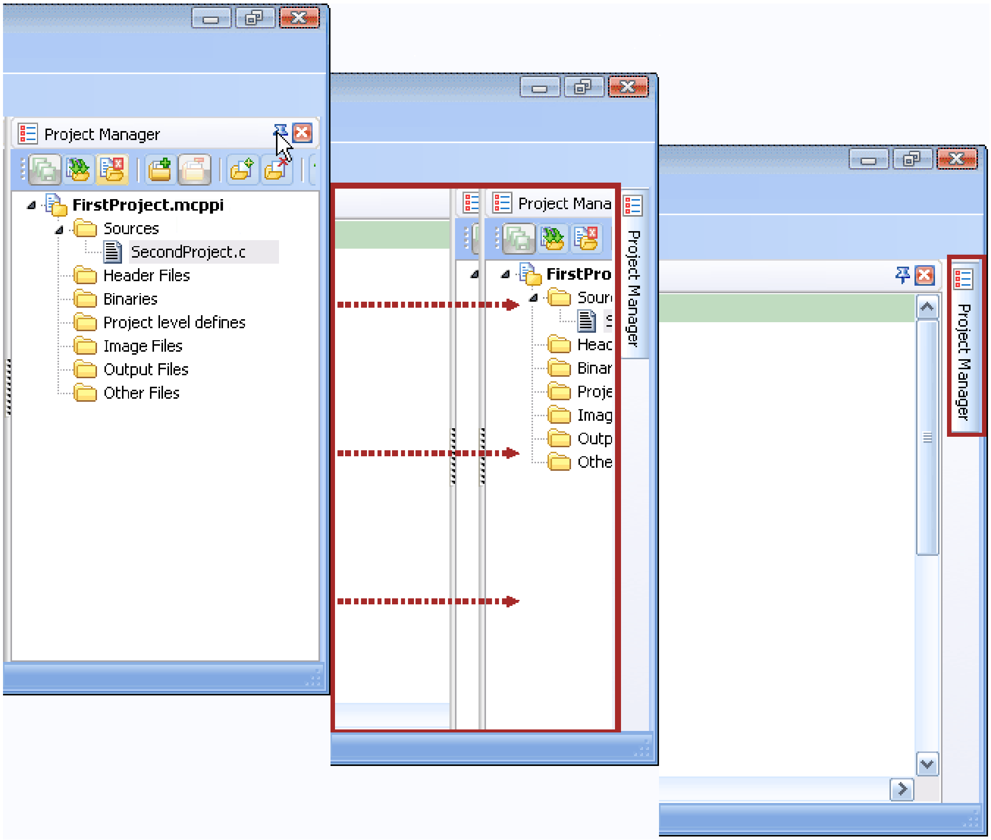
Auto Hide

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.

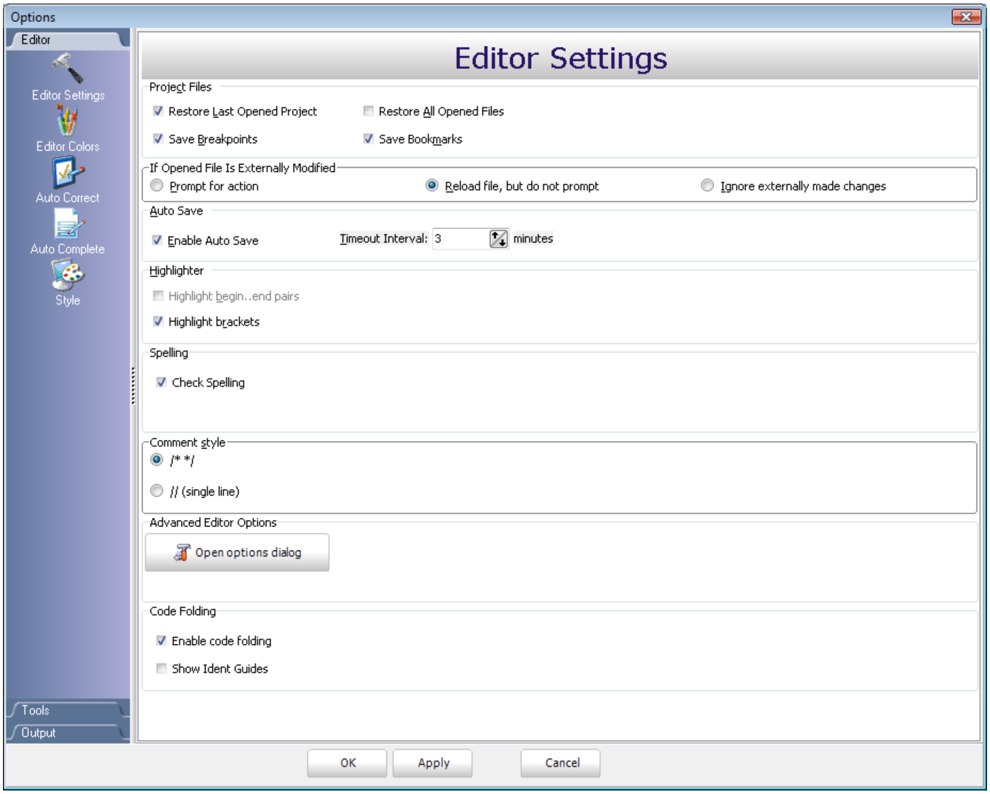
ADVANCED CODE EDITOR

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

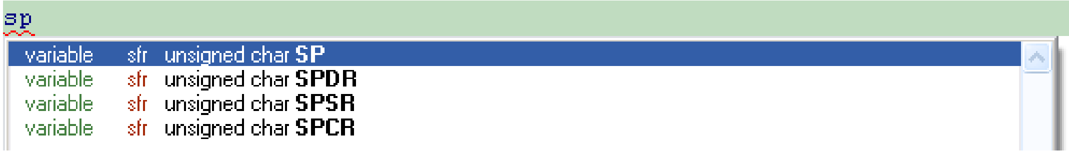
Comment style
● /* */
○ // (single line)

Advanced Editor Options
Open options dialog

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**.

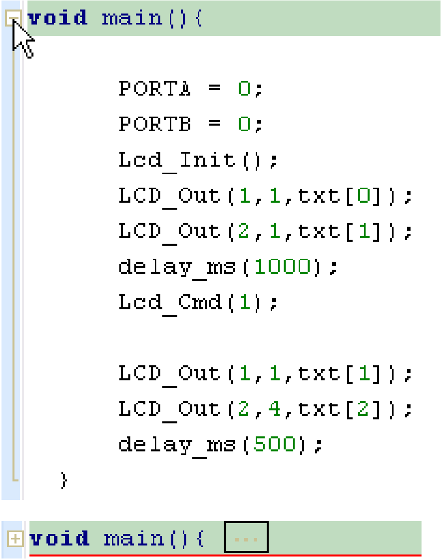


```
sp
variable   sfr   unsigned char SP
variable   sfr   unsigned char SPDR
variable   sfr   unsigned char SPSR
variable   sfr   unsigned char SPCR
```

Code Folding

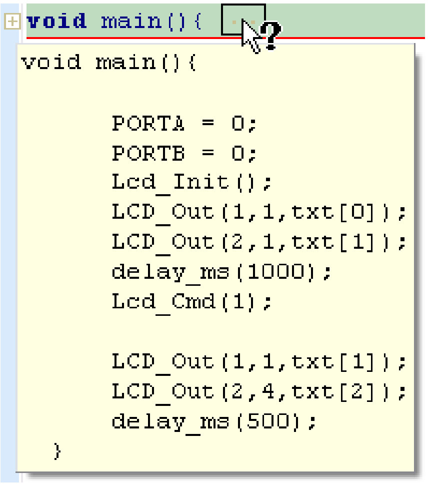
Code folding is IDE feature which allows users to selectively hide and display sections of a source file. In this way it is easier to manage large regions of code within one window, while still viewing only those subsections of the code that are relevant during a particular editing session.

While typing, the code folding symbol (- and +) appear automatically. Use the folding symbols to hide/unhide the code subsections.



```
void main(){

       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);
   }

void main(){ ...
```

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



```
void main(){
```

```
void main(){

      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);
 }
```

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 : char
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 *mikroC PRO for PIC* 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

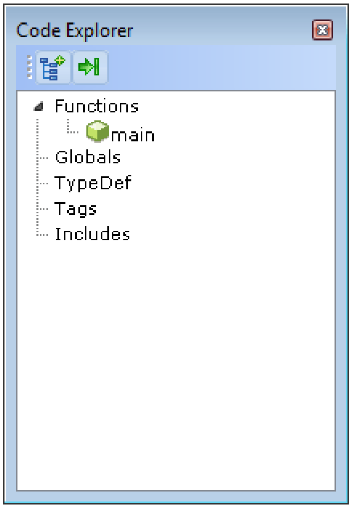
The Goto Line option makes navigation through a large code easier. Use the shortcut **Ctrl+G** to activate this option.

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.



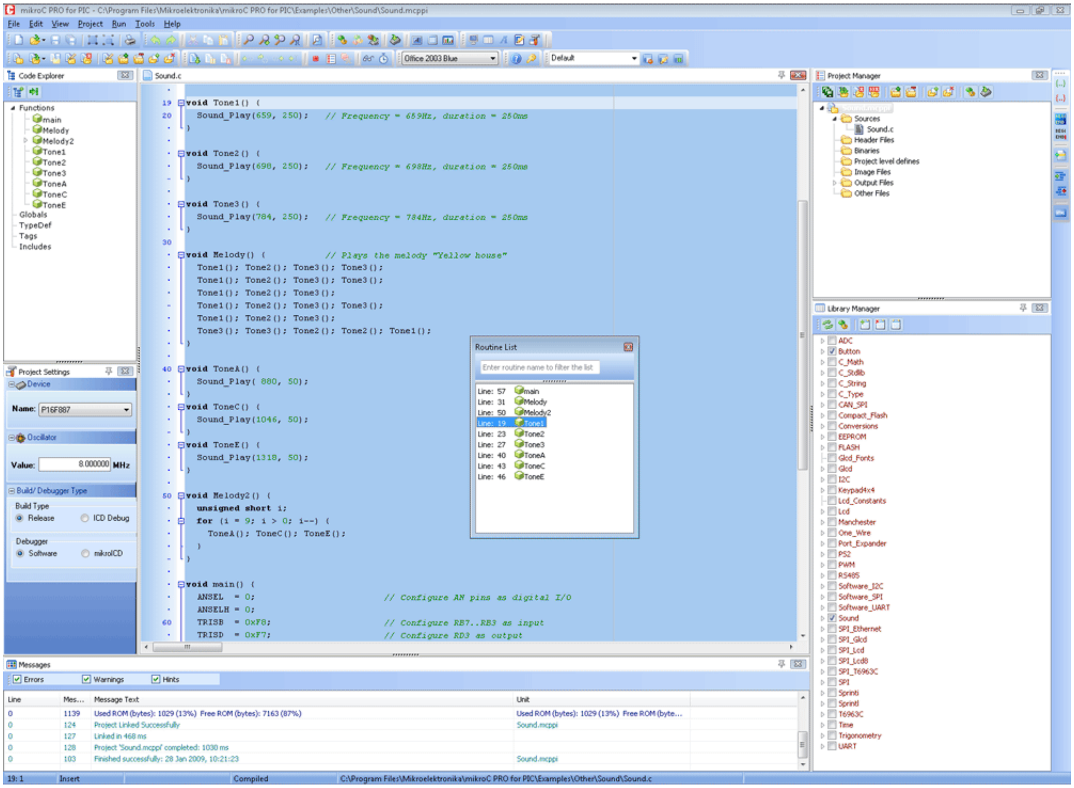
Following options are available in the Code Explorer:

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: P16F887

Value: 8.000000 MHz

```
void Tone1() {
  Sound_Play(659, 250);   // Frequency = 659Hz, duration = 250ms
}

void Tone2() {
  Sound_Play(698, 250);   // Frequency = 698Hz, duration = 250ms
}

void Tone3() {
  Sound_Play(784, 250);   // Frequency = 784Hz, duration = 250ms
}

void Melody() {           // Plays the melody "Yellow house"
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3();
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3();
  Tone3(); Tone3(); Tone2(); Tone2(); Tone1();
}

void ToneA() {
  Sound_Play( 880, 50);
}
void ToneC() {
  Sound_Play(1046, 50);
}
void ToneE() {
  Sound_Play(1318, 50);
}

void Melody2() {
  unsigned short i;
  for (i = 9; i > 0; i--) {
    ToneA(); ToneC(); ToneE();
  }
}

void main() {
  ANSEL  = 0;                        // Configure AN pins as digital I/O
  ANSELH = 0;
  TRISB  = 0xF8;                     // Configure RB7..RB3 as input
  TRISD  = 0xF7;                     // Configure RD3 as output
```

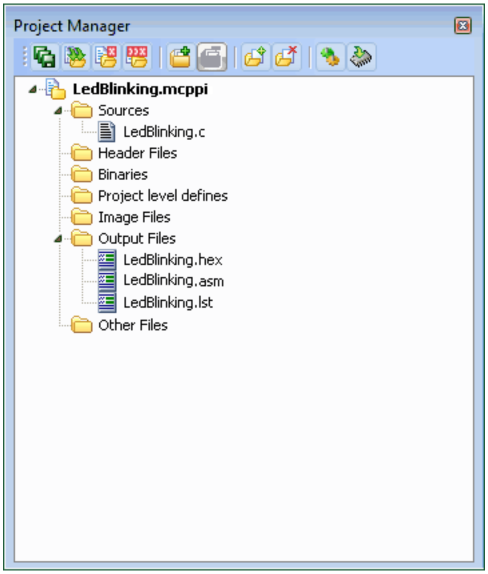
Line: 57 main
Line: 31 Melody
Line: 50 Melody2
Line: 19 Tone1
Line: 23 Tone2
Line: 27 Tone3
Line: 40 ToneA
Line: 43 ToneC
Line: 46 ToneE

Line	Mes...	Message Text	Unit
0	1139	Used ROM (bytes): 1029 (13%) Free ROM (bytes): 7163 (87%)	Used ROM (bytes): 1029 (13%) Free ROM (byte...
0	124	Project Linked Successfully	Sound.mcppi
0	127	Linked in 468 ms	
0	128	Project 'Sound.mcppi' completed: 1030 ms	
0	103	Finished successfully: 28 Jan 2009, 10:21:23	Sound.mcppi

19: 1 Insert Compiled C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Other\Sound\Sound.c

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:

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.

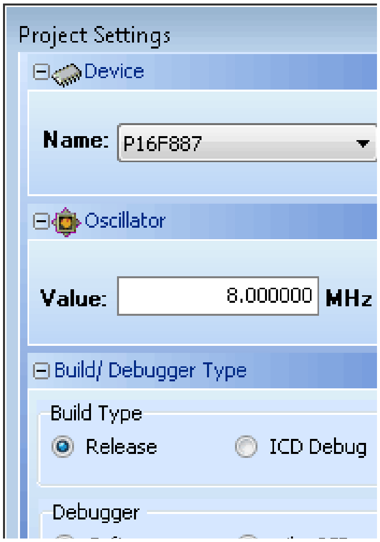
For details about adding and removing files from project see Add/Remove Files from Project.

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.
- Build/Debugger Type - choose debugger type.



Name: P16F887

Value: 8.000000 MHz

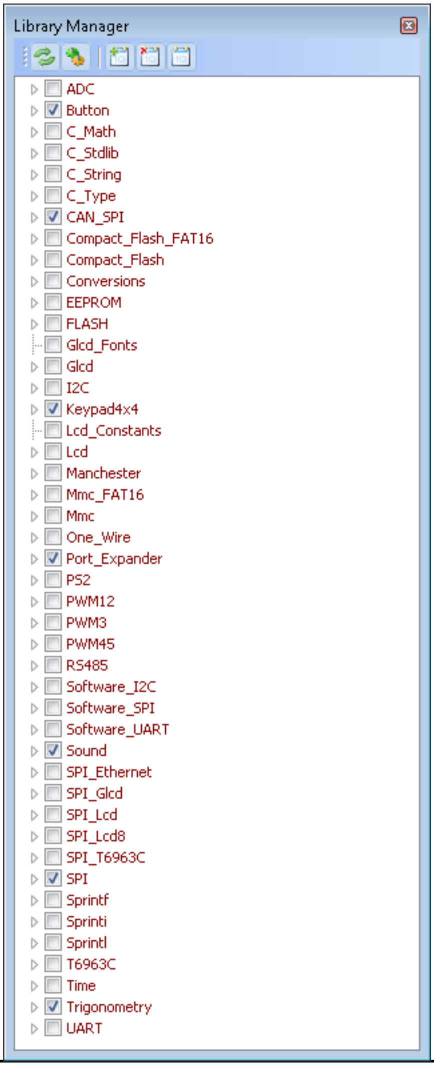
Build Type
● Release ○ ICD Debug

Related topics: Edit Project, Customizing Projects

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_Math
- ☐ C_Stdlib
- ☐ C_String
- ☐ C_Type
- ☑ CAN_SPI
- ☐ Compact_Flash_FAT16
- ☐ Compact_Flash
- ☐ Conversions
- ☐ EEPROM
- ☐ FLASH
- ☐ Glcd_Fonts
- ☐ Glcd
- ☐ I2C
- ☑ Keypad4x4
- ☐ Lcd_Constants
- ☐ Lcd
- ☐ Manchester
- ☐ Mmc_FAT16
- ☐ Mmc
- ☐ One_Wire
- ☑ Port_Expander
- ☐ PS2
- ☐ PWM12
- ☐ PWM3
- ☐ PWM45
- ☐ RS485
- ☐ Software_I2C
- ☐ Software_SPI
- ☐ Software_UART
- ☑ Sound
- ☐ SPI_Ethernet
- ☐ SPI_Glcd
- ☐ SPI_Lcd
- ☐ SPI_Lcd8
- ☐ SPI_T6963C
- ☑ SPI
- ☐ Sprintf
- ☐ Sprinti
- ☐ Sprintl
- ☐ T6963C
- ☐ Time
- ☑ Trigonometry
- ☐ UART

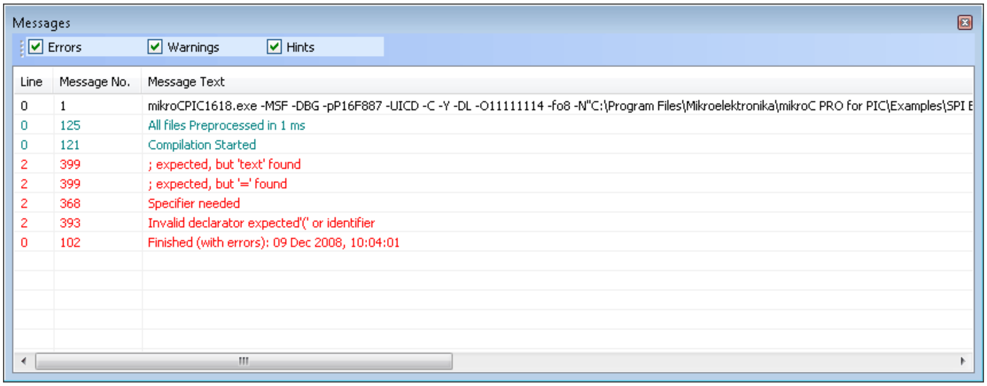
Icon	Description
	Refresh Library by scanning files in "Uses" folder. Useful when new libraries are added by copying files to "Uses" folder.
	Rebuild all available libraries. Useful when library sources are available and need refreshing.
	Include all available libraries in current project.
	No libraries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

Related topics: *mikroC PRO for PIC* 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.



☑ Errors ☑ Warnings ☑ Hints

Line	Message No.	Message Text
0	1	mikroCPIC1618.exe -MSF -DBG -pP16F887 -UICD -C -Y -DL -O11111114 -fo8 -N"C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\SPI
0	125	All files Preprocessed in 1 ms
0	121	Compilation Started
2	399	; expected, but 'text' found
2	399	; expected, but '=' found
2	368	Specifier needed
2	393	Invalid declarator expected'(' or identifier
0	102	Finished (with errors): 09 Dec 2008, 10:04:01

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

Related topics: Error Messages

STATISTICS

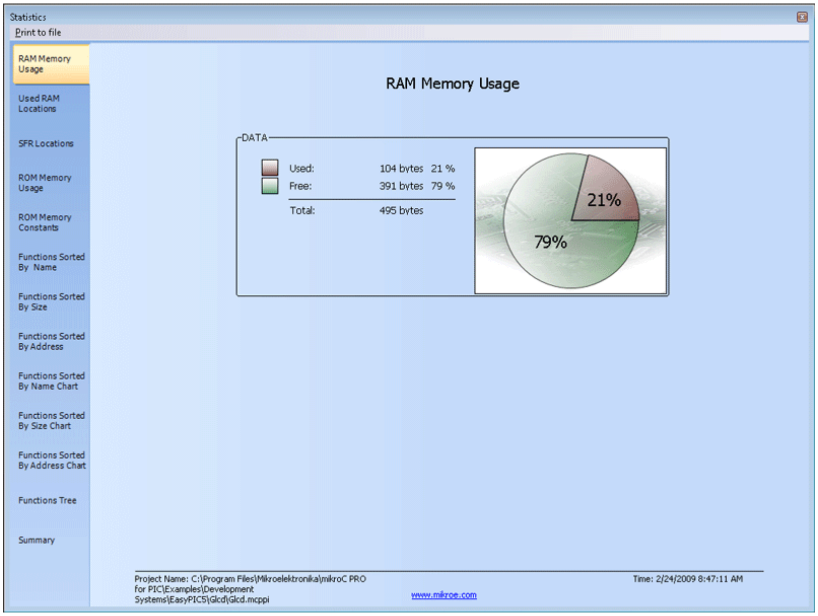
After successful compilation, you can review statistics of your code. Click the Statistics Icon .

Memory Usage Windows

Provides overview of RAM and ROM usage in the various forms.

RAM Memory Usage

Displays RAM memory usage in a pie-like form.



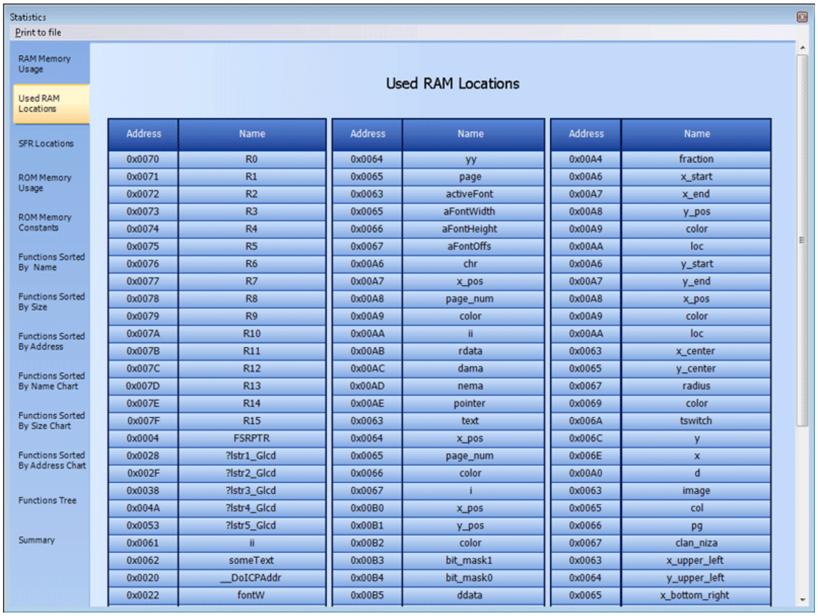
🟥 Used:	104 bytes	21 %
🟩 Free:	391 bytes	79 %
Total:	495 bytes	

Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi

Time: 2/24/2009 8:47:11 AM

Used RAM Locations

Displays used RAM memory locations and their names.

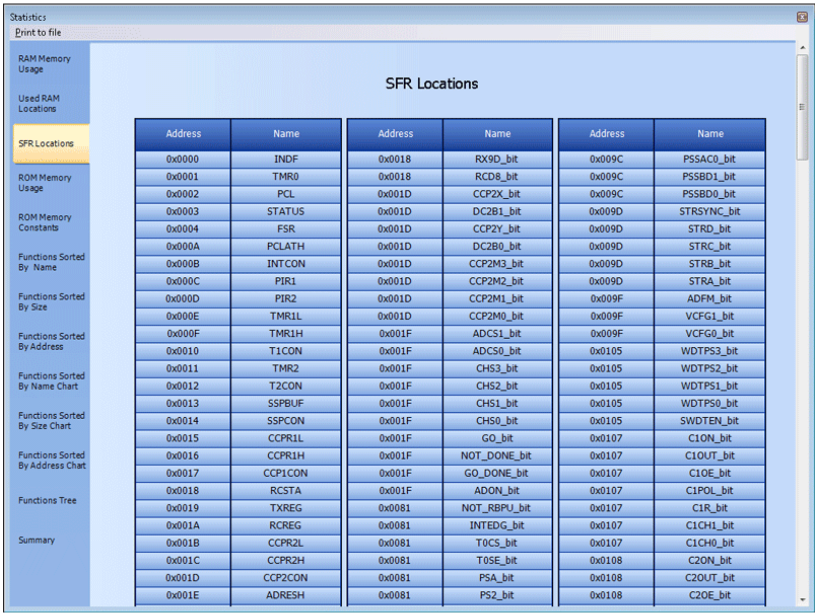


Used RAM Locations

Address	Name	Address	Name	Address	Name
0x0070	R0	0x0064	yy	0x00A4	fraction
0x0071	R1	0x0065	page	0x00A6	x_start
0x0072	R2	0x0063	activeFont	0x00A7	x_end
0x0073	R3	0x0065	aFontWidth	0x00A8	y_pos
0x0074	R4	0x0066	aFontHeight	0x00A9	color
0x0075	R5	0x0067	aFontOffs	0x00AA	loc
0x0076	R6	0x00A6	chr	0x00A6	y_start
0x0077	R7	0x00A7	x_pos	0x00A7	y_end
0x0078	R8	0x00A8	page_num	0x00A8	x_pos
0x0079	R9	0x00A9	color	0x00A9	color
0x007A	R10	0x00AA	ii	0x00AA	loc
0x007B	R11	0x00AB	rdata	0x0063	x_center
0x007C	R12	0x00AC	dama	0x0065	y_center
0x007D	R13	0x00AD	nema	0x0067	radius
0x007E	R14	0x00AE	pointer	0x0069	color
0x007F	R15	0x0063	text	0x006A	tswitch
0x0004	FSRPTR	0x0064	x_pos	0x006C	y
0x0028	?lstr1_Glcd	0x0065	page_num	0x006E	x
0x002F	?lstr2_Glcd	0x0066	color	0x00A0	d
0x0038	?lstr3_Glcd	0x0067	i	0x0063	image
0x004A	?lstr4_Glcd	0x00B0	x_pos	0x0065	col
0x0053	?lstr5_Glcd	0x00B1	y_pos	0x0066	pg
0x0061	ii	0x00B2	color	0x0067	clan_niza
0x0062	someText	0x00B3	bit_mask1	0x0063	x_upper_left
0x0020	__DoICPAddr	0x00B4	bit_mask0	0x0064	y_upper_left
0x0022	fontW	0x00B5	ddata	0x0065	x_bottom_right

SFR Locations

Displays list of used SFR locations.

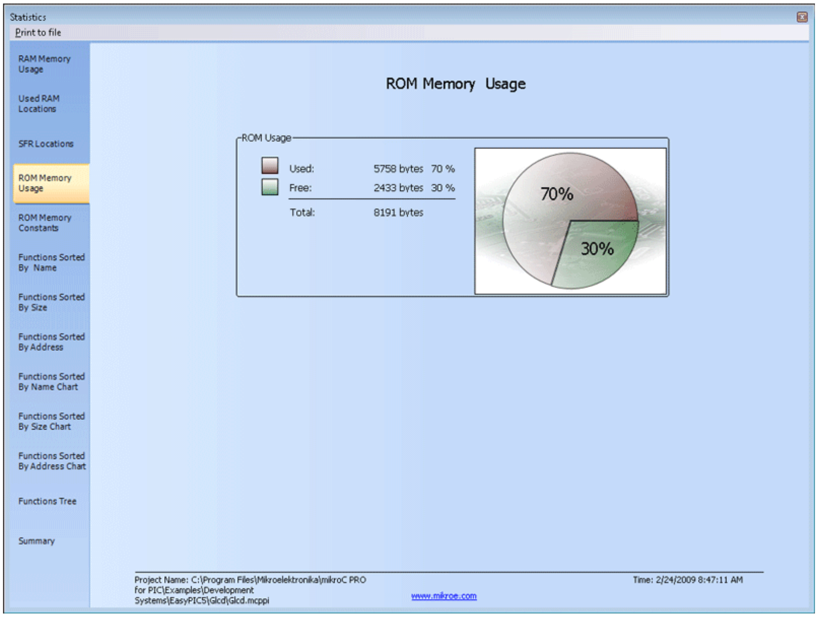


SFR Locations

Address	Name	Address	Name	Address	Name
0x0000	INDF	0x0018	RX9D_bit	0x009C	PSSAC0_bit
0x0001	TMR0	0x0018	RCD8_bit	0x009C	PSSBD1_bit
0x0002	PCL	0x001D	CCP2X_bit	0x009C	PSSBD0_bit
0x0003	STATUS	0x001D	DC2B1_bit	0x009D	STRSYNC_bit
0x0004	FSR	0x001D	CCP2Y_bit	0x009D	STRD_bit
0x000A	PCLATH	0x001D	DC2B0_bit	0x009D	STRC_bit
0x000B	INTCON	0x001D	CCP2M3_bit	0x009D	STRB_bit
0x000C	PIR1	0x001D	CCP2M2_bit	0x009D	STRA_bit
0x000D	PIR2	0x001D	CCP2M1_bit	0x009F	ADFM_bit
0x000E	TMR1L	0x001D	CCP2M0_bit	0x009F	VCFG1_bit
0x000F	TMR1H	0x001F	ADCS1_bit	0x009F	VCFG0_bit
0x0010	T1CON	0x001F	ADCS0_bit	0x0105	WDTPS3_bit
0x0011	TMR2	0x001F	CHS3_bit	0x0105	WDTPS2_bit
0x0012	T2CON	0x001F	CHS2_bit	0x0105	WDTPS1_bit
0x0013	SSPBUF	0x001F	CHS1_bit	0x0105	WDTPS0_bit
0x0014	SSPCON	0x001F	CHS0_bit	0x0105	SWDTEN_bit
0x0015	CCPR1L	0x001F	GO_bit	0x0107	C1ON_bit
0x0016	CCPR1H	0x001F	NOT_DONE_bit	0x0107	C1OUT_bit
0x0017	CCP1CON	0x001F	GO_DONE_bit	0x0107	C1OE_bit
0x0018	RCSTA	0x001F	ADON_bit	0x0107	C1POL_bit
0x0019	TXREG	0x0081	NOT_RBPU_bit	0x0107	C1R_bit
0x001A	RCREG	0x0081	INTEDG_bit	0x0107	C1CH1_bit
0x001B	CCPR2L	0x0081	T0CS_bit	0x0107	C1CH0_bit
0x001C	CCPR2H	0x0081	T0SE_bit	0x0108	C2ON_bit
0x001D	CCP2CON	0x0081	PSA_bit	0x0108	C2OUT_bit
0x001E	ADRESH	0x0081	PS2_bit	0x0108	C2OE_bit

ROM Memory Usage

Displays ROM memory space usage in a pie-like form.



ROM Memory Usage

ROM Usage

🟥	Used:	5758 bytes	70 %
🟩	Free:	2433 bytes	30 %
	Total:	8191 bytes	

Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi

www.mikroe.com

Time: 2/24/2009 8:47:11 AM

ROM Memory Constants

Displays ROM memory constants and their addresses.

ROM Memory Constants

Address	Name
0x0780	?ICS?lstr1_Glcd
0x0787	?ICS?lstr2_Glcd
0x0790	?ICS?lstr3_Glcd
0x07A2	?ICS?lstr4_Glcd
0x07AB	?ICS?lstr5_Glcd
0x0800	Character8x7
0x15EA	font5x7
0x1400	FontSystem5x7_v2
0x06BD	System3x5
0x1000	truck_bmp

Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi

www.mikroe.com

Time: 2/24/2009 8:47:11 AM

Function Sorted by Name

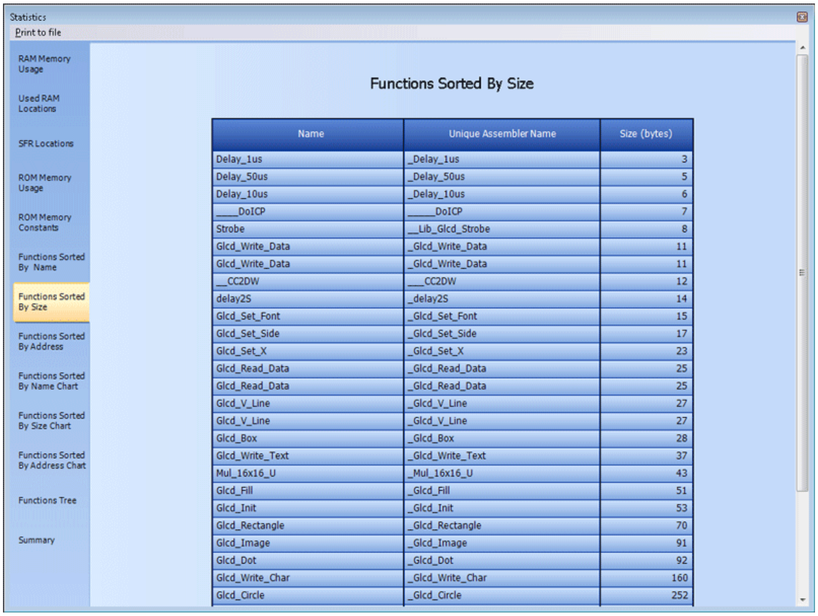
Sorts and displays functions by their addresses, symbolic names, and unique assembler names.

Functions Sorted By Name

Address	Name	Unique Assembler Name
0x006F	____DoICP	_____DoICP
0x01D6	__CC2DW	___CC2DW
0x0005	Delay_10us	_Delay_10us
0x0118	Delay_1us	_Delay_1us
0x0013	Delay_50us	_Delay_50us
0x0540	delay2S	_delay2S
0x0524	Glcd_Box	_Glcd_Box
0x0283	Glcd_Circle	_Glcd_Circle
0x00A1	Glcd_Dot	_Glcd_Dot
0x055D	Glcd_Fill	_Glcd_Fill
0x01BB	Glcd_H_Line	_Glcd_H_Line
0x0228	Glcd_Image	_Glcd_Image
0x04B1	Glcd_Init	_Glcd_Init
0x037F	Glcd_Line	_Glcd_Line
0x0034	Glcd_Read_Data	_Glcd_Read_Data
0x01E2	Glcd_Rectangle	_Glcd_Rectangle
0x054E	Glcd_Set_Font	_Glcd_Set_Font
0x0018	Glcd_Set_Page	_Glcd_Set_Page
0x0023	Glcd_Set_Side	_Glcd_Set_Side
0x004D	Glcd_Set_X	_Glcd_Set_X
0x00FD	Glcd_V_Line	_Glcd_V_Line
0x011B	Glcd_Write_Char	_Glcd_Write_Char
0x0064	Glcd_Write_Data	_Glcd_Write_Data
0x04E6	Glcd_Write_Text	_Glcd_Write_Text
0x0590	main	_main
0x0076	Mul_16x16_U	_Mul_16x16_U

Functions Sorted by Size

Sorts and displays functions by their size, in the ascending order.



Functions Sorted By Size

Name	Unique Assembler Name	Size (bytes)
Delay_1us	_Delay_1us	3
Delay_50us	_Delay_50us	5
Delay_10us	_Delay_10us	6
____DoICP	_____DoICP	7
Strobe	__Lib_Glcd_Strobe	8
Glcd_Write_Data	_Glcd_Write_Data	11
Glcd_Write_Data	_Glcd_Write_Data	11
__CC2DW	___CC2DW	12
delay2S	_delay2S	14
Glcd_Set_Font	_Glcd_Set_Font	15
Glcd_Set_Side	_Glcd_Set_Side	17
Glcd_Set_X	_Glcd_Set_X	23
Glcd_Read_Data	_Glcd_Read_Data	25
Glcd_Read_Data	_Glcd_Read_Data	25
Glcd_V_Line	_Glcd_V_Line	27
Glcd_V_Line	_Glcd_V_Line	27
Glcd_Box	_Glcd_Box	28
Glcd_Write_Text	_Glcd_Write_Text	37
Mul_16x16_U	_Mul_16x16_U	43
Glcd_Fill	_Glcd_Fill	51
Glcd_Init	_Glcd_Init	53
Glcd_Rectangle	_Glcd_Rectangle	70
Glcd_Image	_Glcd_Image	91
Glcd_Dot	_Glcd_Dot	92
Glcd_Write_Char	_Glcd_Write_Char	160
Glcd_Circle	_Glcd_Circle	252

Functions Sorted by Addresses

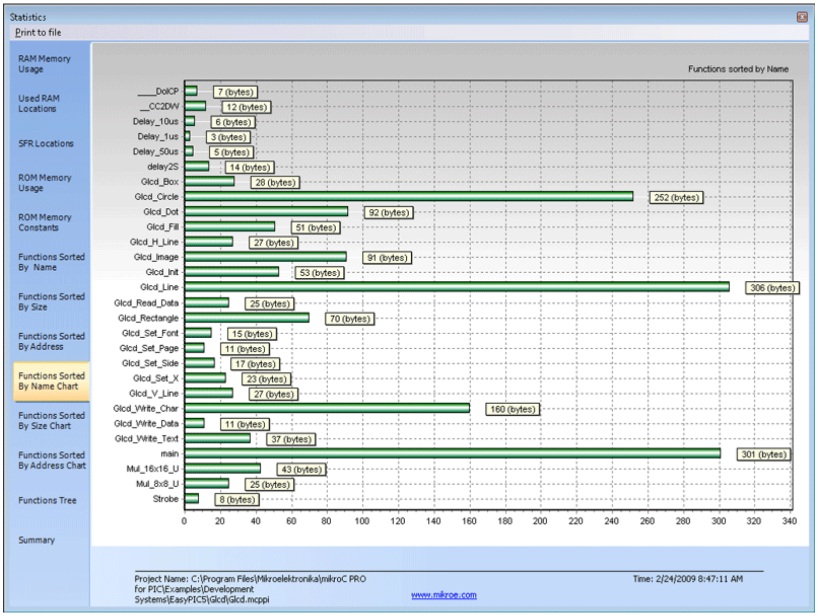
Sorts and displays functions by their addresses, in the ascending order.

Functions Sorted By Address

Address	Name	Unique Assembler Name
0x0005	Delay_10us	_Delay_10us
0x000B	Strobe	__Lib_Glcd_Strobe
0x0013	Delay_50us	_Delay_50us
0x0018	Glcd_Set_Page	_Glcd_Set_Page
0x0023	Glcd_Set_Side	_Glcd_Set_Side
0x0034	Glcd_Read_Data	_Glcd_Read_Data
0x004D	Glcd_Set_X	_Glcd_Set_X
0x0064	Glcd_Write_Data	_Glcd_Write_Data
0x006F	____DoICP	_____DoICP
0x0076	Mul_16x16_U	_Mul_16x16_U
0x00A1	Glcd_Dot	_Glcd_Dot
0x00FD	Glcd_V_Line	_Glcd_V_Line
0x0118	Delay_1us	_Delay_1us
0x011B	Glcd_Write_Char	_Glcd_Write_Char
0x01BB	Glcd_H_Line	_Glcd_H_Line
0x01D6	__CC2DW	___CC2DW
0x01E2	Glcd_Rectangle	_Glcd_Rectangle
0x0228	Glcd_Image	_Glcd_Image
0x0283	Glcd_Circle	_Glcd_Circle
0x037F	Glcd_Line	_Glcd_Line
0x04B1	Glcd_Init	_Glcd_Init
0x04E6	Glcd_Write_Text	_Glcd_Write_Text
0x050B	Mul_8x8_U	_Mul_8x8_U
0x0524	Glcd_Box	_Glcd_Box
0x0540	delay2S	_delay2S
0x054E	Glcd_Set_Font	_Glcd_Set_Font

Functions Sorted by Name Chart

Sorts and displays functions by their names in a chart-like form.

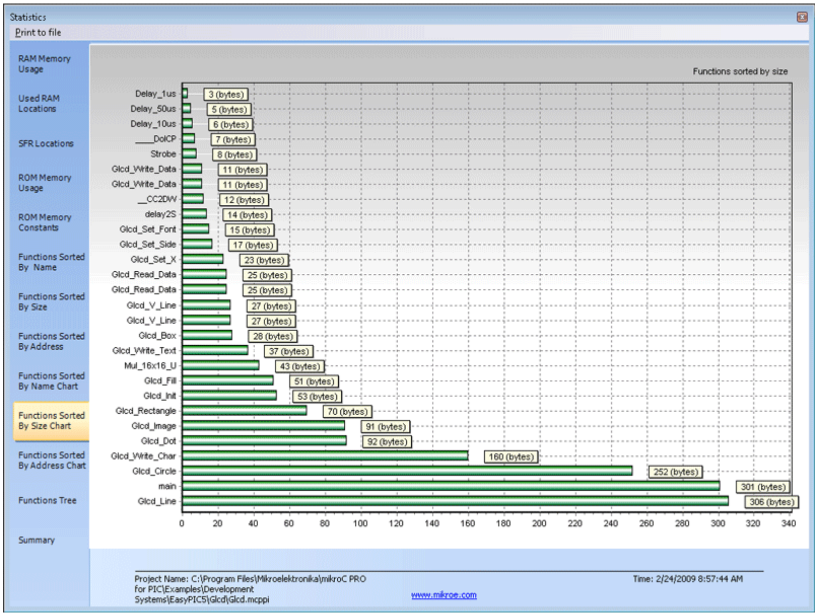


Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi

Time: 2/24/2009 8:47:11 AM

Functions Sorted by Size Chart

Sorts and displays functions by their sizes in a chart-like form.

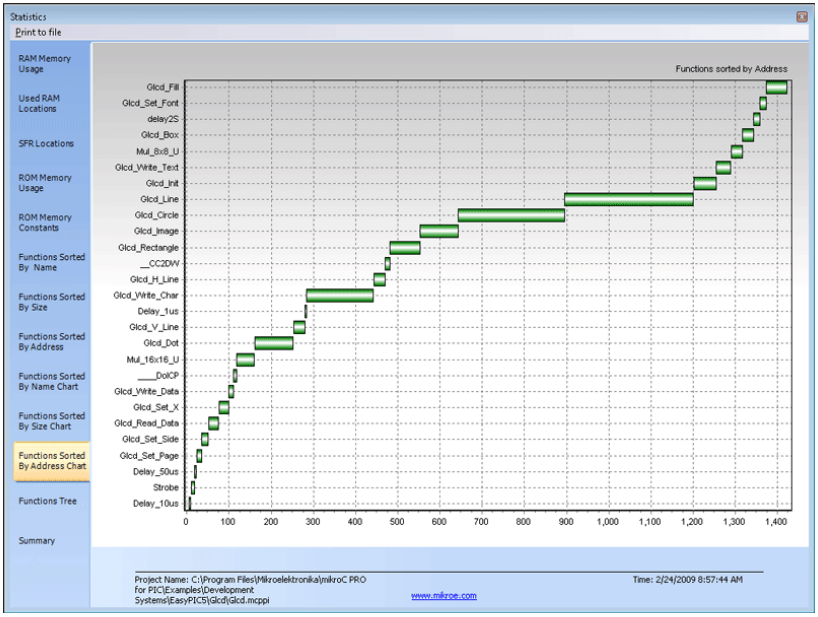


Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi

Time: 2/24/2009 8:57:44 AM

Functions sorted by Address Chart

Sorts and displays functions by their addresses in a chart-like form.

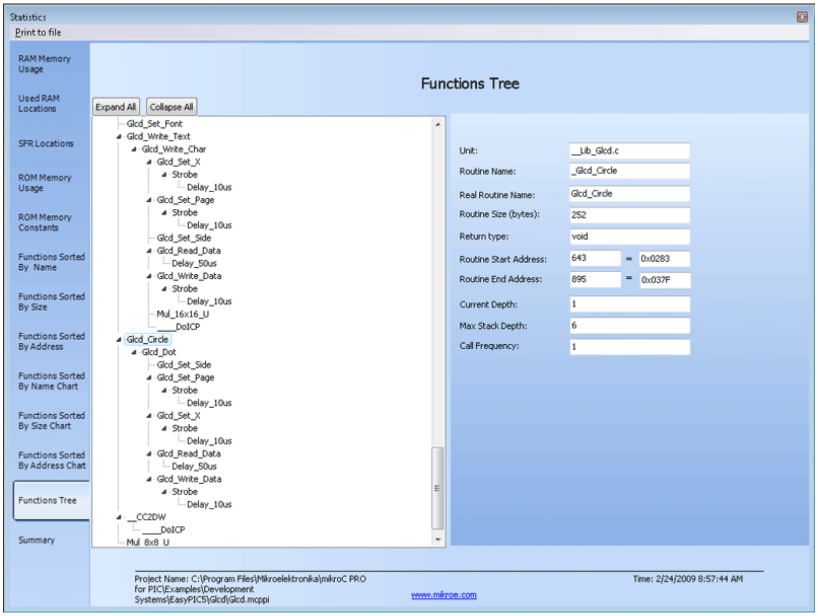


Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi

Time: 2/24/2009 8:57:44 AM

Function Tree

Displays Function Tree with the relevant data for each function.



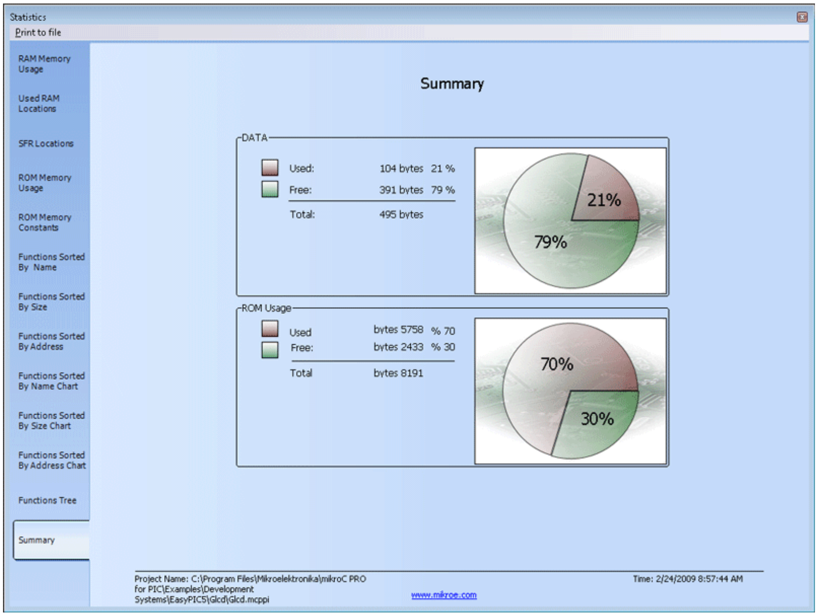
```
Glcd_Set_Font
Glcd_Write_Text
  Glcd_Write_Char
    Glcd_Set_X
      Strobe
        Delay_10us
    Glcd_Set_Page
      Strobe
        Delay_10us
    Glcd_Set_Side
    Glcd_Read_Data
      Delay_50us
    Glcd_Write_Data
      Strobe
        Delay_10us
    Mul_16x16_U
    ____DoICP
Glcd_Circle
  Glcd_Dot
    Glcd_Set_Side
    Glcd_Set_Page
      Strobe
        Delay_10us
    Glcd_Set_X
      Strobe
        Delay_10us
    Glcd_Read_Data
      Delay_50us
    Glcd_Write_Data
      Strobe
        Delay_10us
__CC2DW
  ____DoICP
Mul_8x8_U
```

Unit: __Lib_Glcd.c
Routine Name: _Glcd_Circle
Real Routine Name: Glcd_Circle
Routine Size (bytes): 252
Return type: void
Routine Start Address: 643 = 0x0283
Routine End Address: 895 = 0x037F
Current Depth: 1
Max Stack Depth: 6
Call Frequency: 1

Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi
www.mikroe.com
Time: 2/24/2009 8:57:44 AM

Memory Summary

Displays summary of RAM and ROM memory in a pie-like form.



DATA

🟥	Used:	104 bytes	21 %
🟩	Free:	391 bytes	79 %
	Total:	495 bytes	

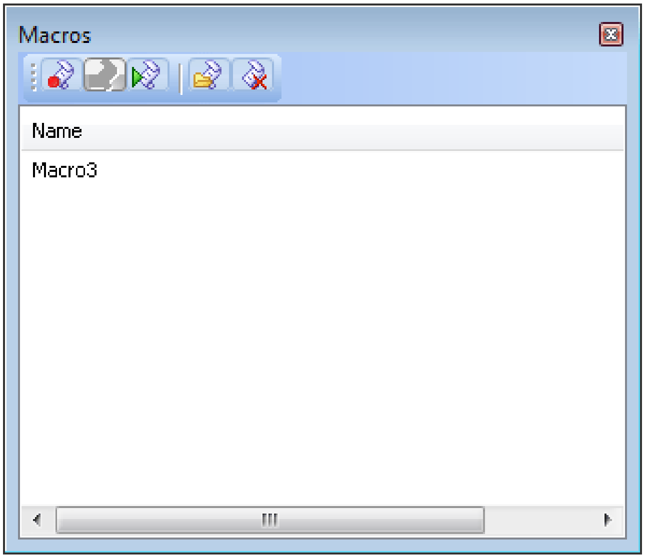
ROM Usage

🟥	Used	bytes 5758	% 70
🟩	Free:	bytes 2433	% 30
	Total	bytes 8191	

Project Name: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Glcd\Glcd.mcppi
www.mikroe.com
Time: 2/24/2009 8:57:44 AM

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.



The Macro offers the following commands:

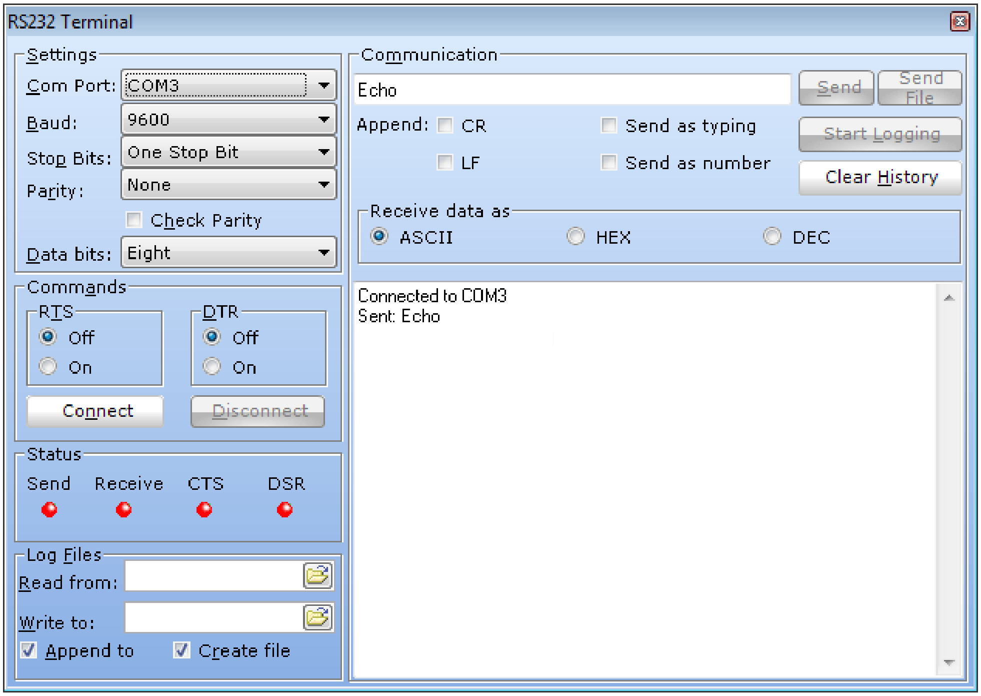
Icon	Description
●	Starts 'recording' keystrokes for later playback.
■	Stops capturing keystrokesthat was started when the Start Recordig command was selected.
▶	Allows a macro that has been recorded to be replayed.
🗁	New macro.
✗	Delete macro.

Related topics: Advanced Code Editor, Code Templates

INTEGRATED TOOLS

USART Terminal

The *mikroC PRO for PIC* 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

Communication

Echo

Append: ☐ CR ☐ LF ☐ Send as typing ☐ Send as number

Receive data as: ● ASCII ○ HEX ○ DEC

Log Files

Read from:

Write to:

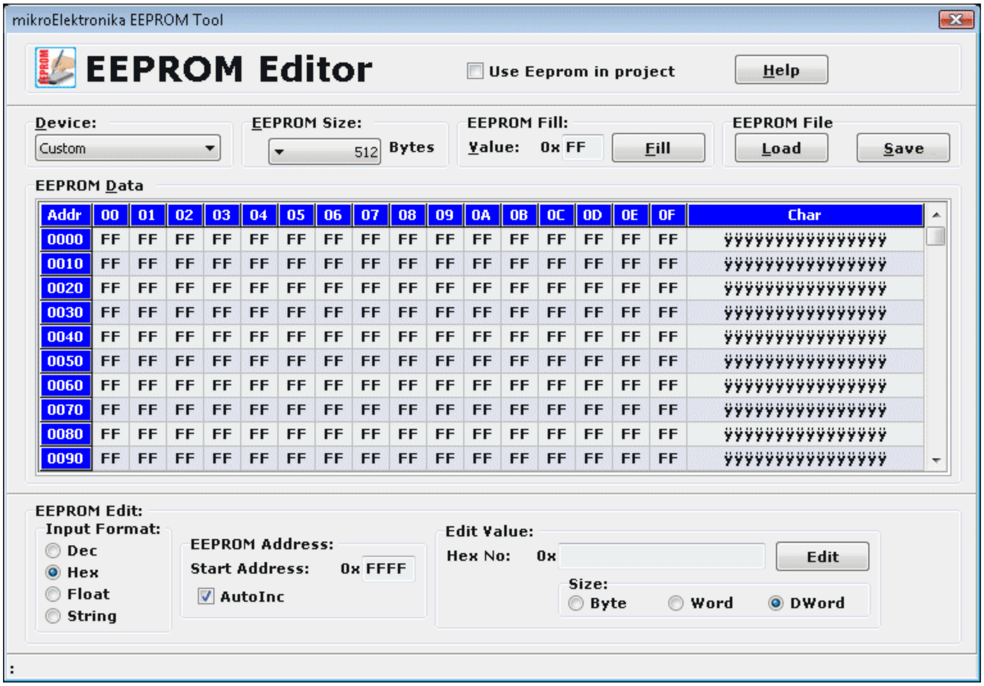
☑ Append to ☑ Create file

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

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 *mikroC PRO for PIC* IDE - `project_name.hex` file will be loaded automatically while `ihex` file must be loaded manually.



☐ Use Eeprom in project

Device: Custom

EEPROM Size: 512 Bytes

EEPROM Fill: Value: 0x FF

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:

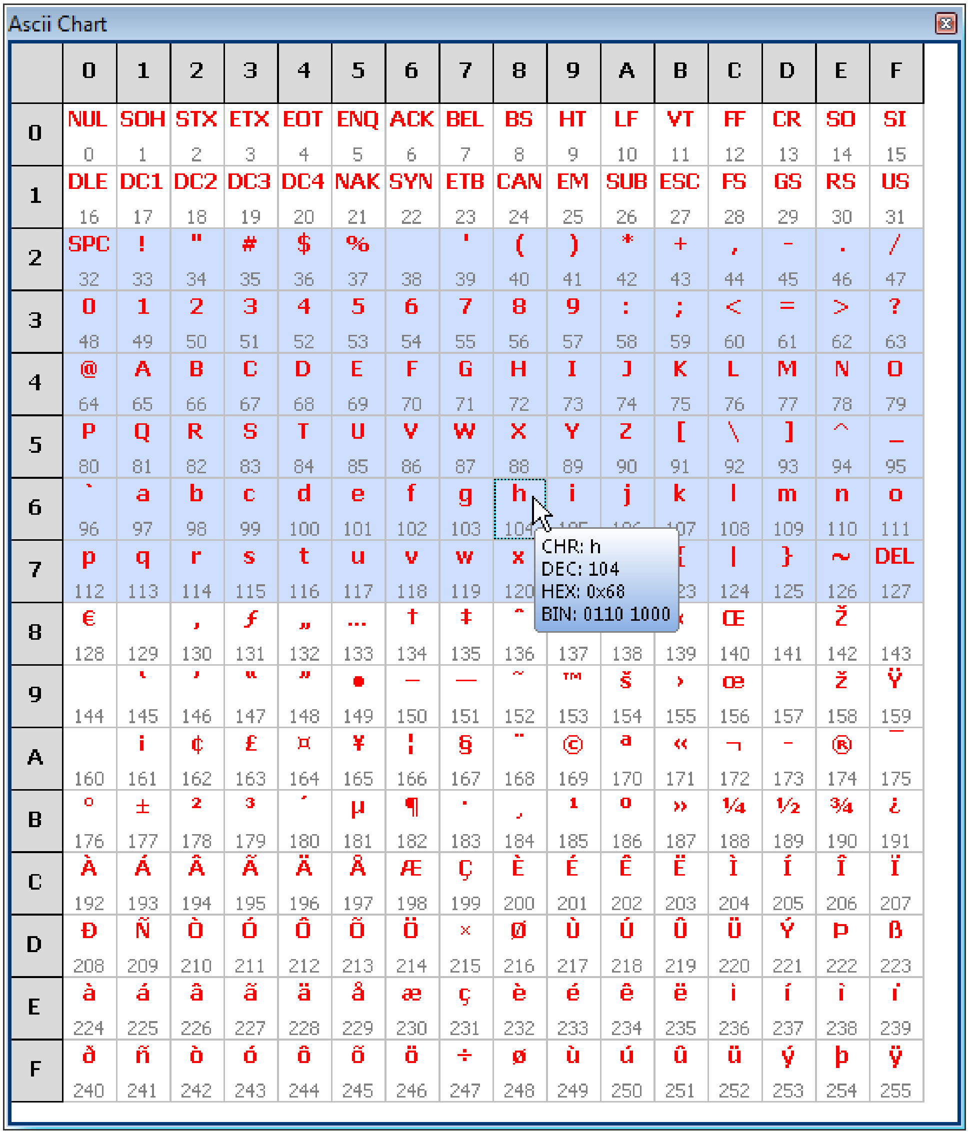
Input Format:
○ Dec
● Hex
○ Float
○ String

EEPROM Address:
Start Address: 0x FFFF
☑ AutoInc

Edit Value:
Hex No: 0x
Size: ○ Byte ○ Word ● DWord

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.

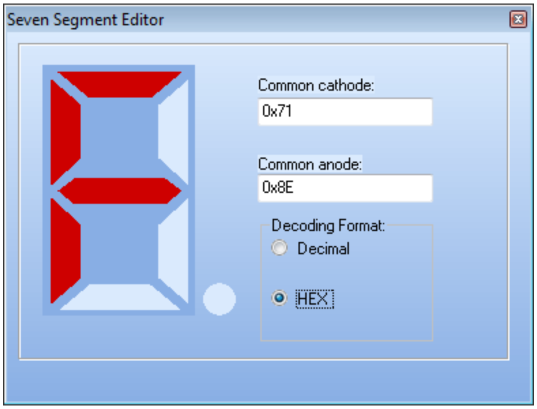


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
2	SPC 32	! 33	" 34	# 35	$ 36	% 37	38	' 39	(40	) 41	* 42	+ 43	, 44	- 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			23	| 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

Seven Segment Converter

The Seven Segment Convertor 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 Convertor** or by clicking the Seven Segment Convertor Icon from Tools toolbar.



Seven Segment Editor

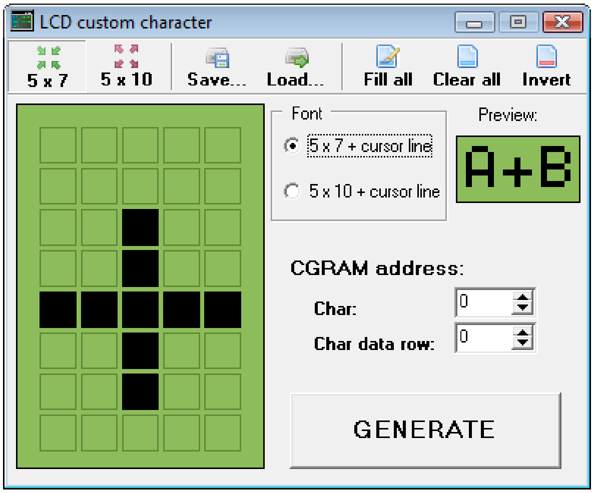
Common cathode: 0x71

Common anode: 0x8E

Decoding Format:
○ Decimal
● HEX

LCD Custom Character

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



LCD custom character

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

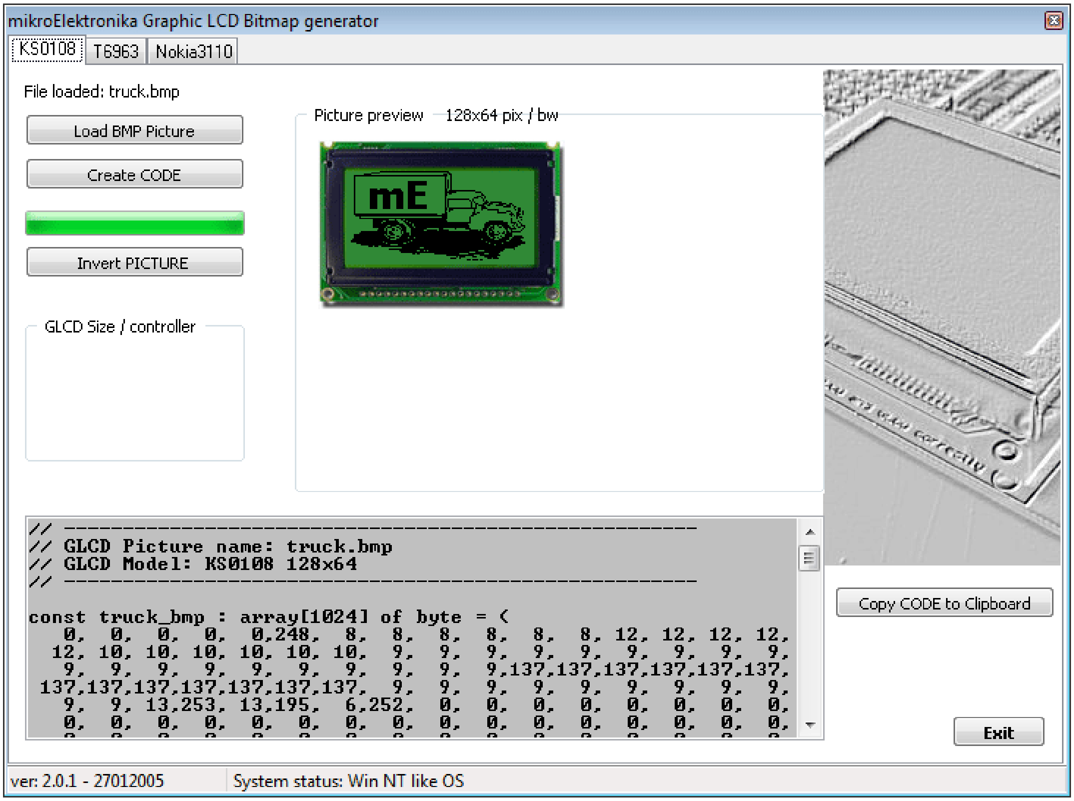
CGRAM address:

Char: 0

Char data row: 0

Graphic LCD Bitmap Editor

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



File loaded: truck.bmp

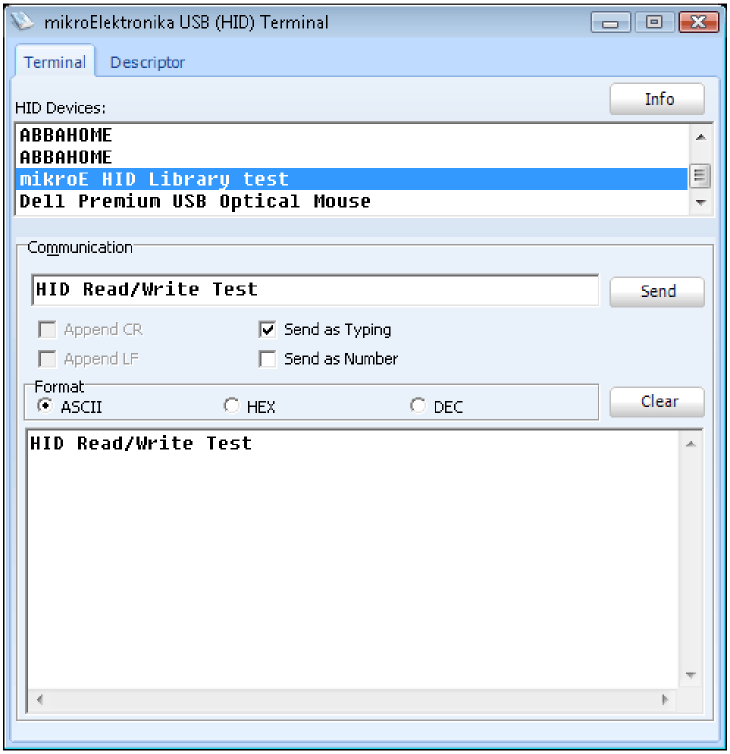
```
// ------------------------------------------------------------
// GLCD Picture name: truck.bmp
// GLCD Model: KS0108 128x64
// ------------------------------------------------------------

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

HID Terminal

The *mikroC PRO for PIC* includes the HID communication terminal for USB communication. You can launch it from the drop-down menu **Tools › HID Terminal**.



HID Devices:

```
ABBAHOME
ABBAHOME
mikroE HID Library test
Dell Premium USB Optical Mouse
```

HID Read/Write Test

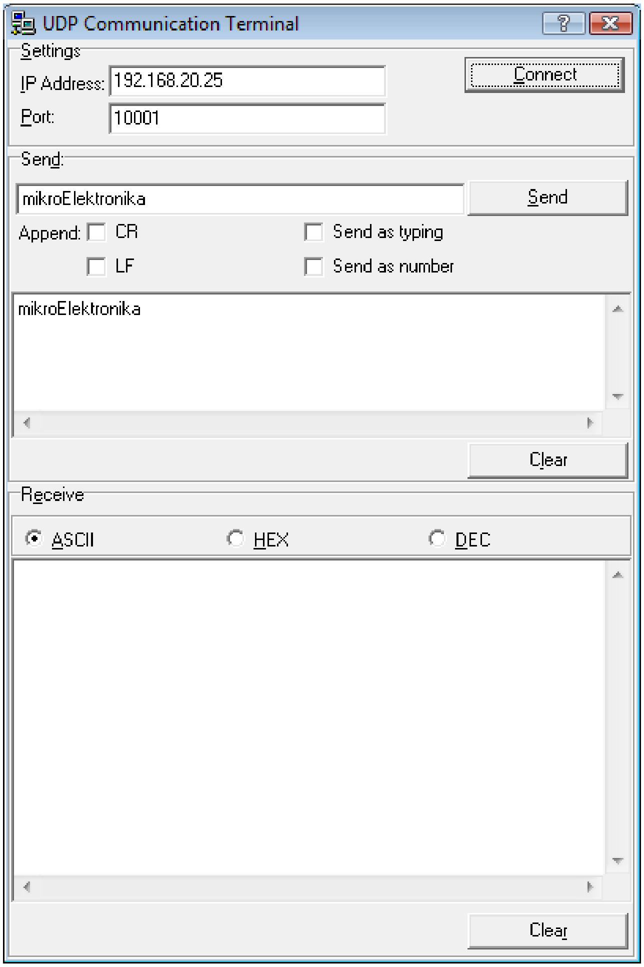
☐ Append CR
☑ Send as Typing
☐ Append LF
☐ Send as Number

Format
● ASCII ○ HEX ○ DEC

```
HID Read/Write Test
```

UDP Terminal

The *mikroC PRO for PIC* 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

Append: ☐ CR ☐ LF

☐ Send as typing

☐ Send as number

```
mikroElektronika
```

● ASCII ○ HEX ○ DEC

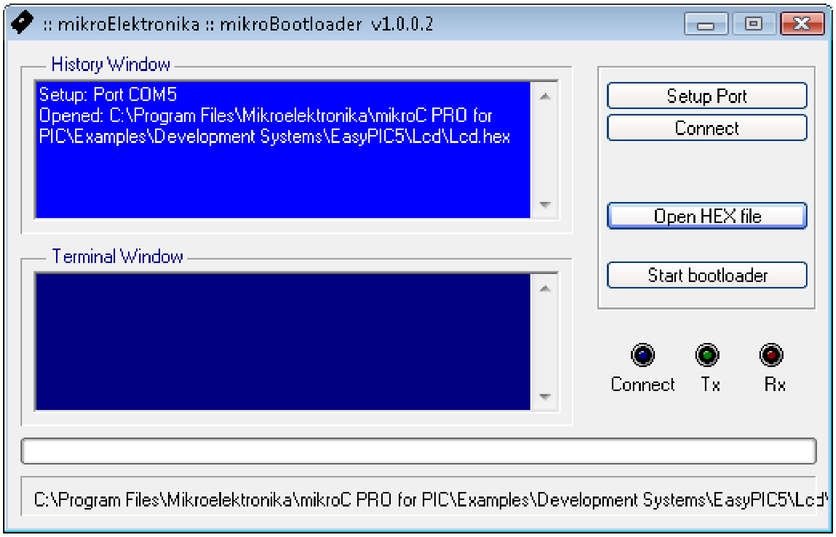
mikroBootloader

(From Microchip's document AN732) The PIC16F87X family of microcontrollers has the ability to write to their own program memory. This feature allows a small bootloader program to receive and write new firmware into memory. In its most simple form, the bootloader starts the user code running, unless it finds that new firmware should be downloaded. If there is new firmware to be downloaded, it gets the data and writes it into program memory. There are many variations and additional features that can be added to improve reliability and simplify the use of the bootloader.

Note: mikroBootloader can be used only with PIC MCUs that support flash write.

How to use mikroBootloader?

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



```
Setup: Port COM5
Opened: C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Lcd\Lcd.hex
```

```
C:\Program Files\Mikroelektronika\mikroC PRO for PIC\Examples\Development Systems\EasyPIC5\Lcd
```

Features

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

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

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

Integrating User Code and Boot Code

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

OPTIONS

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

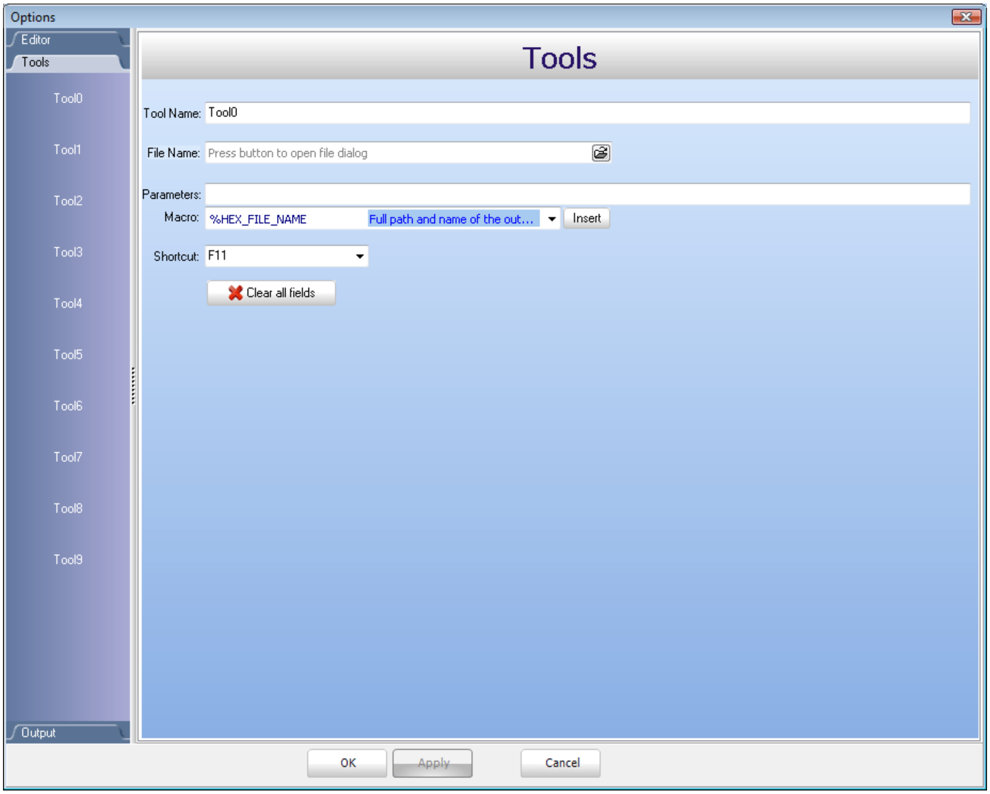
Code editor

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

Tools

The *mikroC PRO for PIC* 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

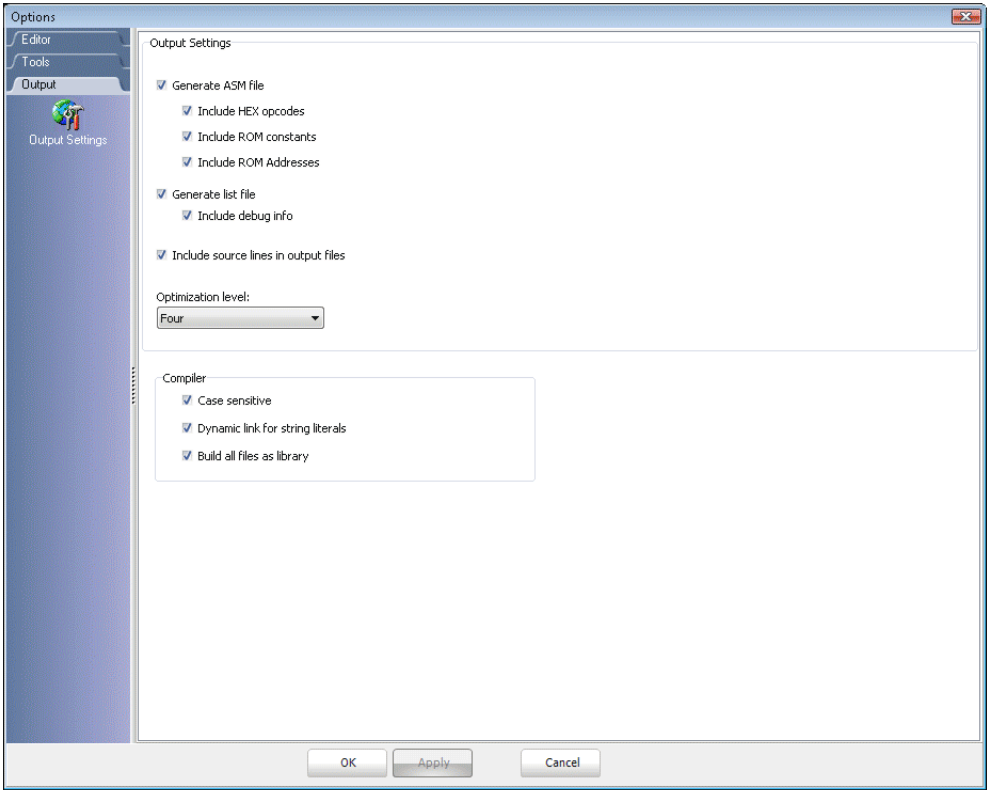
Parameters:

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

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 successfuly 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.

mikroC PRO for PIC COMMAND LINE OPTIONS

Usage: `mikroCPIC1618.exe [ -<opts> [ -<opts>]] [ <infile> [ -<opts>]] [ -<opts>]]` Infile can be of `*.c`, `*.mcl` and `*.pld` 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.
- `IF`: Add directory to the #include search 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.
- `D`: Build all files as libraries.
- `Y`: Dynamic link for string literals.
- `C`: Turn on case sensitivity.
- `UCD`: ICD build type.

Example:

```
     mikroCPIC1618.exe -MSF -DBG -p16F887 -ES -C -O11111114 -fo8 -
N"C:\Lcd\Lcd.mcppi" -SP"C:\Program Files\Mikroelektronika\mikroC PRO
for PIC\Defs\" -SP"C:\Program Files\Mikroelektronika\mikroC PRO for
PIC\Uses\P16\" -SP"C:\Lcd\" "Lcd.c" "__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.
- `p16F887`: MCU 16F887 selected.
- `C`: Turn on case sensitivity.
- `O11111114`: Miscellaneous output options.
- `fo10`: Set oscillator frequency [in MHz].
- `N"C:\Lcd\Lcd.mcppi" -SP"C:\Program Files\Mikroelektronika\mikroC PRO for PIC\defs\"`: Output files generated to file path specified by filename.
- `-SP"C:\Program Files\Mikroelektronika\mikroC PRO for PIC\defs\"`: Add directory to the search path list.
- `SP"C:\Program Files\Mikroelektronika\mikroC PRO for PIC \uses\"`: Add directory to the search path list.
- `-SP"C:\Lcd\"`: Add directory to the search path list.
- `"Lcd.c" "__Lib_Math.mcl" "__Lib_MathDouble.mcl" "__Lib_System.mcl" "__Lib_Delays.mcl" "__Lib_LcdConsts.mcl" "__Lib_Lcd.mcl"`: Specify input files.

PROJECTS

The *mikroC PRO for PIC* organizes applications into projects, consisting of a single project file (extension `.mcppi`) and one or more source files (extension). mikroC PRO for PIC 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,
- header files (*.h),
- binary files (*.mcl),
- image files,
- other files.

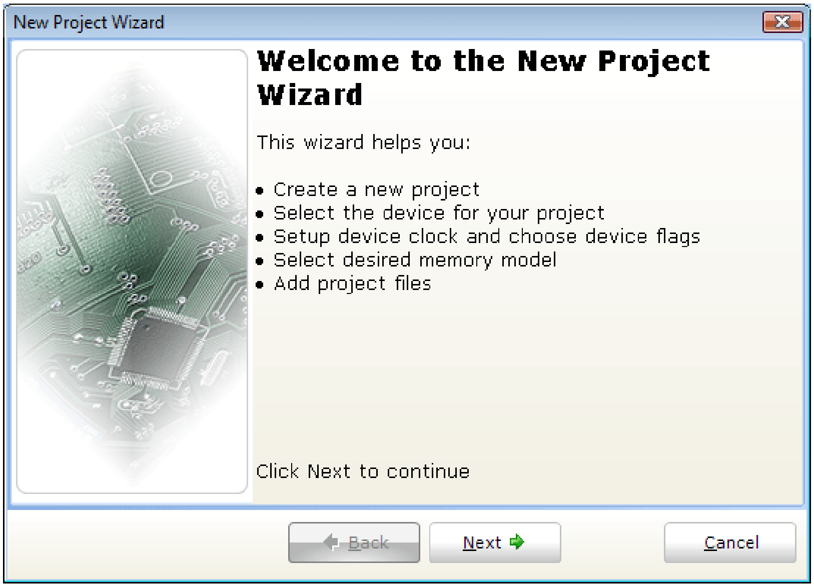
Note that the project does not include files in the same way as preprocessor does, see Add/Remove Files from Project.

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:



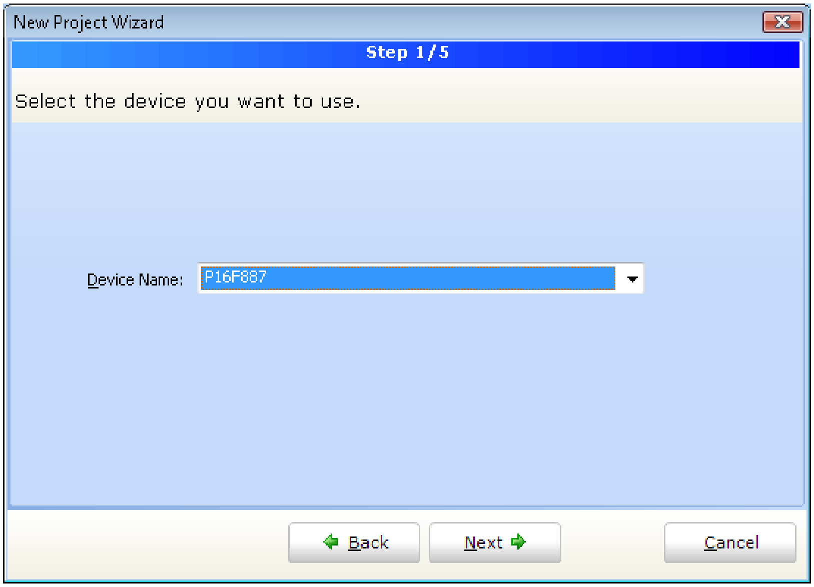
Welcome to the New Project Wizard

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

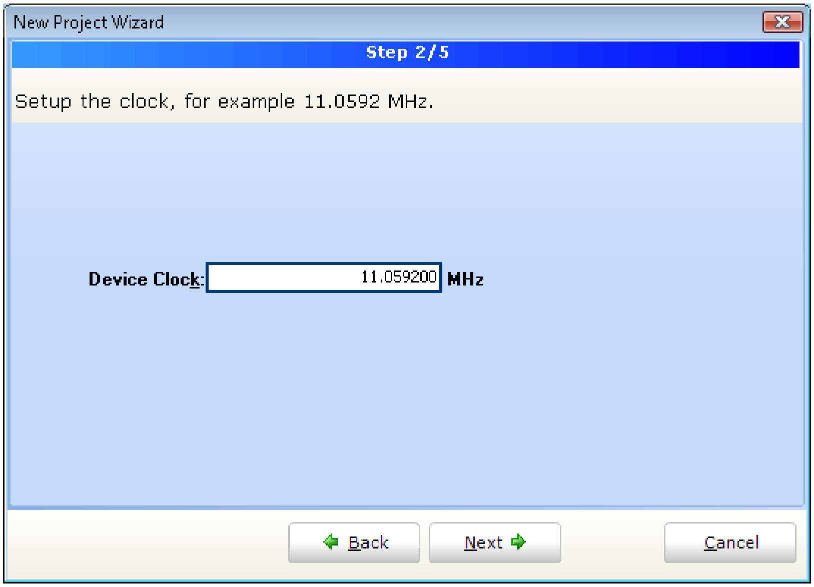
Step One - Select the device from the device drop-down list.



Select the device you want to use.

Device Name: P16F887

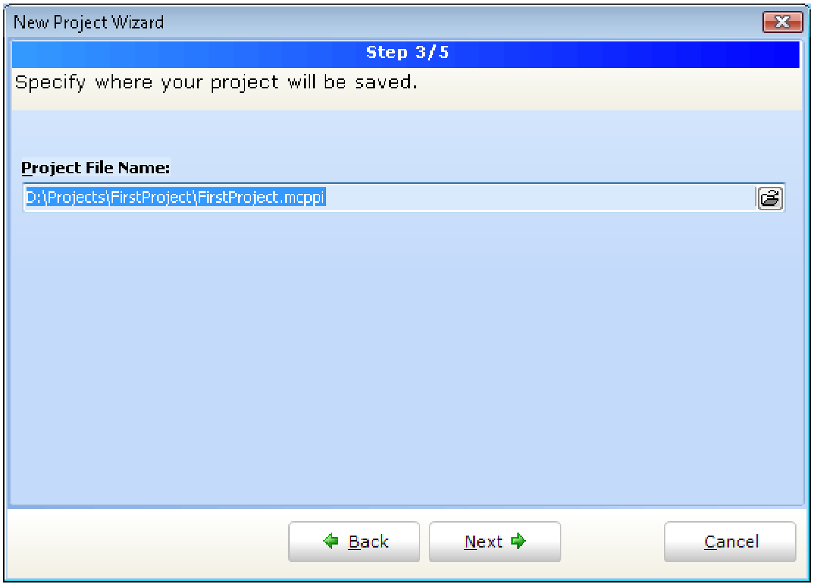
Step Two - Enter the oscillator frequency value.



Setup the clock, for example 11.0592 MHz.

Device Clock: 11.059200 MHz

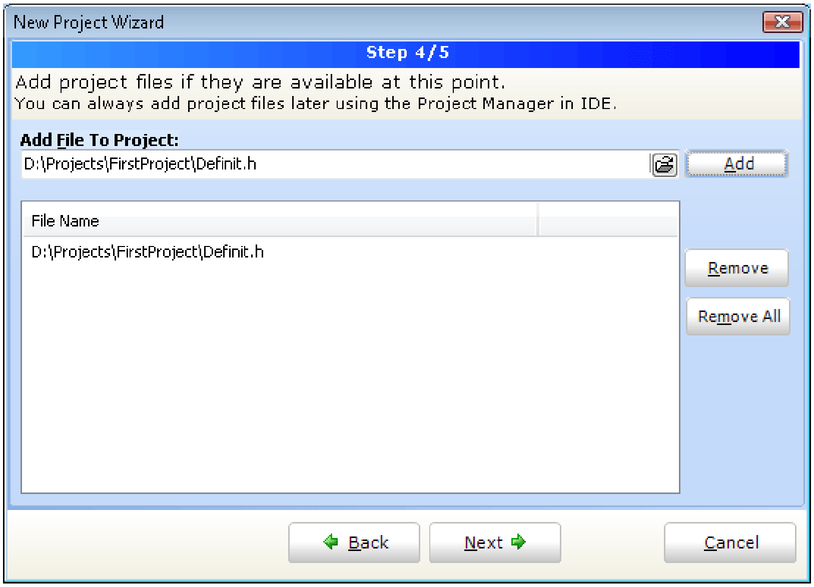
Step Three - Specify the location where your project will be saved.



Specify where your project will be saved.

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

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.

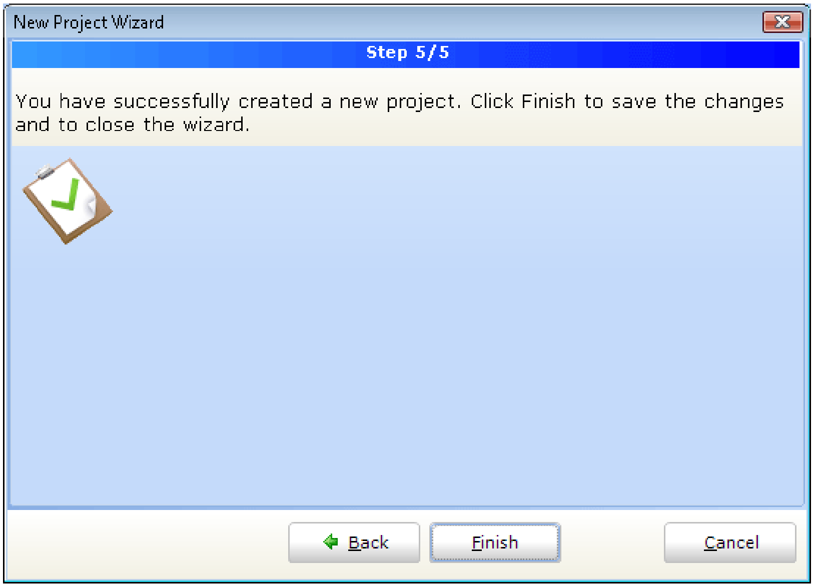


Add project files if they are available at this point.
You can always add project files later using the Project Manager in IDE.

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

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

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



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

Related topics: Project Manager, Project Settings

PROJECTS

The *mikroC PRO for PIC* organizes applications into projects, consisting of a single project file (extension `.mcppi`) and one or more source files (extension). *mikroC PRO for PIC* 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,
- header files (`*.h`),
- binary files (`*.mcl`),
- image files,
- other files.

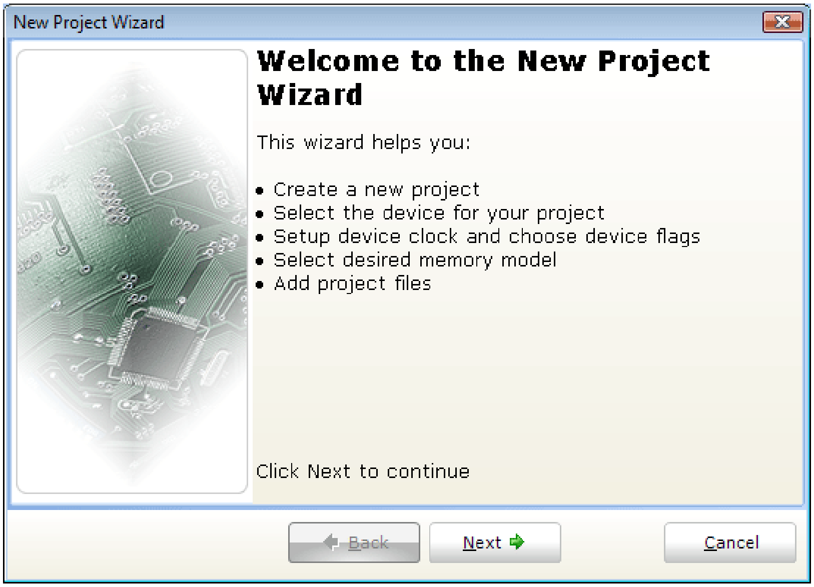
Note that the project does not include files in the same way as preprocessor does, see Add/Remove Files from Project.

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

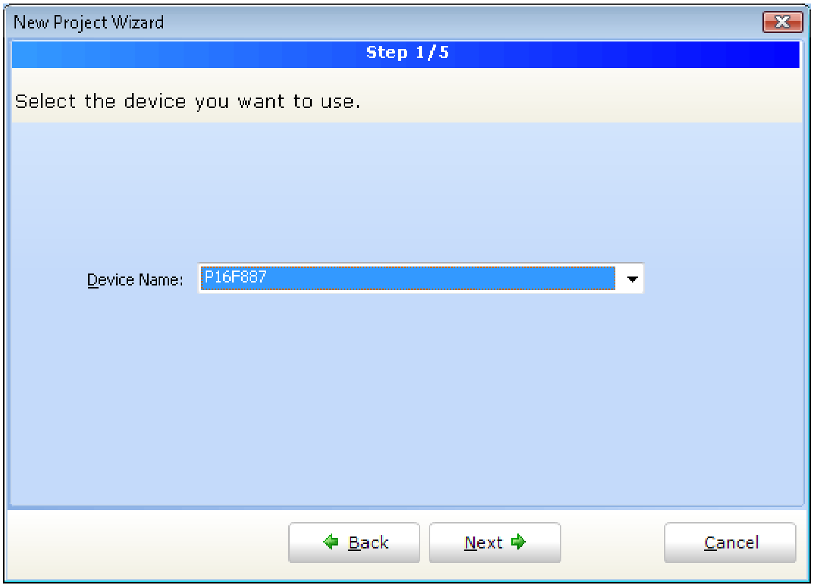
Welcome to the New Project Wizard

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.



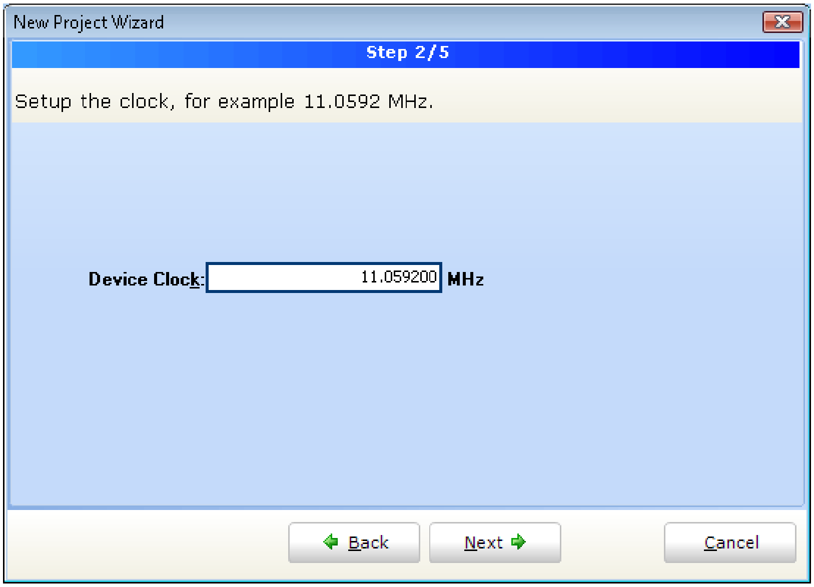
New Project Wizard

Step 1/5

Select the device you want to use.

Device Name: P16F887

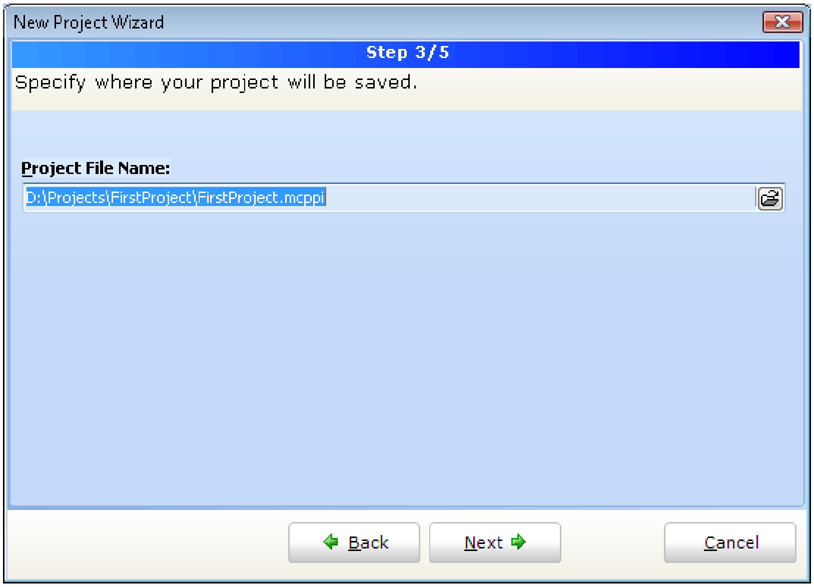
Step Two - Enter the oscillator frequency value.



Setup the clock, for example 11.0592 MHz.

Device Clock: 11.059200 MHz

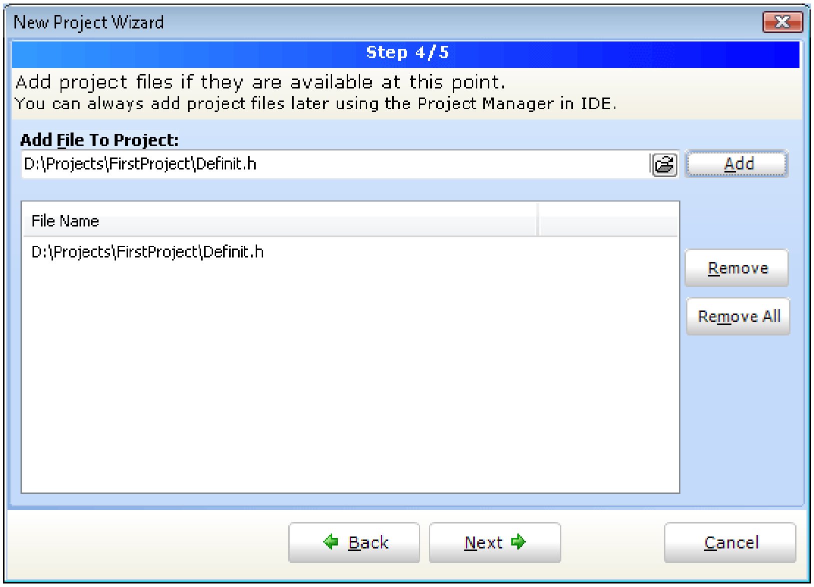
Step Three - Specify the location where your project will be saved.



Specify where your project will be saved.

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

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

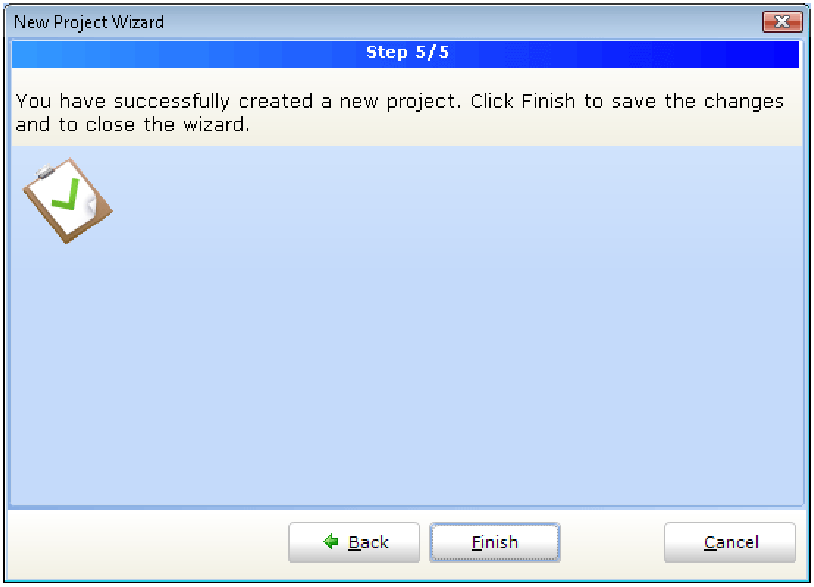
Step 4/5

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.h

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

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.

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. Also, you can change configuration bits of the selected chip in the Edit Project window.

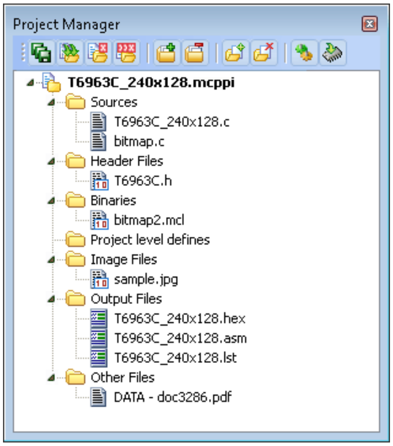
Managing Project Group

mikroC PRO for PIC 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 `.mpgroup`)

Add/Remove Files from Project

The project can contain the following file types:

- source files
- `.h` header 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.mcppi
  - Sources
    - T6963C_240x128.c
    - bitmap.c
  - Header Files
    - T6963C.h
  - 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 `.mcppi`).

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 .

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. A file must contain one definition per line in the following form:

```
<symbol>[ =[ <value>]]
<symbol  (a,b)>[ =[ <value>]]
```

Define a macro named symbol. To specify a value, use `=<value>`. If `=<value>` is omitted, 1 is assumed. Do not enter white-space characters immediately before the `"="`. If a white-space character is entered immediately after the `"="`, the macro is defined as zero token. This option can be specified repeatedly. Each appearance of symbol will be replaced by the value before compilation.
There are two predefined project level defines. See predefined project level defines

Note: For inclusion of the header files (extension `.h`), use the preprocessor directive `#include`. See File Inclusion for more information.

Related topics: Project Manager, Project Settings, Edit Project

SOURCE FILES

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

Use the preprocessor directive `#include` to include header files with the extension `.h`. Do not rely on the preprocessor to include source files other than headers — see Add/Remove Files from Project for more information.

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, will be created automatically. The *mikroC PRO for PIC* 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

1. 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.
2. 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 active.
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 active.
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 active.
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

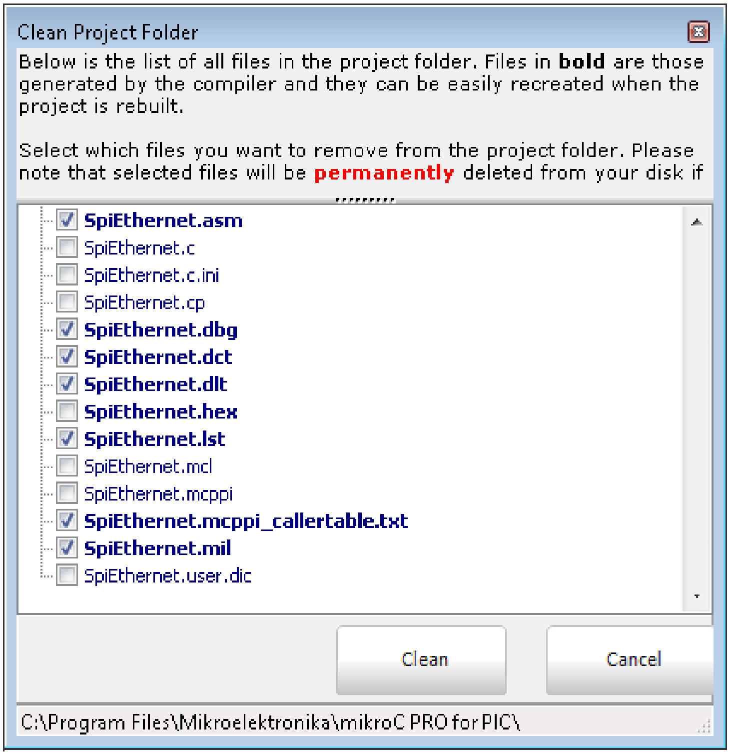
1. Make sure that the tab containing the file that you want to close is the active tab.
2. Select **File › Close** from the drop-down menu, or right click the tab of the file that you want to close and select **Close** option from the context menu.
3. If the file has been changed since it was last saved,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

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

CLEAN PROJECT FOLDER

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.c
- ☐ SpiEthernet.c.ini
- ☐ SpiEthernet.cp
- ☑ **SpiEthernet.dbg**
- ☑ **SpiEthernet.dct**
- ☑ **SpiEthernet.dlt**
- ☐ **SpiEthernet.hex**
- ☑ **SpiEthernet.lst**
- ☐ SpiEthernet.mcl
- ☐ SpiEthernet.mcppi
- ☑ **SpiEthernet.mcppi_callertable.txt**
- ☑ **SpiEthernet.mil**
- ☐ SpiEthernet.user.dic

C:\Program Files\Mikroelektronika\mikroC PRO for PIC\

Related topics: Customizing Projects

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 *mikroC PRO for PIC* will generate output files.

Output Files

Upon successful compilation, the *mikroC PRO for PIC* will generate output files in the project folder (folder which contains the project file `.mcppi`). Output files are summarized in the table below:

Format	Description	File Type
Intel HEX	Intel style hex records. Use this file to program PIC MCU.	`.hex`
Binary	mikro Compiled Library. Binary distribution of application that can be included in other projects.	`.mcl`
List File	Overview of PIC 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 *mikroC PRO for PIC*, 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

ERROR MESSAGES

Compiler Error Messages:

- Syntax error: Expected `[%s]` but `[%s]` found
- Array element cannot be function
- Function cannot return array
- Inconsistent storage class
- Inconsistent type
- `[%s]` tag redefined `[%s]`
- Illegal typecast `[%s]` `[%s]`
- `"%s"` is not valid identifier
- Invalid statement
- Constant expression required
- Internal error `[%s]`
- Too many actual parameters
- Not enough parameters.
- Invalid expression
- Identifier expected, but `[%s]` found
- Operator `[%s]` is not applicable to these operands `[%s]`
- Assigning to non-lvalue `[%s]`
- Cannot cast `[%s]` to `[%s]`
- Cannot assign `[%s]` to `[%s]`
- Lvalue required
- Pointer required
- Argument is out of range
- Undeclared identifier `[%s]` in expression
- Too many initializers
- Cannot establish this baud rate at `[%s]` MHz clock
- Stack overflow
- Invalid operator `[%s]`
- Expected variable, but constant `[%s]` found
- Expected constant, but `[%s]` found
- `[%s]` cannot be used outside a loop
- Unknown type `[%s]`
- Variable `[%s]` is redeclared
- Undeclared identifier `[%s]`
- Output limit has raised 2K words
- `[%s]` has already been declared `[%s]`
- Type mismatch: expected `[%s]`, but `[%s]` found
- File `[%s]` not found `[%s]`
- There is not enough RAM space for all variables
- There is not enough ROM space
- Invalid type in Array
- Division by zero
- Incompatible types: `[%s]` `[%s]`
- Too many characters

- Assembler instruction `[%s]` was not found
- Project name must be specified
- Unknown command line Option: `[%s]`
- File extension missing: `[%s]`
- Bad FO argument: `[%s]`
- Preprocessor exited with error code `[%s]`
- Bad absolute address `[%s]`
- Recursion or cross-calling of `[%s]`
- Reentrancy is not allowed: function`[%s]` called from two threads
- no files specified
- Device parameter missing (for example -P16F...)
- Invalid parameter string
- Project name must be set
- Specifier needed
- `[%s]` not found `[%s]`
- Index out of bounds
- Array dimension must be greater than 0
- Const expression expected
- Integer const expected
- Recursion in definition
- Array corrupted
- Arguments cannot be of void type
- Arguments cannot have explicit memory specificator
- Bad storage class
- Pointer to function required
- Function required
- Illegal pointer conversion to double
- Integer type needed
- Members cannot have memory specifier
- Members cannot be of bit or sbit type
- Too many initializers
- Too many initializers of subaggregate
- Already used `[%s]`
- Illegal expression with void
- Address must be greater than 0
- `[%s]` Identifier redefined
- User abort
- Expression must be greater than 0
- Invalid declarator expected `"("` or identifier
- typdef name redefined: `[%s]`
- Declarator error
- Specifer/qualifier list expected
- `[%s]` already used
- ILevel can be used only with interrupt service routines
- ; expected, but `[%s]` found

- Expected `"{ "`
- `[ %s]` Identifier redefined
- `"("` expected, but `[ %s]` found
- `")"` expected, but `[ %s]` found
- `"case"` out of switch
- `":"` expected, but `[ %s]` found
- `"default"` label out of switch
- switch expression must evaluate to integral type
- while expected, but `[ %s]` found
- void func cannot return values
- `"continue"` outside of loop
- Unreachable code
- Label redefined
- void type in expression
- Too many chars
- Unresolved type
- Arrays of objects containing zero-size arrays are illegal
- Invalid enumerator
- ILevel can be used only with interrupt service routines
- ILevel value must be integral constant
- ILevel out of range `"0..4"`
- `"} "` expected `[ %s]` found
- `")"` expected, but `[ %s]` found
- `"break"` outside of loop or switch
- Empty char
- Nonexistent field `[ %s]`
- Illegal char representation: `[ %s]`
- Initializer syntax error: multidimensional array missing subscript
- Too many initializers of subaggregate
- At least one Search Path must be specified
- Not enough RAM for call stack
- Demo Limit
- Parameter `[ %s]` must not be of bit or sbit type
- Function must not have return value of bit or sbit type

Compiler Warning Messages:

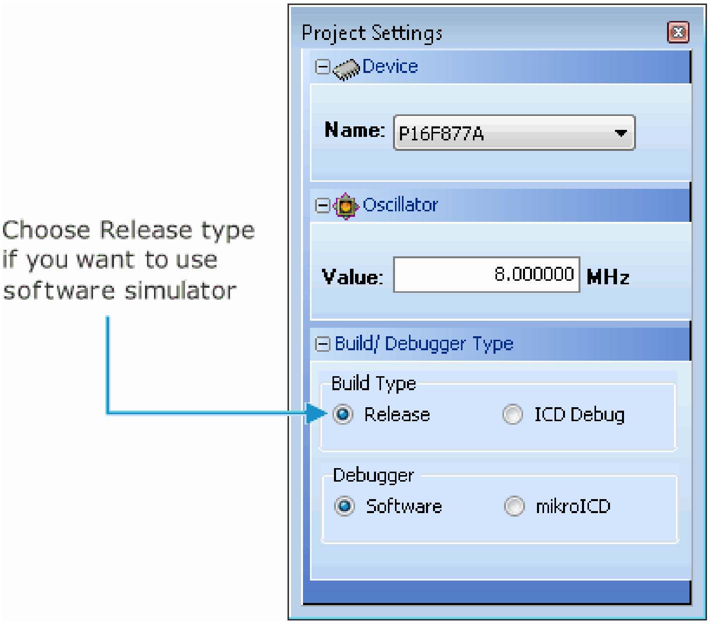
- Bad or missing fosc parameter. Default value 8MHz used
- Specified search path does not exist: `[%s]`
- Specified include path does not exist: `[%s]`
- Result is not defined in function: `[%s]`
- Initialization of extern object `[%s]`
- Suspicious pointer conversion
- Implicit conversion of pointer to `int`
- Unknown pragma line ignored: `[%s]`
- Implicit conversion of `int` to `ptr`
- Generated baud rate is `[%s]` bps (error = `[%s]` percent)
- Unknown memory model `[%s]` , small memory model used instead
- IRP bit must be set manually for indirect access to `[%s]` variable
- Variable `[%s]` has been declared, but not used'
- Illegal file type: `[%s]`

Linker Error Messages:

- Redefinition of `[%s]` already defined in `[%s]`
- main function is not defined
- System routine is not found for initialization of: `[%s]`
- Bad aggregate definition `[%s]`
- Unresolved extern `[%s]`
- Bad function absolute address `[%s]`
- Not enough RAM `[%s]`

SOFTWARE SIMULATOR OVERVIEW

The Source-level Software Simulator is an integral component of the *mikroC PRO for PIC* environment. It is designed to simulate operations of the PIC MCUs and assist the users in debugging C code written for these devices.
Upon completion of writing your program, choose **Release** build Type in the Project Settings window:



Choose Release type if you want to use software simulator

Name: P16F877A

Value: 8.000000 MHz

Build Type
● Release ○ ICD Debug

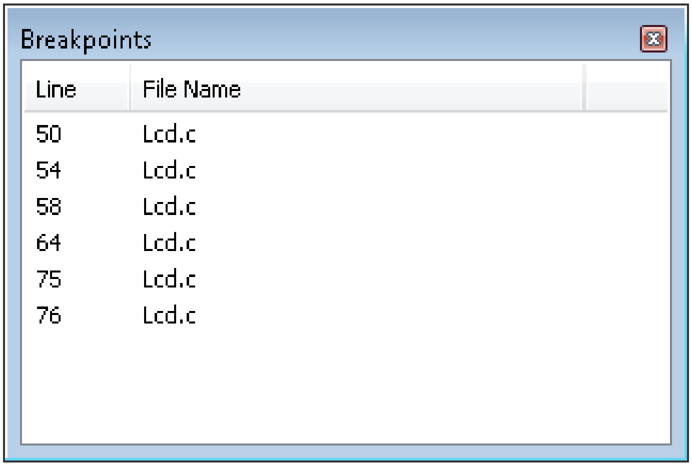
Debugger
● Software ○ mikroICD

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 PIC device behavior, i.e. it doesn't update timers, interrupt flags, etc.

Breakpoints Window

The Breakpoints window manages the list of currently set breakpoints in the project. Doubleclicking the desired breakpoint will cause cursor to navigate to the corresponding location in source code.



Breakpoints

Line	File Name
50	Lcd.c
54	Lcd.c
58	Lcd.c
64	Lcd.c
75	Lcd.c
76	Lcd.c

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:

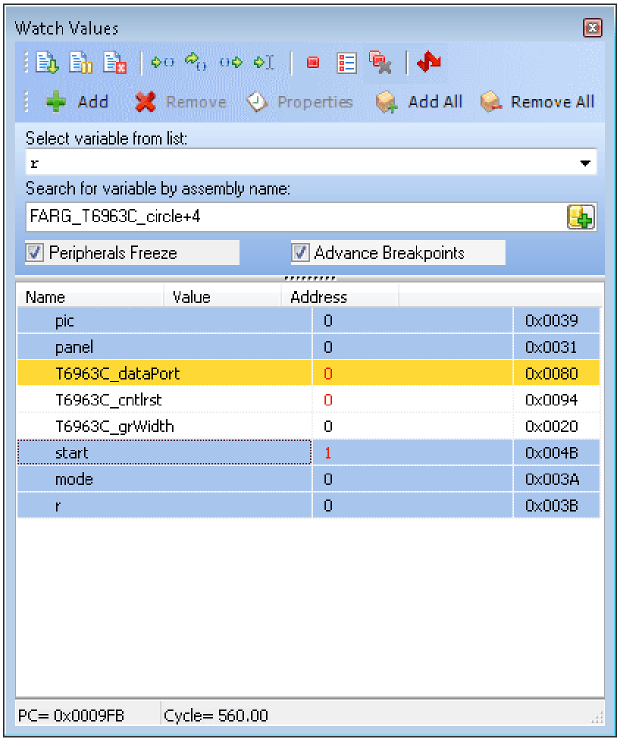
- by its real name (variable's name in "C" code). Just select desired variable/register from ***Select variable from list*** drop-down menu and click the Add Button Add .
- by its name ID (assembly variable name). Simply type name ID of they variable/register you want to display into **Search the variable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



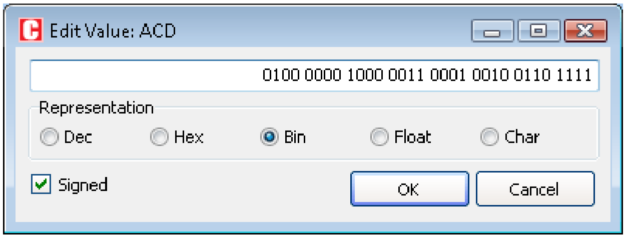
Select variable from list:
r
Search for variable by assembly name:
FARG_T6963C_circle+4
☑ Peripherals Freeze
☑ Advance Breakpoints

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

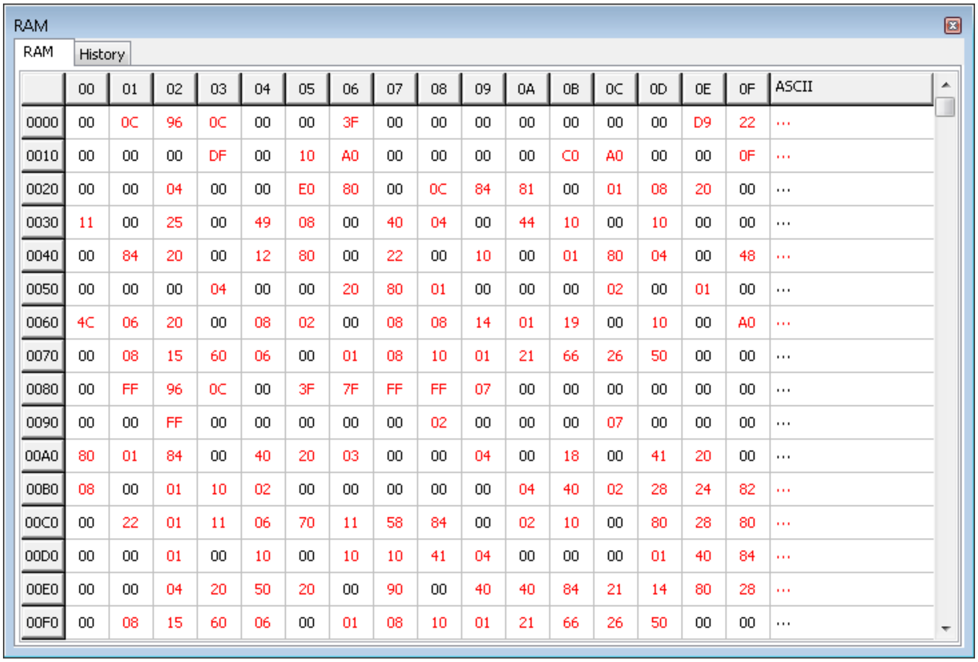
○ Dec ○ Hex ● Bin ○ Float ○ Char

☑ Signed

View RAM Window

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

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



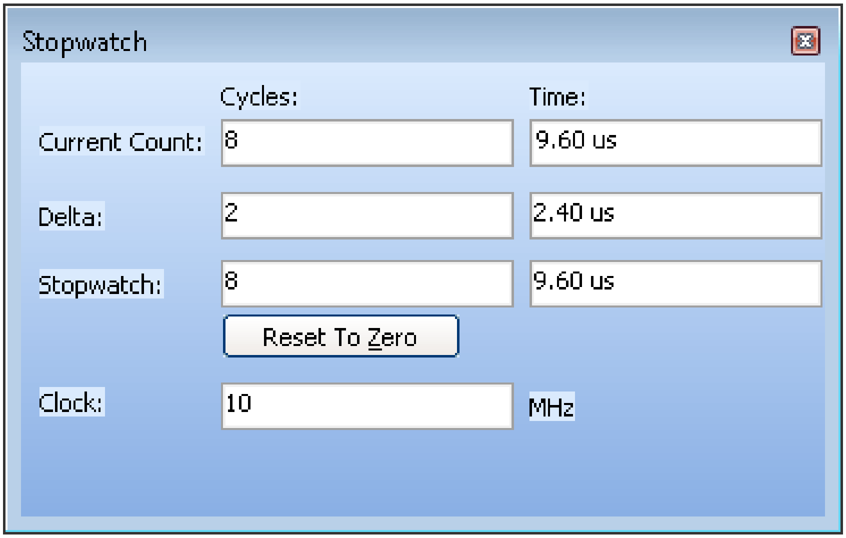
	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	0C	96	0C	00	00	3F	00	00	00	00	00	00	00	D9	22	...
0010	00	00	00	DF	00	10	A0	00	00	00	00	C0	A0	00	00	0F	...
0020	00	00	04	00	00	E0	80	00	0C	84	81	00	01	08	20	00	...
0030	11	00	25	00	49	08	00	40	04	00	44	10	00	10	00	00	...
0040	00	84	20	00	12	80	00	22	00	10	00	01	80	04	00	48	...
0050	00	00	00	04	00	00	20	80	01	00	00	00	02	00	01	00	...
0060	4C	06	20	00	08	02	00	08	08	14	01	19	00	10	00	A0	...
0070	00	08	15	60	06	00	01	08	10	01	21	66	26	50	00	00	...
0080	00	FF	96	0C	00	3F	7F	FF	FF	07	00	00	00	00	00	00	...
0090	00	00	FF	00	00	00	00	00	02	00	00	00	07	00	00	00	...
00A0	80	01	84	00	40	20	03	00	00	04	00	18	00	41	20	00	...
00B0	08	00	01	10	02	00	00	00	00	00	04	40	02	28	24	82	...
00C0	00	22	01	11	06	70	11	58	84	00	02	10	00	80	28	80	...
00D0	00	00	01	00	10	00	10	10	41	04	00	00	00	01	40	84	...
00E0	00	00	04	20	50	20	00	90	00	40	40	84	21	14	80	28	...
00F0	00	08	15	60	06	00	01	08	10	01	21	66	26	50	00	00	...

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

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

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

1. Create a new C source file, see Managing Source Files

2. Save the file in one of the subfolders of the compiler's Uses folder:

   ```
   DriveName:\Program Files\Mikroelektronika\mikroC PRO for
   PIC\Uses\P16\
   DriveName:\Program Files\Mikroelektronika\mikroC PRO for
   PIC\Uses\P18\
   ```

   If you are creating library for PIC16 MCU family the file should be saved in P16 folder.
   If you are creating library for PIC18 MCU family the file should be saved in P18 fodler.
   If you are creating library for PIC16 and PIC18 MCU families the file should be saved in both folders.

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 built 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 ...`\mikroC PRO for PIC\Uses\` 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\mikroC PRO for
   PIC\Defs\
   ```

   and it is named `MCU_NAME.mlk,` for example `16F887.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 16F887 is different from UART library for 18F4520 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.

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

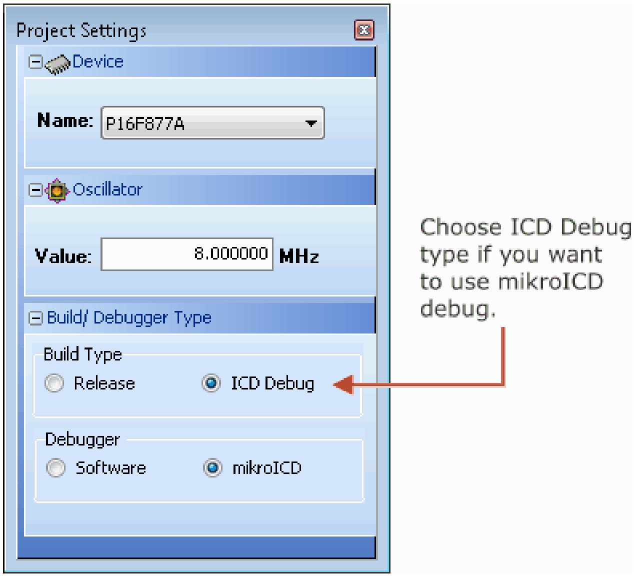
CHAPTER 3

MIKROICD (IN-CIRCUIT DEBUGGER)

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

Step No. 1

If you have appropriate hardware and software for using mikroICD, then, upon completion of writing your program, you will have to choose **ICD Debug** build type.



Name: P16F877A

Value: 8.000000 MHz

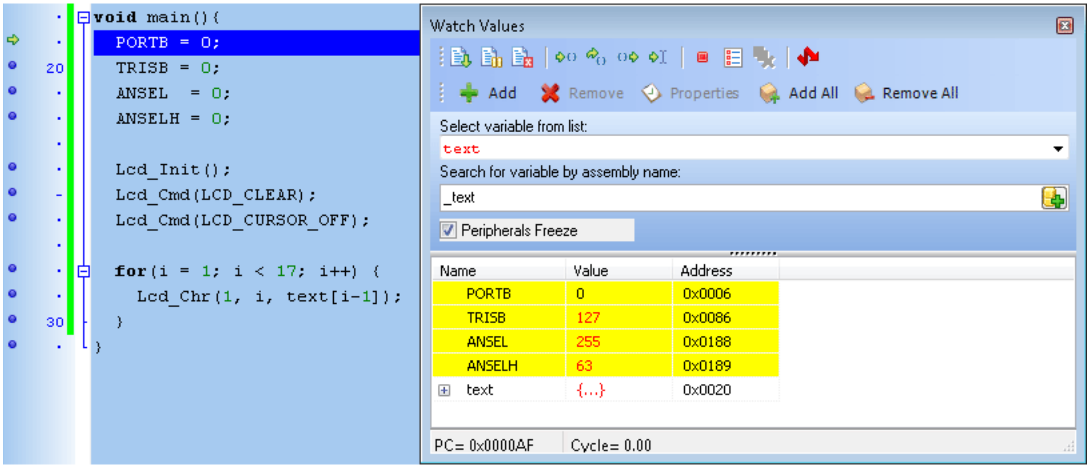
Build Type
○ Release ● ICD Debug

Debugger
○ Software ● mikroICD

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

Step No. 2

You can run the mikroICD by selecting **Run › Debug** from the drop-down menu, or by clicking Debug Icon . Starting the Debugger makes more options available: Step Into, Step Over, Run to Cursor, etc. Line that is to be executed is color highlighted (blue by default). There is also notification about program execution and it can be found on Watch Window (yellow status bar). Note that some functions take time to execute, so running of program is indicated on Watch Window.



```
void main(){
  PORTB = 0;
  TRISB = 0;
  ANSEL  = 0;
  ANSELH = 0;

  Lcd_Init();
  Lcd_Cmd(LCD_CLEAR);
  Lcd_Cmd(LCD_CURSOR_OFF);

  for(i = 1; i < 17; i++) {
    Lcd_Chr(1, i, text[i-1]);
  }
}
```

Select variable from list: text

Search for variable by assembly name: _text

☑ Peripherals Freeze

Name	Value	Address
PORTB	0	0x0006
TRISB	127	0x0086
ANSEL	255	0x0188
ANSELH	63	0x0189
⊞ text	{...}	0x0020

PC= 0x0000AF Cycle= 0.00

mikroICD Debugger Options

Name	Description	Function Key
`Debug`	Start Debugger.	[F9]
`Run/Pause Debugger`	Run or pause Debugger.	[F6]
`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]
`Step Into`	Execute the current C (single or multi–cycle) instruction, then halt. If the instruction is a routine call, enter the routine and halt at the first instruction following the call.	[F7]
`Step Over`	Execute the current C (single or multi–cycle) instruction, then halt. If the instruction is a routine call, skip it and halt at the first instruction following the call.	[F8]
`Flush RAM`	Flush current PIC RAM. All variable values will be changed according to values from watch window.	N/A
`Disassembly View`	Toggle between disassembly and C source view.	[Alt+D]

mikroICD Debugger Examples

Here is a step by step mikroICD Debugger Example.

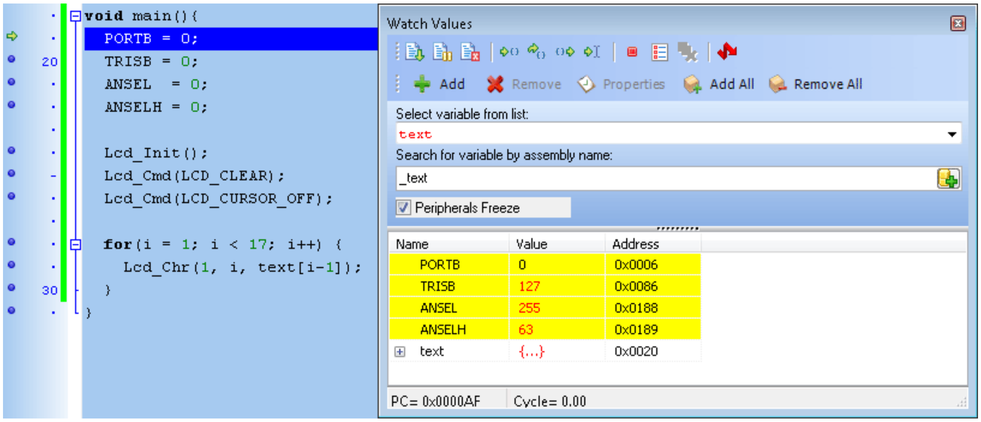
Step No.1

First you have to write a program. We will show how mikroICD works using this example:

```
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

char text[17] = "mikroElektronika";
char i;

void main(){
  PORTB = 0;
  TRISB = 0;
  ANSEL  = 0;
  ANSELH = 0;

  Lcd_Init();
  Lcd_Cmd(_LCD_CLEAR);
  Lcd_Cmd(_LCD_CURSOR_OFF);

  for(i = 1; i < 17; i++) {
    Lcd_Chr(1, i, text[i-1]);
  }
}
```

Step No. 2

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



```
void main(){
  PORTB = 0;
  TRISB = 0;
  ANSEL  = 0;
  ANSELH = 0;

  Lcd_Init();
  Lcd_Cmd(LCD_CLEAR);
  Lcd_Cmd(LCD_CURSOR_OFF);

  for(i = 1; i < 17; i++) {
    Lcd_Chr(1, i, text[i-1]);
  }
}
```

Select variable from list: text

Search for variable by assembly name: _text

☑ Peripherals Freeze

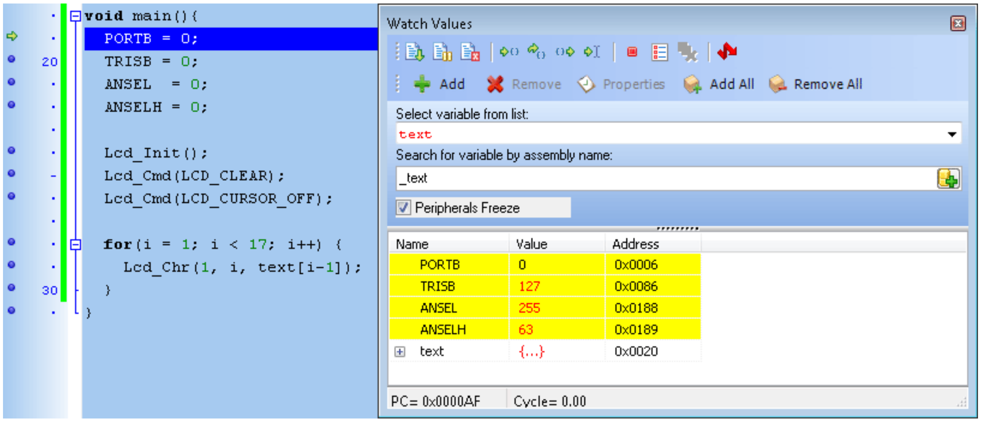
Name	Value	Address
PORTB	0	0x0006
TRISB	127	0x0086
ANSEL	255	0x0188
ANSELH	63	0x0189
⊞ text	{...}	0x0020

PC= 0x0000AF Cycle= 0.00

Step No. 3

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

All changes are read from PIC and loaded into Watch Window. Note that **PORTB, TRISB, ANSEL** and **ANSELH** changed its values. 255 to 0.



```
void main(){
  PORTB = 0;
  TRISB = 0;
  ANSEL  = 0;
  ANSELH = 0;

  Lcd_Init();
  Lcd_Cmd(LCD_CLEAR);
  Lcd_Cmd(LCD_CURSOR_OFF);

  for(i = 1; i < 17; i++) {
    Lcd_Chr(1, i, text[i-1]);
  }
}
```

Select variable from list: text

Search for variable by assembly name: _text

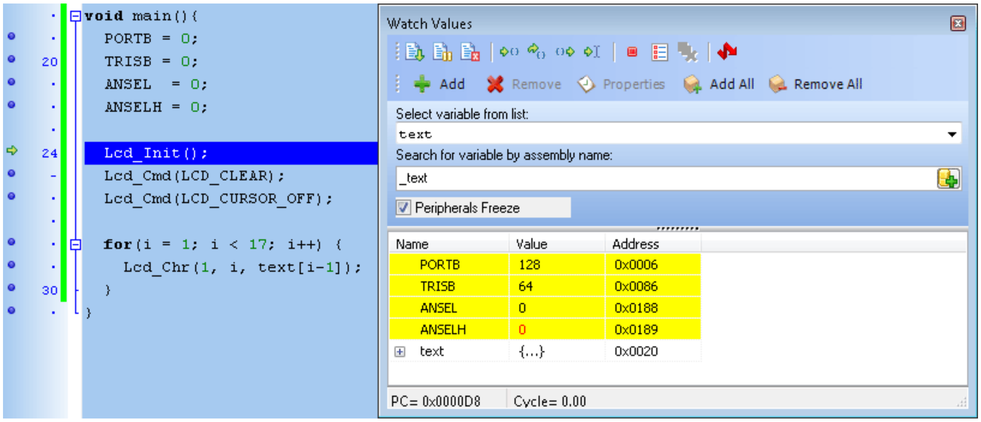
☑ Peripherals Freeze

Name	Value	Address
PORTB	0	0x0006
TRISB	127	0x0086
ANSEL	255	0x0188
ANSELH	63	0x0189
⊞ text	{...}	0x0020

PC= 0x0000AF Cycle= 0.00

Step No. 4

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



```
void main(){
  PORTB = 0;
  TRISB = 0;
  ANSEL  = 0;
  ANSELH = 0;

  Lcd_Init();
  Lcd_Cmd(LCD_CLEAR);
  Lcd_Cmd(LCD_CURSOR_OFF);

  for(i = 1; i < 17; i++) {
    Lcd_Chr(1, i, text[i-1]);
  }
}
```

Select variable from list: text

Search for variable by assembly name: _text

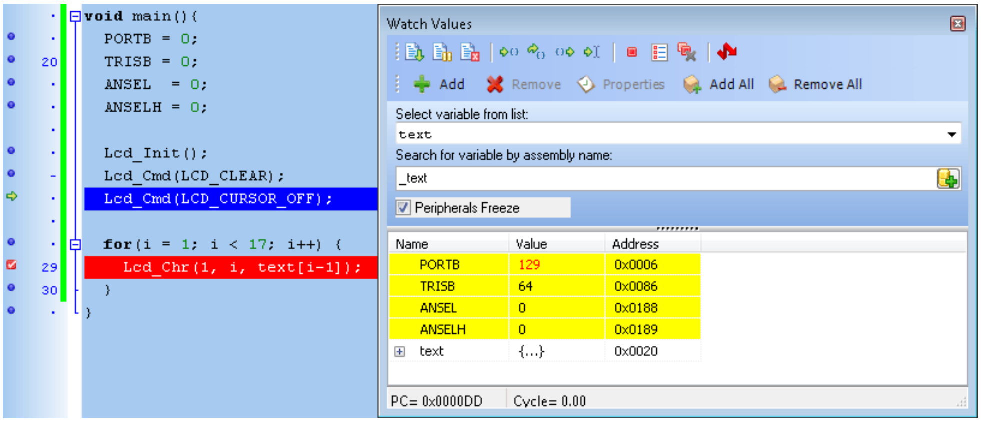
☑ Peripherals Freeze

Name	Value	Address
PORTB	128	0x0006
TRISB	64	0x0086
ANSEL	0	0x0188
ANSELH	0	0x0189
⊞ text	{...}	0x0020

PC= 0x0000D8 Cycle= 0.00

Step No. 5

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



```
void main(){
  PORTB = 0;
  TRISB = 0;
  ANSEL  = 0;
  ANSELH = 0;

  Lcd_Init();
  Lcd_Cmd(LCD_CLEAR);
  Lcd_Cmd(LCD_CURSOR_OFF);

  for(i = 1; i < 17; i++) {
    Lcd_Chr(1, i, text[i-1]);
  }
}
```

Select variable from list: text

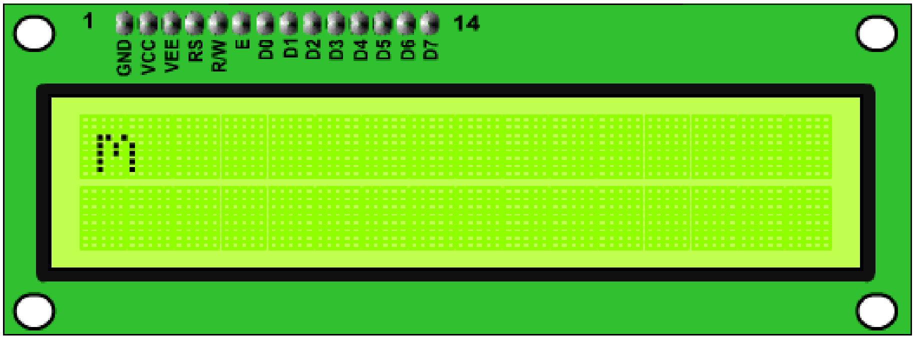
Search for variable by assembly name: _text

☑ Peripherals Freeze

Name	Value	Address
PORTB	129	0x0006
TRISB	64	0x0086
ANSEL	0	0x0188
ANSELH	0	0x0189
⊞ text	{...}	0x0020

PC= 0x0000DD Cycle= 0.00

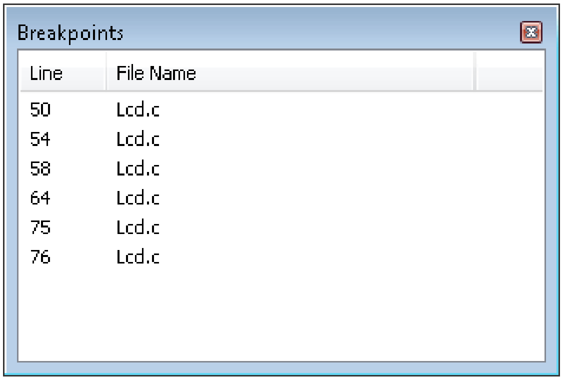
Breakpoints has been separated into two groups. There are hardware and software break points. Hardware breakpoints are placed in PIC and they provide fastest debug. Number of hardware breakpoints is limited (1 for P16 and 1 or 3 for P18). If all hardware brekpoints are used, next breakpoints that will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



mikroICD (In-Circuit Debugger) Overview

Breakpoints Window

The Breakpoints window manages the list of currently set breakpoints in the project. Doubleclicking the desired breakpoint will cause cursor to navigate to the corresponding location in source code.

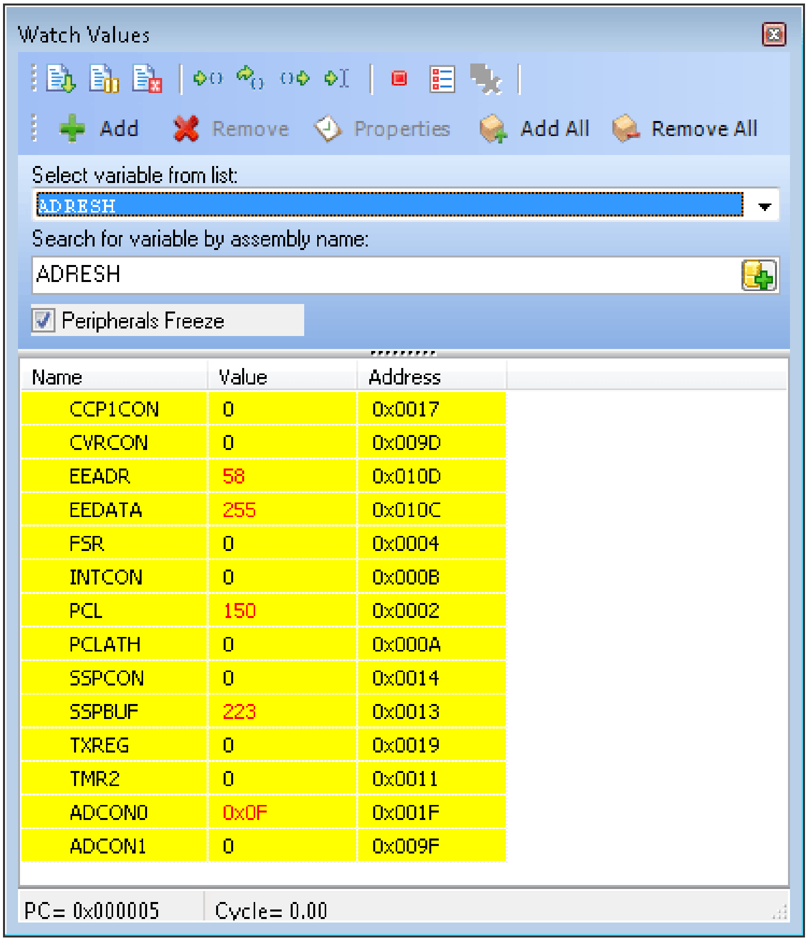


Line	File Name
50	Lcd.c
54	Lcd.c
58	Lcd.c
64	Lcd.c
75	Lcd.c
76	Lcd.c

Watch Window

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

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



Select variable from list: ADRESH

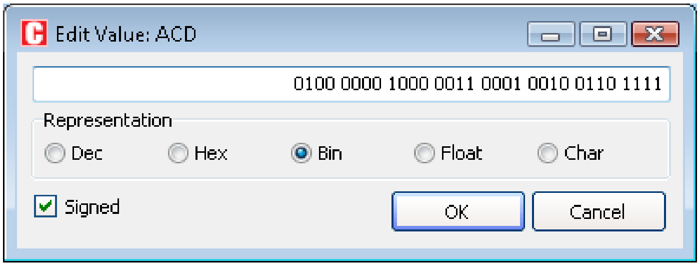
Search for variable by assembly name: ADRESH

☑ Peripherals Freeze

Name	Value	Address
CCP1CON	0	0x0017
CVRCON	0	0x009D
EEADR	58	0x010D
EEDATA	255	0x010C
FSR	0	0x0004
INTCON	0	0x000B
PCL	150	0x0002
PCLATH	0	0x000A
SSPCON	0	0x0014
SSPBUF	223	0x0013
TXREG	0	0x0019
TMR2	0	0x0011
ADCON0	0x0F	0x001F
ADCON1	0	0x009F

PC= 0x000005 Cycle= 0.00

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



0100 0000 1000 0011 0001 0010 0110 1111

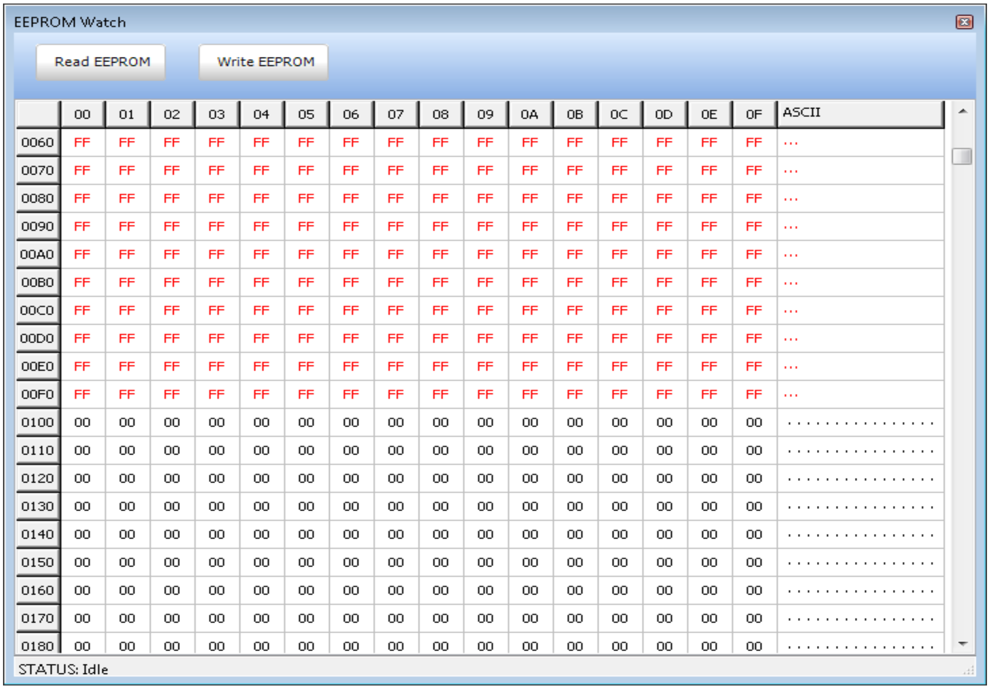
○ Dec ○ Hex ● Bin ○ Float ○ Char

☑ Signed

EEPROM Watch Window

mikroICD EEPROM Watch Window is available from the drop-down menu, **View › Debug Windows › View EEPROM**.

The EEPROM Watch window shows current values written into PIC internal EEPROM memory. There are two action buttons concerning EEPROM Watch window - **Write EEPROM** and **Read EEPROM**. **Write EEPROM** writes data from EEPROM Watch window into PIC internal EEPROM memory. **Read EEPROM** reads data from PIC internal EEPROM memory and loads it up in EEPROM window.



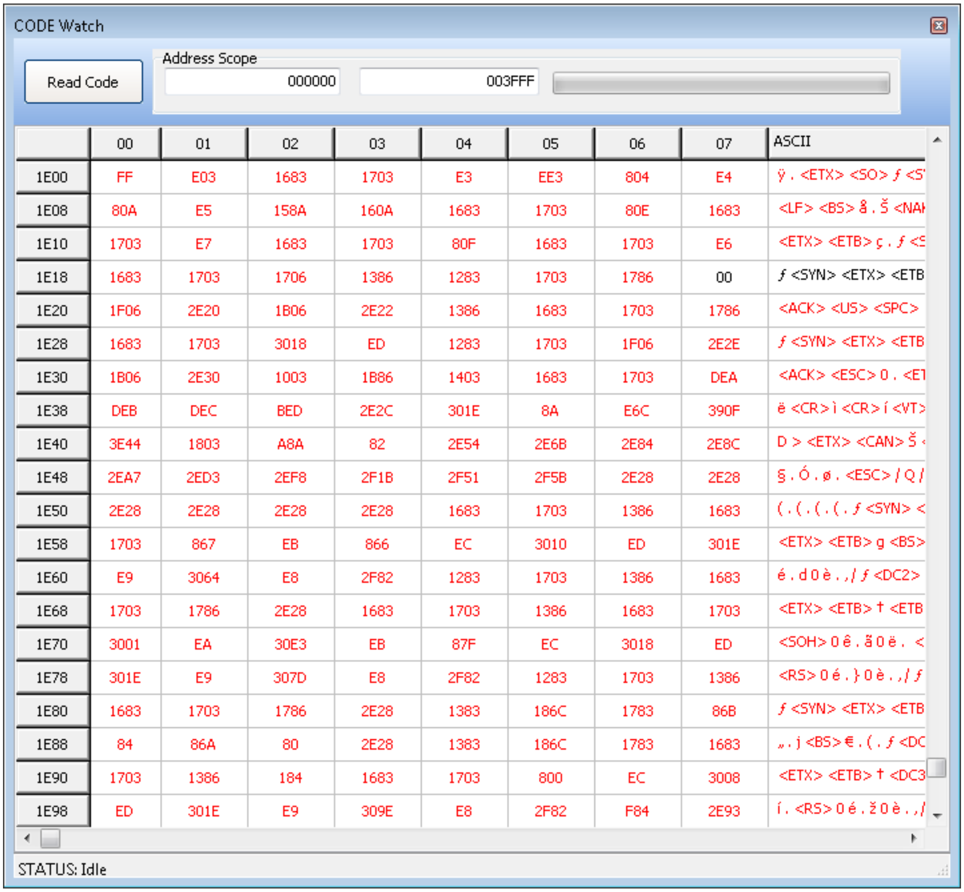
	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
00E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
00F0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	...
0100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

STATUS: Idle

Code Watch Window

mikroICD Code Watch Window is available from the drop-down menu, **View › Debug Windows › View Code**.

The Code Watch window shows code (hex code) written into PIC. There is action button concerning Code Watch window - **Read Code**. **Read Code** reads code from PIC and loads it up in View Code Window.
Also, you can set an address scope in which hex code will be read.



Address Scope 000000 003FFF

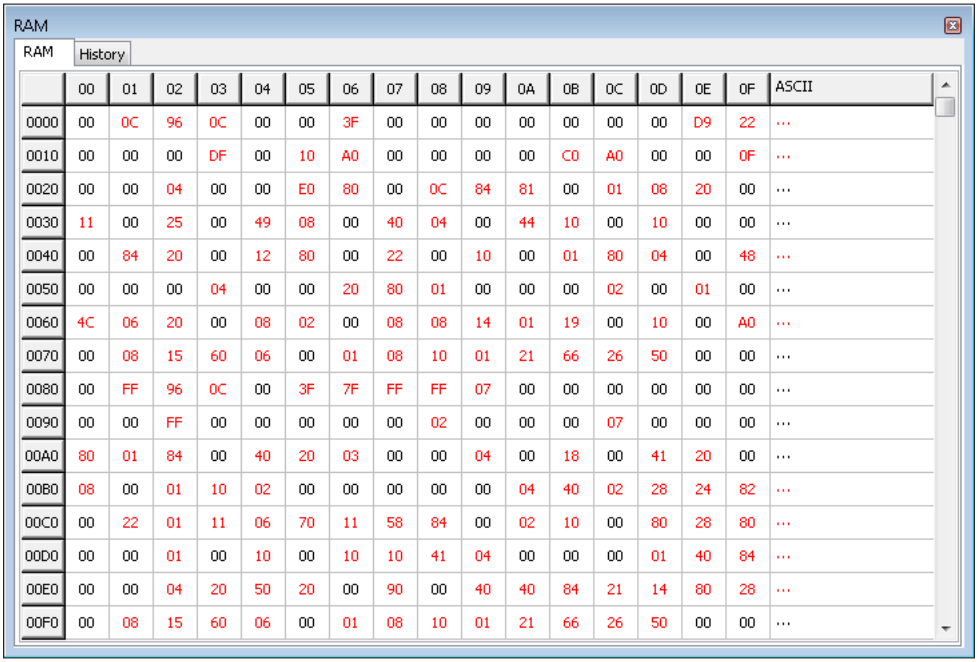
	00	01	02	03	04	05	06	07	ASCII
1E00	FF	E03	1683	1703	E3	EE3	804	E4	ÿ . <ETX> <SO> ƒ <S
1E08	80A	E5	158A	160A	1683	1703	80E	1683	<LF> <BS> å . Š <NA
1E10	1703	E7	1683	1703	80F	1683	1703	E6	<ETX> <ETB> ç . ƒ <S
1E18	1683	1703	1706	1386	1283	1703	1786	00	ƒ <SYN> <ETX> <ETB
1E20	1F06	2E20	1B06	2E22	1386	1683	1703	1786	<ACK> <US> <SPC>
1E28	1683	1703	3018	ED	1283	1703	1F06	2E2E	ƒ <SYN> <ETX> <ETB
1E30	1B06	2E30	1003	1B86	1403	1683	1703	DEA	<ACK> <ESC> 0 . <ET
1E38	DEB	DEC	BED	2E2C	301E	8A	E6C	390F	ë <CR> ì <CR> í <VT>
1E40	3E44	1803	A8A	82	2E54	2E6B	2E84	2E8C	D > <ETX> <CAN> Š
1E48	2EA7	2ED3	2EF8	2F1B	2F51	2F5B	2E28	2E28	§ . Ó . ø . <ESC> / Q /
1E50	2E28	2E28	2E28	2E28	1683	1703	1386	1683	(. (. (. (. ƒ <SYN> <
1E58	1703	867	EB	866	EC	3010	ED	301E	<ETX> <ETB> g <BS>
1E60	E9	3064	E8	2F82	1283	1703	1386	1683	é . d 0 è . , / ƒ <DC2>
1E68	1703	1786	2E28	1683	1703	1386	1683	1703	<ETX> <ETB> † <ETB
1E70	3001	EA	30E3	EB	87F	EC	3018	ED	<SOH> 0 ê . ã 0 ë . <
1E78	301E	E9	307D	E8	2F82	1283	1703	1386	<RS> 0 é . } 0 è . , / ƒ
1E80	1683	1703	1786	2E28	1383	186C	1783	86B	ƒ <SYN> <ETX> <ETB
1E88	84	86A	80	2E28	1383	186C	1783	1683	„ . j <BS> € . (. ƒ <DC
1E90	1703	1386	184	1683	1703	800	EC	3008	<ETX> <ETB> † <DC3
1E98	ED	301E	E9	309E	E8	2F82	F84	2E93	í . <RS> 0 é . ž 0 è . , /

STATUS: Idle

View RAM Memory

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

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



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	0C	96	0C	00	00	3F	00	00	00	00	00	00	00	D9	22	...
0010	00	00	00	DF	00	10	A0	00	00	00	00	C0	A0	00	00	0F	...
0020	00	00	04	00	00	E0	80	00	0C	84	81	00	01	08	20	00	...
0030	11	00	25	00	49	08	00	40	04	00	44	10	00	10	00	00	...
0040	00	84	20	00	12	80	00	22	00	10	00	01	80	04	00	48	...
0050	00	00	00	04	00	00	20	80	01	00	00	00	02	00	01	00	...
0060	4C	06	20	00	08	02	00	08	08	14	01	19	00	10	00	A0	...
0070	00	08	15	60	06	00	01	08	10	01	21	66	26	50	00	00	...
0080	00	FF	96	0C	00	3F	7F	FF	FF	07	00	00	00	00	00	00	...
0090	00	00	FF	00	00	00	00	00	02	00	00	00	07	00	00	00	...
00A0	80	01	84	00	40	20	03	00	00	04	00	18	00	41	20	00	...
00B0	08	00	01	10	02	00	00	00	00	00	04	40	02	28	24	82	...
00C0	00	22	01	11	06	70	11	58	84	00	02	10	00	80	28	80	...
00D0	00	00	01	00	10	00	10	10	41	04	00	00	00	01	40	84	...
00E0	00	00	04	20	50	20	00	90	00	40	40	84	21	14	80	28	...
00F0	00	08	15	60	06	00	01	08	10	01	21	66	26	50	00	00	...

Common Errors

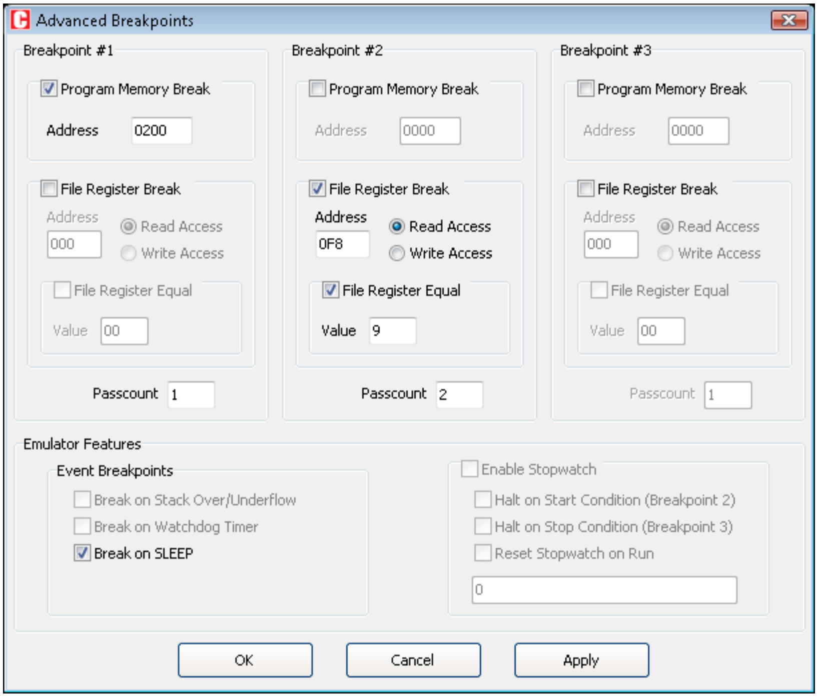
- Trying to program PIC while mikroICD is active.
- Trying to debug **Release** build Type version of program.
- Trying to debug changed program code which hasn't been compiled and pro grammed into PIC.
- Trying to select line that is empty for Run to cursor **[F4]** and Toggle Breakpoints **[F5]** functions.
- Trying to debug PIC with mikroICD while Watch Dog Timer is enabled.
- Trying to debug PIC with mikroICD while Power Up Timer is enabled.
- It is not possible to force Code Protect while trying to debug PIC with mikroICD.
- Trying to debug PIC with mikroICD with pull-up resistors set to ON on RB6 and RB7.
- For correct mikroICD debugging do not use pull-ups.

MIKROICD ADVANCED BREAKPOINTS

mikroICD provides the possibility to use the Advanced Breakpoints. Advanced Breakpoints can be used with PIC18 and PIC18FJ MCUs. To enable Advanced *Breakpoints* set the Advanced Breakpoints checkbox inside Watch window:

☑ Advanced Breakpoints

To configure Advanced Breakpoints, start mikroICD **[F9]** and select **View › Debug Windows › Advanced Breakpoints** option from the drop-down menu or use **[Ctrl+Shift+A]** shortcut.



Advanced Breakpoints

Breakpoint #1
☑ Program Memory Break
Address 0200
☐ File Register Break
Address 000
● Read Access
○ Write Access
☐ File Register Equal
Value 00
Passcount 1

Breakpoint #2
☐ Program Memory Break
Address 0000
☑ File Register Break
Address 0F8
● Read Access
○ Write Access
☑ File Register Equal
Value 9
Passcount 2

Breakpoint #3
☐ Program Memory Break
Address 0000
☐ File Register Break
Address 000
● Read Access
○ Write Access
☐ File Register Equal
Value 00
Passcount 1

Emulator Features
Event Breakpoints
☐ Break on Stack Over/Underflow
☐ Break on Watchdog Timer
☑ Break on SLEEP
☐ Enable Stopwatch
☐ Halt on Start Condition (Breakpoint 2)
☐ Halt on Stop Condition (Breakpoint 3)
☐ Reset Stopwatch on Run
0

Note: When Advanced Breakpoints are enabled mikroICD operates in Real-Time mode, so it will support only the following set of commands: `Start Debugger` **[F9]**, `Run/Pause Debugger` **[F6]** and `Stop Debugger` **[Ctrl+F2]**. Once the Advanced Breakpoint is reached, the Advanced Breakpoints feature can be disabled and mikroICD debugging can be continued with full set of commands. If needed, Advanced Breakepoints can be re-enabled without restarting mikroICD.

Note: Number of Advanced Breakpoints is equal to number of Hardware breakpoints and it depends on used MCU.

Program Memory Break

Program Memory Break is used to set the Advanced Breakpoint to the specific address in program memory. Because of PIC pipelining mechanism program execution may stop one or two instructions after the address entered in the `Address` field. Value entered in the `Address` field must be in hex format.

Note: Program Memory Break can use the Passcount option. The program execution will stop when the specified program address is reached for the N-th time, where N is the number entered in the `Passcount` field. When some Advanced Breakpoint stops the program execution, passcount counters for all Advanced Breakpoints will be cleared.

Program Memory Break

Program Memory Break is used to set the Advanced Breakpoint to the specific address in program memory. Because of PIC pipelining mechanism program execution may stop one or two instructions after the address entered in the Address field. Value entered in the Address field must be in hex format.

Note: Program Memory Break can use the Passcount option. The program execution will stop when the specified program address is reached for the N-th time, where N is the number entered in the `Passcount` field. When some Advanced Breakpoint stops the program execution, passcount counters for all Advanced Breakpoints will be cleared.

File Register Break

File Register Break can be used to stop the code execution when read/write access to the specific data memory location occurs. If `Read Access` is selected, the *File Register Equal* option can be used to set the matching value. The program execution will be stopped when the value read from the specified data memory location is equal to the number written in the `Value` field. Values entered in the `Address` and `Value` fields must be in hex format.

Note: File Register Break can also use the Passcount option in the same way as Program Memory Break.

Emulator Features

Event Breakpoints

- **Break on Stack Overflow/Underflow**: not implemented.
- **Break on Watchdog Timer**: not implemented.
- **Break on SLEEP**: break on SLEEP instruction. SLEEP instruction will not be executed. If you choose to continue the mikroICD debugging **[F6]** then the program execution will start from the first instruction following the SLEEP instruction.

Stopwatch

Stopwatch uses `Breakpoint#2` and `Breakpoint#3` as a Start and Stop conditions. To use the Stopwatch define these two Breakpoints and check the `Enable Stopwatch` checkbox.

Stopwatch options:

Halt on Start Condition

- **Halt on Start Condition (Breakpoint#2)**: when checked, the program execution will stop on `Breakpoint#2`. Otherwise, `Breakpoint#2` will be used only to start the Stopwatch.
- **Halt on Stop Condition (Breakpoint#3)**: when checked, the program execution will stop on `Breakpoint#3`. Otherwise, `Breakpoint#3` will be used only to stop the Stopwatch.
- **Reset Stopwatch on Run**: when checked, the Stopwatch will be cleared before continuing program execution and the next counting will start from zero. Otherwise, the next counting will start from the previous Stopwatch value

CHAPTER 4

mikroC PRO for PIC Specifics

The following topics cover the specifics of mikroC PRO for PIC compiler:

- ANSI Standard Issues
- Predefined Globals and Constants
- Accessing Individual Bits
- Interrupts
- PIC Pointers
- Linker Directives
- Built-in Routines
- Code Optimization
- Memory Type Specifiers

ANSI Standard Issues

Divergence from the ANSI C Standard

- Tentative declarations are not supported.

C Language Exstensions

mikroC PRO for PIC has additional set of keywords that do not belong to the ANSI standard C language keywords:

- code
- data
- rx
- at
- sbit
- bit
- sfr

Related topics: Keywords, PIC Specific

Predefined Globals and Constants

To facilitate programming of PIC compliant MCUs, the *mikroC PRO for PIC* implements a number of predefined globals and constants.

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

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

MIKROELEKTRONIKA - SOFTWARE AND HARDWARE SOLUTIONS FOR EMBEDDED WORLD

Predefined project level defines

There are four predefined project level defines for any project you make. These defines are based on values that you have entered/edited in the current project:

- First one is equal to the name of selected device for the project i.e. if 16F887 is selected device, then 16F887 token will be defined as 1, so it can be used for conditional compilation:
  ```
  #ifdef P16F887
  ...
  #endif
  ```
- The second one is __FOSC__ value of frequency (in Khz) for which the project is built.
- Third one is for identifying *mikroC PRO for PIC* compiler:
  ```
  #ifdef __MIKROC_PRO_FOR_PIC__
  ...
  #endif
  ```
- Fourth one is for identifying the build version. For instance, if a desired build version is 142, user should put this in his code:
  ```
  #if __MIKROC_PRO_FOR_PIC_BUILD__ == 142
  ...
  #endif
  ```

User can define custom project level defines.

Accessing Individual Bits

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

Accessing Individual Bits Of Variables

If you are familiar with a particular MCU, you can access bits by name:

```
// Clear Global Interrupt Bit (GIE)
GIE_bit = 0;
```

Also, you can simply use the direct member selector (.) with a variable, followed by one of identifiers `B0, B1, … , B7,` or `F0, F1, … F7`, with `F7` being the most significant bit:

```
// Clear bit 0 in INTCON register
INTCON.B0 = 0;
// Set bit 5 in ADCON0 register
ADCON0.F5 = 1;
```

There is no need of any special declarations. This kind of selective access is an intrinsic feature of *mikroC PRO for PIC* 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.

Note: If aiming at portability, avoid this style of accessing individual bits, use the bit fields instead.

sbit type

The *mikroC PRO for PIC* compiler has sbit data type which provides access to bit-addressable SFRs. You can access them in the following manner:

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

sbit LEDB at PORTB.F0;
sbit bit_name at sfr-name.F<bit-position>;

// If you are familiar with a particular MCU and its ports and direc-
tion registers (TRIS), you can access bits by their names:
sbit LEDC at RC0_bit;
sbit bit_name at R<port-letter><bit-position>_bit;

sbit TRISC0 at TRISC0_bit;
sbit bit_name at TRIS<port-letter><bit-position>_bit;
```

bit type

The *mikroC PRO for PIC* 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.

```
bit bf;     // bit variable
```

There are no pointers to bit variables:

~~bit *ptr;~~   `// invalid`

An array of type bit is not valid:

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

Note:

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

Related topics: Bit fields, Predefined globals and constants

Interrupts

Interrupts can be easily handled by means of reserved word `interrupt.` *mikroC PRO for PIC* implictly declares function `interrupt` which cannot be redeclared. Its prototype is:

```
void interrupt(void);
```

For P18 low priorty interrupts reserved word is `interrupt_low`:

```
void interrupt_low(void);
```

You are expected to write your own definition (function body) to handle interrupts in your application.

mikroC PRO for PIC saves the following SFR on stack when entering interrupt and pops them back upon return:

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

Use the `#pragma disablecontexsaving` to instruct the compiler not to automatically perform context-switching. This means that no regiser will be saved/restored by the compiler on entrance/exit from interrupt service routine. This enables the user to manually write code for saving registers upon entrance and to restore them before exit from interrupt.

P18 priority interrupts

Note: For the P18 family both low and high interrupts are supported.

1. function with name `interrupt` will be linked as ISR (interrupt service routine) for high level interrupt
2. function with name `interrupt_low` will be linked as ISR for low level interrupt_low

If interrupt priority feature is to be used then the user should set the appropriate SFR bits to enable it. For more information refer to datasheet for specific device.

Function Calls from Interrupt

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

Interrupt Examples

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

```
void interrupt() {
  counter++;
  TMR0 = 96;
  INTCON = $20;
}
```

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

```
void interrupt() {
  if (INTCON.TMR0IF) {
    counter++;
    TMR0 = 96;
    INTCON.TMR0F = 0;
  }
  else if (INTCON.RBIF) {
    counter++;
    TMR0 = 96;
    INTCON.RBIF = 0;
  }
}
```

Linker Directives

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

Directive absolute

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

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

```
short x absolute 0x22;
// Variable x will occupy 1 byte at address 0x22

int y absolute 0x23;
// Variable y will occupy 2 bytes at addresses 0x23 and 0x24
```

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

```
char i absolute 0x33;
// Variable i will occupy 1 byte at address 0x33

long jjjj absolute 0x30;
// Variable will occupy 4 bytes at 0x30, 0x31, 0x32, 0x33; thus,
// changing i changes jjjj highest byte at the same time, and vice versa
```

Directive org

Directive `org` specifies a starting address of a routine in ROM.

Directive `org` is appended to the function definition. Directives applied to non-defining declarations will be ignored, with an appropriate warning issued by the linker.

Here is a simple example:

```
void func(int par) org 0x200 {
// Function will start at address 0x200
  asm nop;
}
```

It is possible to use org directive with functions that are defined externally (such as library functions). Simply add org directive to function declaration:

```
void UART_Write1(char data) org 0x200;
```

Note: Directive `org` can be applied to any routine except for interrupt.

Directive orgall

If the user wants to place his routines, constants, etc, above a specified address in ROM, `#pragma orgall` directive should be used:

```
#pragma orgall 0x200
```

Directive funcorg

You can use the `#pragma funcorg` directive to specify the starting address of a routine in ROM using routine name only:

```
 #pragma funcorg <func_name>  <starting_address>
```

Related topics: Indirect Function Calls

Indirect Function Calls

If the linker encounters an indirect function call (by a pointer to function), it assumes that any of the functions addresses of which were taken anywhere in the program, can be called at that point. Use the `#pragma funcall` directive to instruct the linker which functions can be called indirectly from the current function:

```
#pragma funcall <func_name> <called_func>[ , <called_func>,...]
```

A corresponding pragma must be placed in the source module where the function `func_name` is implemented. This module must also include declarations of all functions listed in the `called_func` list.

These functions will be linked if the function `func_name` is called in the code no matter whether any of them was called or not.

Note: The `#pragma funcall` directive can help the linker to optimize function frame allocation in the compiled stack.

Related topics: Linker Directives

Built-in Routines

mikroC PRO for PIC compiler provides a set of useful built-in utility functions. Built-in functions do not require any header files to be included; you can use them in any part of your project.

Built-in routines are implemented as “inline”; i.e. code is generated in the place of the call, so the call doesn’t count against the nested call limit. The only exceptions are `Vdelay_ms, Delay_Cyc` and `Get_Fosc_kHz` which are actual C routines.

Note: **Lo, Hi, Higher** and **Highest** functions are not implemented in compiler any more. If you want to use these functions you must include `built_in.h` into your project.

- Lo
- Hi
- Higher
- Highest
- Delay_us
- Delay_ms
- Vdelay_ms
- Delay_Cyc
- Clock_Khz
- Clock_Mhz
- Get_Fosc_kHz

Lo

Prototype	`unsigned short Lo(long number);`
Returns	Returns the lowest 8 bits (byte) of `number`, bits 0..7.
Description	Function returns the lowest byte of `number`. Function does not interpret bit patterns of `number` – it merely returns 8 bits as found in register. 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	`unsigned short Hi(long number);`
Returns	Returns next to the lowest byte of `number`, bits 8..15.
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 = Hi(d);   // Equals 0x30`

Higher

Prototype	`unsigned short Higher(long number);`
Returns	Returns next to the highest byte of `number`, bits 16..23.
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 = Higher(d);   // Equals 0xAC`

Highest

Prototype	`unsigned short Highest(long number);`
Returns	Returns the highest byte of `number`, bits 24..31.
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 = Highest(d);   // Equals 0x01`

Delay_us

Prototype	`void Delay_us(const time_in_us);`
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. This routine generates nested loops using registers `R13, R12, R11` and `R10`. The number of used registers varies from 0 to 4, depending on requested `time_in_us`.
Requires	Nothing.
Example	`Delay_us(10);  /* Ten microseconds pause */`

Delay_ms

Prototype	`void Delay_ms(const time_in_ms);`
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. This routine generates nested loops using registers `R13, R12, R11` and `R10`. The number of used registers varies from 0 to 4, depending on requested `time_in_ms`.
Requires	Nothing.
Example	`Delay_ms(1000);  /* One second pause */`

Vdelay_ms

Prototype	`void Vdelay_ms(unsigned time_in_ms);`
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	`void Delay_Cyc(char Cycles_div_by_10);`
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 3 and 255.
Requires	Nothing.
Example	`Delay_Cyc(10);  /* Hundred MCU cycles pause */`

Clock_Khz

Prototype	`unsigned Clock_Khz(void);`
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	`unsigned short Clock_Mhz(void);`
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();`

Get_Fosc_kHz

Prototype	`unsigned long Get_Fosc_kHz(void);`
Returns	Device clock in KHz, rounded to the nearest integer.
Description	Function returns device clock in KHz, rounded to the nearest integer. Note that `Get_Fosc_kHz` is library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Requires	Nothing.
Example	`clk = Clock_Khz();`

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" elimination

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

Related topics: PIC specifics, *mikroC PRO for PIC* specifics, Memory type specifiers

CHAPTER 5

PIC SPECIFICS

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

Types Efficiency

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

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

Nested Calls Limitations

Nested call represents a function call within function body, either to itself (*recursive calls*) or to another function. Recursive function calls are supported by *mikroC PRO for PIC* but with limitations. Recursive function calls can't contain any function parameters and local variables due to the PIC's stack and memory limitations.

mikroC PRO for PIC limits the number of non-recursive nested calls to:

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

Note that some of the built-in routines do not count against this limit, due to their "inline" implementation.

Number of the allowed nested calls decreases by one if you use any of the following operators in the code: `* / %.` It further decreases if you use interrupts in the program. Number of decreases is specified by number of functions called from interrupt. Check functions reentrancy.

If the allowed number of nested calls is exceeded, the compiler will report a stack overflow error.

PIC18FxxJxx Specifics

Shared Address SFRs

mikroC PRO for PIC does not provide auto setting of bit for acessing alternate register. This is new feature added to pic18fxxjxx family and will be supported in future. In several locations in the SFR bank, a single address is used to access two different hardware registers. In these cases, a “legacy” register of the standard PIC18 SFR set (such as OSCCON, T1CON, etc.) shares its address with an alternate register. These alternate registers are associated with enhanced configuration options for peripherals, or with new device features not included in the standard PIC18 SFR map. A complete list of shared register addresses and the registers associated with them is provided in datasheet.

PIC16 Specifics

Breaking Through Pages

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

Limits of Indirect Approach Through FSR

Pointers with PIC16 are “near”: they carry only the lower 8 bits of the address. Compiler will automatically clear the 9th bit upon startup, so that pointers will refer to banks 0 and 1. To access the objects in banks 2 or 3 via pointer, user should manually set the IRP, and restore it to zero after the operation. The stated rules apply to any indirect approach: arrays, structures and unions assignments, etc.

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

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

Related topics: *mikroC PRO for PIC* specifics

MEMORY TYPE SPECIFIERS

The *mikroC PRO for PIC* 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
- sfr

Memory type specifiers can be included in variable declaration.

For example:

```
char data             data_buffer;    // puts data_buffer in data ram
const char code txt[] = "ENTER PARAMETER:"; // 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 char code txt[] = "ENTER PARAMETER:";`

data

Description	This memory specifier is used when storing variable to the internal data SRAM.
Example	`// puts PORTG in data ram` `sfr data unsigned short PORTG absolute 0x65;`

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` `sfr char rx y;`

sfr

Description	This memory specifier in combination with (`rx, data`) allows user to access special function registers. It also instructs compiler to maintain same identifier in C and assembly.
Example	`sfr rx char y;`

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: Accessing individual bits, SFRs, Constants, Functions

CHAPTER 6

mikroC PRO for PIC
Language Reference

The mikroC PRO for PIC Language Reference describes the syntax, semantics and implementation of the mikroC PRO for PIC language.

The aim of this reference guide is to provide a more understandable description of the mikroC PRO for PIC language to the user.

LEXICAL ELEMENTS OVERVIEW

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

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

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

```
int i; float f;
```

and

```
int
  i;

    float f;
```

are lexically equivalent and parse identically to give six tokens:

```
int
i
;
float
f
;
```

Whitespace in Strings

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

```
char name[ ] = "mikro foo";
```

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

```
char
name
[
]
=
"mikro foo"    /* just one token here! */
;
```

Line Splicing with Backslash (\)

A special case occurs if a line ends with a backslash (\). Both backslash and new line character are discarded, allowing two physical lines of a text to be treated as one unit. So, the following code

```
"mikroC PRO \

for PIC Compiler"
```

parses into `"mikroC PRO for PIC Compiler"`. Refer to String Constants for more information.

COMMENTS

Comments are pieces of a text used to annotate a program and technically are 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 delineate comments: the C method and the C++ method. Both are supported by *mikroC PRO for PIC*.

You should also follow the guidelines on the use of whitespace and delimiters in comments, discussed later in this topic to avoid other portability problems.

C comments

C comment is any sequence of characters placed after the symbol pair /*. The comment terminates at the first occurance of the pair */ following the initial /*. The entire sequence, including four comment-delimiter symbols, is replaced by one space after macro expansion.

In the *mikroC PRO for PIC*,

```
int /* type */ i /* identifier */;
```

parses as:

```
int i;
```

Note that the *mikroC PRO for PIC* does not support a nonportable token pasting strategy using /**/. For more information on token pasting, refer to the Preprocessor Operators.

C++ comments

The *mikroC PRO for PIC* allows single-line comments using two adjacent slashes (//). The comment can start in any position and extends until the next new line.

The following code

```
int i;  // this is a comment
int j;
```

parses as:

```
int i;
int j;
```

Nested comments

ANSI C doesn't allow nested comments. The attempt to nest a comment like this

```
/*  int /* declaration */ i; */
```

fails, because the scope of the first `/*` ends at the first `*/`. This gives us

```
i; */
```

which would generate a syntax error.

TOKENS

Token is the smallest element of a C 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.

The *mikroC PRO for PIC* 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:

```
inter =  a+++b;
```

First, note that `inter` would be parsed as a single identifier, rather than as the keyword `int` followed by the identifier `er`.

The programmer who has written the code might have intended to write `inter = a + (++b)`, but it wouldn't work that way. The compiler would parse it into the seven following tokens:

```
inter     // variable identifier
=         // assignment operator
a         // variable identifier
++        // postincrement operator
+         // addition operator
b         // variable identifier
;         // statement terminator
```

Note that `+++` parses as `++` (the longest token possible) followed by `+`.

According to the operator precedence rules, our code sequence is actually:

```
inter (a++)+b;
```

CONSTANTS

Constants or *literals* are tokens representing fixed numeric or character values.

The *mikroC PRO for PIC* supports:

- integer constants
- floating point constants
- character constants
- string constants (strings literals)
- enumeration constants

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

Integer Constants

Integer constants can be decimal (base 10), hexadecimal (base 16), binary (base 2), or octal (base 8). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value.

Long and Unsigned Suffixes

The suffix `L` (or `l`) attached to any constant forces that constant to be represented as a `long`. Similarly, the suffix `U` (or `u`) forces a constant to be `unsigned`. Both `L` and `U` suffixes can be used with the same constant in any order or case: `ul, Lu, UL`, etc.

In the absence of any suffix (`U`, `u`, `L`, or `l`), a constant is assigned the “smallest” of the following types that can accommodate its value: `short, unsigned short, int, unsigned int, long int, unsigned long int`.

Otherwise:

- If a constant has the `U` suffix, its data type will be the first of the following that can accommodate its value: `unsigned short, unsigned int, unsigned long int.`
- If a constant has the `L` suffix, its data type will be the first of the following that can accommodate its value: `long int, unsigned long int`.
- If a constant has both `L` and `U` suffixes, (`LU` or `UL`), its data type will be `unsigned long int`.

Decimals

Decimal constants from -2147483648 to 4294967295 are allowed. Constants exceeding these bounds will produce an “Out of range” error. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10;    /* decimal 10 */
int i = 010;   /* decimal 8  */
int i = 0;     /* decimal 0 = octal 0 */
```

In the absence of any overriding suffixes, the data type of a decimal constant is derived from its value, as shown below:

Value Assigned to Constant	Assumed Type
< -2147483648	Error: Out of range!
-2147483648 – -32769	`long`
-32768 – -129	`int`
-128 – 127	`short`
128 – 255	`unsigned short`
256 – 32767	`int`
32768 – 65535	`unsigned int`
65536 – 2147483647	`long`
2147483648 – 4294967295	`unsigned long`
> 4294967295	Error: Out of range!

Hexadecimal Constants

All constants starting with `0x` (or `0X`) are taken to be hexadecimal. In the absence of any overriding suffixes, the data type of an hexadecimal constant is derived from its value, according to the rules presented above. For example, `0xC367` will be treated as `unsigned int`.

Binary Constants

All constants starting with `0b` (or `0B`) are taken to be binary. In the absence of any overriding suffixes, the data type of an binary constant is derived from its value, according to the rules presented above. For example, `0b11101` will be treated as `short`.

Octal Constants

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. In the absence of any overriding suffixes, the data type of an octal constant is derived from its value, according to the rules presented above. For example, `0777` will be treated as `int`.

Floating Point Constants

A floating-point constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- `e` or `E` and a signed integer exponent (optional)
- Type suffix: `f` or `F` or `l` or `L` (optional)

Either decimal integer or decimal fraction (but not both) can be omitted. Either decimal point or letter `e` (or `E`) with a signed integer exponent (but not both) can be omitted. These rules allow conventional and scientific (exponent) notations.

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

The *mikroC PRO for PIC* limits floating-point constants to the 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
```

The *mikroC PRO for PIC* floating-point constants are of the type `double`. Note that the *mikroC PRO for PIC*'s implementation of ANSI Standard considers `float` and `double` (together with the `long double` variant) to be the same type.

Character Constants

A character constant is one or more characters enclosed in single quotes, such as `'A'`, `'+'`, or `'\n'`. In the mikroC PRO for PIC, single-character constants are of the `unsigned int` type. Multi-character constants are referred to as *string constants* or *string literals*. For more information refer to String Constants.

Escape Sequences

A backslash character (`\`) is used to introduce an escape sequence, which allows a visual representation of certain nongraphic characters. One of the most common escape constants is the newline character (`\n`).
A backslash is used with octal or hexadecimal numbers to represent an ASCII symbol or control code corresponding to that value; for example, `'\x3F'` for the question mark. Any value within legal range for data type char (`0` to `0xFF` for the *mikroC PRO for PIC*) can be used. Larger numbers will generate the compiler error “Out of range”. For example, the octal number `\777` is larger than the maximum value allowed (`\377`) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Note: You must use the sequence `\\` to represent an ASCII backslash, as used in operating system paths.

The following table shows the available escape sequences:

Sequence	Value	Char	Description
`\a`	0x07	BEL	Audible bell
`\b`	0x08	BS	Backspace
`\f`	0x0C	FF	Formfeed
`\n`	0x0A	LF	Newline (Linefeed)
`\r`	0x0D	CR	Carriage Return
`\t`	0x09	HT	Tab (horizontal)
`\v`	0x0B	VT	Vertical Tab
`\\`	0x5C	\	Backslash
`\'`	0x27	‘	Single quote (Apostrophe)
`\"`	0x22	‘’	Double quote
`\?`	0x3F	?	Question mark
`\O`		any	O = string of up to 3 octal digits
`\xH`		any	H = string of hex digits
`\XH`		any	H = string of hex digits

Disambiguation

Some ambiguous situations might arise when using escape sequences.

Here is an example:

```
Lcd_Out_Cp("\x091.0 Intro");
```

This is intended to be interpreted as `\x09` and `"1.0 Intro"`. However, the mikroC PRO for PIC compiles it as the hexadecimal number `\x091` and literal string `".0 Intro"`. To avoid such problems, we could rewrite the code in the following way:

```
Lcd_Out_Cp("\x09" "1.0 Intro");
```

For more information on the previous line, refer to String Constants.

Ambiguities might also arise if an octal escape sequence is followed by a nonoctal digit. For example, the following constant:

```
"\118"
```

would be interpreted as a two-character constant made up of the characters `\11` and `8`, because `8` is not a legal octal digit.

String Constants

String constants, also known as *string literals*, are a special type of constants which store fixed sequences of characters. A string literal is a sequence of any number of characters surrounded by double quotes:

```
"This is a string."
```

The *null string*, or empty string, is written like `""`. A literal string is stored internally as a given sequence of characters plus a final null character. A null string is stored as a single null character.

The characters inside the double quotes can include escape sequences. This code, for example:

```
"\t\"Name\"\\\tAddress\n\n"
```

prints like this:

```
   "Name"\     Address
```

The "Name" is preceded by two tabs; The Address is preceded by one tab. The line is followed by two new lines. The `\"` provides interior double quotes. The escape character sequence `\\` is translated into `\` by the compiler.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. For example:

```
"This is "  "just"
    " an example."
```

is equivalent to

```
"This is just an example."
```

Line Continuation with Backslash

You can also use the backslash (\) as a continuation character to extend a string constant across line boundaries:

```
"This is really \
        a one-line string."
```

Enumeration Constants

Enumeration constants are identifiers defined in `enum` type declarations. The identifiers are usually chosen as mnemonics to contribute to legibility. Enumeration constants are of `int` type. They can be used in any expression where integer constants are valid.

For example:

```
enum weekdays { SUN = 0, MON, TUE, WED, THU, FRI, SAT };
```

The identifiers (enumerators) used must be unique within the scope of the `enum` declaration. Negative initializers are allowed. See Enumerations for details about `enum` declarations.

Pointer Constants

A pointer or pointed-at object can be declared with the const modifier. Anything declared as const cannot change its value. It is also illegal to create a pointer that might violate a non-assignability of the constant object.

Consider the following examples:

```
int i;                      // i is an int
int * pi;                   // pi is a pointer to int (uninitialized)
int * const cp = &i;        // cp is a constant pointer to int
const int ci = 7;           // ci is a constant int
const int * pci;            // pci is a pointer to constant int
const int * const cpc = &ci;     // cpc is a constant pointer to a
                                 // constant int
```

The following assignments are legal:

```
i = ci;              // Assign const-int to int
*cp = ci;            // Assign const-int to
                     // object-pointed-at-by-a-const-pointer
++pci;               // Increment a pointer-to-const
pci = cpc;           // Assign a const-pointer-to-a-const to a
                     // pointer-to-const
```

The following assignments are illegal:

```
ci = 0;              // NO--cannot assign to a const-int
ci--;                // NO--cannot change a const-int
*pci = 3;            // NO--cannot assign to an object
                     // pointed at by pointer-to-const.
cp = &ci;            // NO--cannot assign to a const-pointer,
                     // even if value would be unchanged.
cpc++;               // NO--cannot change const-pointer
pi = pci;            // NO--if this assignment were allowed,
                     // you would be able to assign to *pci
                     // (a const value) by assigning to *pi.
```

Similar rules are applayed to the `volatile` modifier. Note that both `const` and `volatile` can appear as modifiers to the same identifier.

Constant Expressions

A constant expressions can be evaluated during translation rather that runtime and accordingly may be used in any place that a constant may be.

Constant expressions can consist only of the following:

- literals,
- enumeration constants,
- simple constants (no constant arrays or structures),
- `sizeof` operators.

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a `sizeof` operator: assignment, comma, decrement, function call, increment.

Each constant expression can evaluate to a constant that is in the range of representable values for its type.

Constant expression can be used anywhere a constant is legal.

KEYWORDS

Keywords are words reserved for special purposes and must not be used as normal identifier names.
Beside standard C keywords, all relevant SFR are defined as global variables and represent reserved words that cannot be redefined (for example: `TMR0, PCL,` etc). Probe the Code Assistant for specific letters (**Ctrl+Space** in Editor) or refer to Predefined Globals and Constants.

Here is an alphabetical listing of keywords in C:

- `asm`
- `auto`
- `break`
- `case`
- `char`
- `const`
- `continue`
- `default`
- `do`
- `double`
- `else`
- `enum`
- `extern`
- `float`
- `for`
- `goto`
- `if`
- `int`
- `long`
- `register`
- `return`
- `short`
- `signed`
- `sizeof`
- `static`
- `struct`
- `switch`
- `typedef`
- `union`
- `unsigned`
- `void`
- `volatile`
- `while`

Also, the *mikroC PRO for PIC* includes a number of predefined identifiers used in libraries. You could replace them by your own definitions, if you want to develop your own libraries. For more information, see mikroC PRO for PIC 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 with the meaning of *object* in object-oriented programming).

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

Case Sensitivity

The *mikroC PRO for PIC* identifiers aren't case sensitive by default, so that `Sum`, `sum`, and `suM` represent an equivalent identifier. Case sensitivity can be activated or suspended in Output Settings window. Even if case sensitivity is turned off Keywords remain case sensitive and they must be written in lower case.

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 and sharing the same name space then error arises. Duplicate names are legal for different name spaces regardless of scope rules. For more information on scope, refer to Scope and Visibility.

Identifier Examples

Here are some valid identifiers:

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

… and here are some invalid identifiers:

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

PUNCTUATORS

The *mikroC PRO for PIC* punctuators (also known as separators) are:

- [ ] – Brackets
- ( ) – Parentheses
- { } – Braces
- , – Comma
- ; – Semicolon
- : – Colon
- * – Asterisk
- = – Equal sign
- # – Pound sign

Most of these punctuators also function as operators.

Brackets

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

```
char ch, str[] = "mikro";

int mat[3][4];        /* 3 x 4 matrix */
ch = str[3];          /* 4th element */
```

Parentheses

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

```
d = c * (a + b);      /* override normal precedence */

if (d == z) ++x;      /* essential with conditional statement */
func();                /* function call, no args */
void func2(int n);    /* function declaration with parameters */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during an expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

For more information, refer to Operators Precedence And Associativity and Expressions.

Braces

Braces `{ }` indicate the start and end of a compound statement:

```
if (d == z) {
  ++x;
  func();
}
```

Closing brace serves as a terminator for the compound statement, so a semicolon is not required after `}`, except in structure declarations. Sometimes, the semicolon can be illegal, as in

```
if (statement)
   { ... };     /* illegal semicolon! */
else
   { ... };
```

For more information, refer to the Compound Statements.

Comma

Comma (`,`) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

Comma is also used as an operator in comma expressions. Mixing two uses of comma is legal, but you must use parentheses to distinguish them. Note that (exp1, exp2) evalutates both but is equal to the second:

```
func(i, j);                           /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5)); /* also calls func with two
args! */
```

Semicolon

Semicolon (`;`) is a statement terminator. Any legal C expression (including the empty expression) followed by a semicolon is interpreted as a statement, known as an expression statement. The expression is evaluated and its value is discarded. If the expression statement has no side effects, the *mikroC PRO for PIC* might ignore it.

```
a + b;     /* Evaluate a + b, but discard value */
++a;       /* Side effect on a, but discard value of ++a */
;          /* Empty expression, or a null statement */
```

Semicolons are sometimes used to create an empty statement:

```
for (i = 0; i < n; i++);
```

For more information, see the Statements.

Colon

Use colon (`:`) to indicate the labeled statement:

```
start:  x = 0;
  ...
goto start;
```

Labels are discussed in the Labeled Statements.

Asterisk (Pointer Declaration)

Asterisk (*) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr;  /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;          /* a pointer to an array of integers */
double ***double_ptr;  /* a pointer to a matrix of doubles */
```

You can also use asterisk as an operator to either dereference a pointer or as multiplication operator:

```
i = *int_ptr;
a = b * 3.14;
```

For more information, see the Pointers.

Equal Sign

Equal sign (`=`) separates variable declarations from initialization lists:

```
int test[ 5]  = {  1,  2,  3,  4,  5 } ;
int x = 5;
```

Equal sign is also used as an assignment operator in expressions:

```
int a, b, c;
a = b + c;
```

For more information, see Assignment Operators.

Pound Sign (Preprocessor Directive)

Pound sign (`#`) indicates a preprocessor directive when it occurs as the first non-whitespace character on a line. It signifies a compiler action, not necessarily associated with a code generation. See the Preprocessor Directives for more information.

`#` and `##` are also used as operators to perform token replacement and merging during the preprocessor scanning phase. See the Preprocessor Operators.

CONCEPTS

This section covers some basic concepts of language, essential for understanding of how C programs work. First, we need to establish the following terms that will be used throughout the help:

- Objects and lvalues
- Scope and Visibility
- Name Spaces
- Duration

Objects

An object is a specific region of memory that can hold a fixed or variable value (or set of values). This use of a term *object* is different from the same term, used in object-oriented languages, which is more general. Our definiton of the word would encompass functions, variables, symbolic constants, user-defined data types, and labels.

Each value has an associated name and type (also known as a data type). The name is used to access the object and can be a simple identifier or complex expression that uniquely refers the object.

Objects and Declarations

Declarations establish a necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type.

Associating identifiers with objects requires each identifier to have at least two attributes: storage class and type (sometimes referred to as data type). The *mikroC PRO for PIC* compiler deduces these attributes from implicit or explicit declarations in the source code. Usually, only the type is explicitly specified and the storage class specifier assumes the automatic value auto.

Generally speaking, an identifier cannot be legally used in a program before its declaration point in the source code. Legal exceptions to this rule (known as forward references) are labels, calls to undeclared functions, and struct or union tags.
The range of objects that can be declared includes:

- Variables
- Functions
- Types
- Arrays of other types
- Structure, union, and enumeration tags

- Structure members
- Union members
- Enumeration constants
- Statement labels
- Preprocessor macros

The recursive nature of the declarator syntax allows complex declarators. You'll probably want to use typedefs to improve legibility if constructing complex objects.

Lvalues

Lvalue is an object locator: an expression that designates an object. An example of lvalue expression is `*P`, where `P` is any expression evaluating to a non-null pointer. A modifiable lvalue is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A const pointer to a constant, for example, is not a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, l stood for "left", meaning that lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment operator. For example, if a and b are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as `a = 1` and `b = a + b` are legal.

Rvalues

The expression `a + b` is not lvalue: `a + b = a` is illegal because the expression on the left is not related to an object. Such expressions are sometimes called *rvalues* (short for right values).

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: block (or local), function, function prototype, and file. These categories depend on how and where identifiers are declared.

- **Block**: The scope of an identifier with block (or local) scope starts at the declara tion point and ends at the end of the block containing the declaration (such block is known as the enclosing block). Parameter declarations with a function defini tion also have block scope, limited to the scope of the function body.
- **File**: File scope identifiers, also known as *globals*, are declared outside of all blocks; their scope is from the point of declaration to the end of the source file.
- **Function**: The only identifiers having function scope are statement labels. Label names can be used with goto statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing `label_name`: fol lowed by a statement. Label names must be unique within a function.
- **Function prototype**: Identifiers declared within the list of parameter declarations in a function prototype (not as a part of a function definition) have a function pro totype scope. This scope ends at the end of the function prototype.

Visibility

The visibility of an identifier is a region of the program source code from which an identifier's associated object can be legally accessed.

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

Technically, visibility cannot exceed a scope, but a scope can exceed visibility. See the following example:

```
void f (int i) {
 int j;           // auto by default
 j = 3;           // int i and j are in scope and visible

   {               // nested block
    double j;     // j is local name in the nested block
    j = 0.1;      // i and double j are visible;
                 // int j = 3 in scope but hidden
   }
```

```
                    // double j out of scope
 j += 1;           // int j visible and = 4
}
// i and j are both out of scope
```

Name Spaces

Name space is a scope within which an identifier must be unique. The *mikroC PRO for PIC* uses four distinct categories of identifiers:

1. `goto` label names - must be unique within the function in which they are declared.
2. Structure, union, and enumeration tags - must be unique within the block in which they are defined. Tags declared outside of any function must be unique.
3. Structure and union member names - must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.
4. Variables, typedefs, functions, and enumeration members - must be unique with in the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Duplicate names are legal for different name spaces regardless of the scope rules.

For example:

```
int blue = 73;

{  // open a block
   enum colors {  black, red, green, blue, violet, white }  c;
   /* enumerator blue = 3 now hides outer declaration of int blue */

   struct colors {  int i, j; } ;    // ILLEGAL: colors duplicate tag
   double red = 2;                   // ILLEGAL: redefinition of red
}

blue = 37;                            // back in int blue scope
```

Duration

Duration, closely related to a storage class, defines a period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike typedefs and types, have real memory allocated during run time. There are two kinds of duration: *static* and *local*.

Static Duration

Memory is allocated to objects with static duration as soon as execution is underway; this storage allocation lasts until the program terminates. Static duration objects usually reside in fixed data segments allocated according to the memory specifier in force. All globals have static duration. All functions, wherever defined, are objects with static duration. Other variables can be given static duration by using the explicit `static` or `extern` storage class specifiers.

In the *mikroC PRO for PIC*, static duration objects are *not* initialized to zero (or null) in the absence of any explicit initializer.

Don’t mix static duration with file or global scope. An object can have static duration *and* local scope – see the example below.

Local Duration

Local duration objects are also known as *automatic* objects. They are created on the stack (or in a register) when an enclosing block or a function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable.
The storage class specifier `auto` can be used when declaring local duration variables, but it is usually redundant, because `auto` is default for variables declared within a block.

An object with local duration also has local scope because it does not exist outside of its enclosing block. On the other hand, a local scope object *can* have static duration. For example:

```
void f() {
  /* local duration variable; init a upon every call to f */
  int a = 1;
  /* static duration variable; init b only upon first call to f */
  static int b = 1;
  /* checkpoint! */
  a++;
  b++;
}

void main() {
/* At checkpoint, we will have: */
f();   // a=1, b=1, after first call,
f();   // a=1, b=2, after second call,
f();   // a=1, b=3, after third call,
       // etc.
}
```

TYPES

The *mikroC PRO for PIC* is a strictly typed language, which means that every object, function, and expression must have a strictly defined type, known in the time of compilation. Note that the *mikroC PRO for PIC* works exclusively with numeric types.

The type serves:

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

The *mikroC PRO for PIC* supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers with various precisions, arrays, structures, and unions. In addition, pointers to most of these objects can be established and manipulated in memory.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as a set of values (often implementation-dependent) that identifiers of that type can assume, together with a set of operations allowed with these values. The compile-time operator `sizeof` allows you to determine the size in bytes of any standard or user-defined type.

The *mikroC PRO for PIC* standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that the *mikroC PRO for PIC* can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Type Categories

A common way to categorize types is to divide them into:

- fundamental
- derived

The fudamental types represent types that cannot be split up into smaller parts. They are sometimes referred to as *unstructured* types. The fundamental types are `void`, `char`, `int`, `float`, and `double`, together with `short`, `long`, `signed`, and `unsigned` variants of some of them. For more information on fundamental types, refer to the topic Fundamental Types.

The derived types are also known as *structured* types and they include pointers to other types, arrays of other types, function types, structures, and unions. For more information on derived types, refer to the topic Derived Types.

Fundamental Types

The fudamental types represent types that cannot be divided into more basic elements, and are the model for representing elementary data on machine level. The fudamental types are sometimes referred to as *unstructured types*, and are used as elements in creating more complex derived or user-defined types.

The fundamental types include:

- Arithmetic Types
- Enumerations
- Void Type

Arithmetic Types

The arithmetic type specifiers are built up from the following keywords: `void`, `char`, `int`, `float` and `double`, together with the prefixes `short`, `long`, `signed` and `unsigned`. From these keywords you can build both integral and floating-point types.

Integral Types

The types `char` and `int`, together with their variants, are considered to be integral data types. Variants are created by using one of the prefix modifiers `short`, `long`, `signed` and `unsigned`.

In the table below is an overview of the integral types – keywords in parentheses can be (and often are) omitted.

The modifiers `signed` and `unsigned` can be applied to both `char` and `int`. In the absence of the unsigned prefix, signed is automatically assumed for integral types. The only exception is `char`, which is `unsigned` by default. The keywords `signed` and `unsigned`, when used on their own, mean `signed int` and `unsigned int`, respectively.

The modifiers `short` and `long` can only be applied to `int`. The keywords `short` and `long`, used on their own, mean `short int` and `long int`, respectively.

Type	Size in Bytes	Range
`(unsigned) char`	1	0 .. 255
`signed char`	1	- 128 .. 127
`(signed) short (int)`	1	- 128 .. 127
`unsigned short (int)`	1	0 .. 255
`(signed) int`	2	-32768 .. 32767
`unsigned (int)`	2	0 .. 65535
`(signed) long (int)`	4	-2147483648 .. 2147483647
`unsigned long (int)`	4	0 .. 4294967295

Floating-point Types

The types `float` and `double`, together with the `long double` variant, are considered to be floating-point types. The *mikroC PRO for PIC*'s implementation of an ANSI Standard considers all three to be the same type.

Floating point in the *mikroC PRO for PIC* is implemented using the Microchip AN575 32-bit format (IEEE 754 compliant).

An overview of the floating-point types is shown in the table below:

Type	Size in Bytes	Range
`float`	4	-1.5 * 1045 .. +3.4 * 1038
`double`	4	-1.5 * 1045 .. +3.4 * 1038
`long double`	4	-1.5 * 1045 .. +3.4 * 1038

Enumerations

An enumeration data type is used for representing an abstract, discreet set of values with appropriate symbolic names.

Enumeration Declaration

Enumeration is declared like this:

```
enum tag {enumeration-list};
```

Here, `tag` is an optional name of the enumeration; `enumeration-list` is a comma-delimited list of discreet values, enumerators (or enumeration constants). Each enumerator is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator is set to zero, and the value of each succeeding enumerator is set to a value of its predecessor increased by one.

Variables of the `enum` type are declared the same as variables of any other type. For example, the following declaration:

```
enum colors { black, red, green, blue, violet, white } c;
```

establishes a unique integral type, `enum colors`, variable `c` of this type, and set of enumerators with constant integer values (black = 0, red = 1, ...). In the *mikroC PRO for PIC*, a variable of an enumerated type can be assigned any value of the type `int` – no type checking beyond that is enforced. That is:

```
c = red;      // OK
c = 1;        // Also OK, means the same
```

With explicit integral initializers, you can set one or more enumerators to specific values. The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). Any subsequent names without initializers will be increased by one. These values are usually unique, but duplicates are legal.

The order of constants can be explicitly re-arranged. For example:

```
enum colors { black,       // value 0
              red,         // value 1
              green,       // value 2
              blue=6,      // value 6
              violet,      // value 7
              white=4 };   // value 4
```

Initializer expression can include previously declared enumerators. For example, in the following declaration:

```
enum memory_sizes { bit = 1, nibble = 4 * bit, byte = 2 * nibble,
                    kilobyte = 1024 * byte };
```

nibble would acquire the value 4, byte the value 8, and kilobyte the value 8192.

Anomous Enum Type

In our previous declaration, the identifier `colors` is an optional enumeration tag that can be used in subsequent declarations of enumeration variables of the `enum colors` type:

```
enum colors bg, border;  /* declare variables bg and border */
```

Like with struct and union declarations, you can omit the tag if no further variables of this `enum` type are required:

```
/* Anonymous enum type: */
enum { black, red, green, blue, violet, white } color;
```

Enumeration Scope

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```
int blue = 73;

{ // open a block
    enum colors { black, red, green, blue, violet, white } c;
    /* enumerator blue = 3 now hides outer declaration of int blue */

    struct colors { int i, j; };    // ILLEGAL: colors duplicate tag
    double red = 2;                 // ILLEGAL: redefinition of red
}

blue = 37;                         // back in int blue scope
```

Void Type

`void` is a special type indicating the absence of any value. There are no objects of `void`; instead, `void` is used for deriving more complex types.

Void Functions

Use the `void` keyword as a function return type if the function does not return a value.

```
void print_temp(char temp) {
  Lcd_Out_Cp("Temperature:");
  Lcd_Out_Cp(temp);
  Lcd_Chr_Cp(223);  // degree character
  Lcd_Chr_Cp('C');
}
```

Use `void` as a function heading if the function does not take any parameters. Alternatively, you can just write empty parentheses:

```
main(void) {  // same as main()
 ...
}
```

Generic Pointers

Pointers can be declared as `void`, which means that they can point to any type. These pointers are sometimes called `generic`.

Derived Types

The derived types are also known as *structured types*. They are used as elements in creating more complex user-defined types.

The derived types include:

- arrays
- pointers
- structures
- unions

Arrays

Array is the simplest and most commonly used structured type. A variable of array type is actually an array of objects of the same type. These objects represent elements of an array and are identified by their position in array. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

Array Declaration

Array declaration is similar to variable declaration, with the brackets added after identifer:

```
type array_name[constant-expression]
```

This declares an array named as `array_name` and composed of elements of `type`. The `type` can be any scalar type (except `void`), user-defined type, pointer, enumeration, or another array. Result of `constant-expression` within the brackets determines a number of elements in array. If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is a number of elements in an array.

Each of the elements of an array is indexed from 0 to the number of elements minus one. If a number of elements is `n`, elements of array can be approached as variables `array_name[0] .. array_name[n-1]` of `type`.

Here are a few examples of array declaration:

```
#define MAX = 50
int    vector_one[10];        /* declares an array of 10 integers */
float  vector_two[MAX];        /* declares an array of 50 floats   */
float  vector_three[MAX - 20]; /* declares an array of 30 floats   */
```

Array Initialization

An array can be initialized in declaration by assigning it a comma-delimited sequence of values within braces. When initializing an array in declaration, you can omit the number of elements – it will be automatically determined according to the number of elements assigned. For example:

```
/* Declare an array which holds number of days in each month: */
int days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

/* This declaration is identical to the previous one */
int days[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

If you specify both the length and starting values, the number of starting values must not exceed the specified length. The opposite is possible, in this case the trailing “excess” elements will be assigned to some encountered runtime values from memory.

In case of array of `char`, you can use a shorter *string literal* notation. For example:

```
/* The two declarations are identical: */
const char msg1[] = {'T', 'e', 's', 't', '\0'};
const char msg2[] = "Test";
```

For more information on string literals, refer to String Constants.

Arrays n Expressions

When the name of an array comes up in expression evaluation (except with operators `&` and `sizeof`), it is implicitly converted to the pointer pointing to array’s first element. See Arrays and Pointers for more information.

Multi-dimensional Arrays

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

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

```
float m[50][20];   /* 2-dimensional array of size 50x20 */
```

A variable m is an array of 50 elements, which in turn are arrays of 20 floats each. Thus, we have a matrix of 50x20 elements: the first element is `m[0][0]`, the last one is `m[49][19]`. The first element of the 5th row would be `m[4][0]`.

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

```
int a[3][2][4];   /* 3-dimensional array of size 3x2x4 */

void func(int n[][2][4]) {  /* we can omit first dimension */
  ...
  n[2][1][3]++;   /* increment the last element*/
}

void main() {
  ...
  func(a);
}
```

You can initialize a multi-dimensional array with an appropriate set of values within braces. For example:

```
int a[3][2]  = {{1,2}, {2,6}, {3,7}};
```

Pointers

Pointers are special objects for holding (or “pointing to”) memory addresses. In the *mikroC PRO for PIC*, address of an object in memory can be obtained by means of an unary operator `&`. To reach the pointed object, we use an indirection operator (`*`) on a pointer.

A pointer of type “pointer to object of type” holds the address of (that is, points to) an object of type. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed to include arrays, structures, and unions.

A pointer to a function is best thought of as an address, usually in a code segment, where that function’s executable code is stored; that is, the address to which control is transferred when that function is called.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for declarations, assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Pointer Declarations

Pointers are declared the same as any other variable, but with `*` ahead of identifier. A type at the beginning of declaration specifies the type of a pointed object. A pointer must be declared as pointing to some particular type, even if that type is `void`, which really means a pointer to anything. Pointers to `void` are often called *generic pointers*, and are treated as pointers to `char` in the *mikroC PRO for PIC*.

If `type` is any predefined or user-defined type, including `void`, the declaration

```
type *p;    /* Uninitialized pointer */
```

declares p to be of type "pointer to `type`". All scoping, duration, and visibility rules are applied to the `p` object just declared. You can view the declaration in this way: if `*p` is an object of `type`, then `p` has to be a pointer to such object (object of `type`).

Note: You must initialize pointers before using them! Our previously declared pointer `*p` is not initialized (i.e. assigned a value), so it cannot be used yet.

Note: In case of multiple pointer declarations, each identifier requires an indirect operator. For example:

```
int *pa, *pb, *pc;

/* is same as: */

int *pa;
int *pb;
int *pc;
```

Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. The mikroC PRO for PIC lets you reassign pointers without typecasting, but the compiler will warn you unless the pointer was originally declared to be pointing to `void`. You can assign the `void*` pointer to the `non-void*` pointer – refer to `void` for details.

Null Pointers

A *null pointer* value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

For example:

```
int *pn = 0;       /* Here's one null pointer */

/* We can test the pointer like this: */
if ( pn == 0 ) {  ... }
```

The pointer type “pointer to void” must not be confused with the null pointer. The declaration

```
void *vp;
```

declares that `vp` is a generic pointer capable of being assigned to by any “pointer to type” value, including null, without complaint.

Assignments without proper casting between a “pointer to `type1`” and a “pointer to `type2`”, where `type1` and `type2` are different types, can invoke a compiler warning or error. If `type1` is a function and `type2` isn’t (or vice versa), pointer assignments are illegal. If `type1` is a pointer to `void`, no cast is needed. If `type2` is a pointer to `void`, no cast is needed.

Function Pointers

Function Pointers are pointers, i.e. variables, which point to the address of a function.

```
// Define a function pointer
   int (*pt2Function) (float, char, char);
```

Note: Thus functions and function pointers with different calling convention (argument order, arguments type or return type is different) are incompatible with each other.

Assign an address to a Function Pointer

It's quite easy to assign the address of a function to a function pointer. Simply take the name of a suitable and known function. Using the address operator & infront of the function's name is optional.

```
   //Assign an address to the function pointer

  int DoIt  (float a, char b, char c){ return a+b+c; }
  pt2Function = &DoIt;   // assignment
```

Example:

```
int addC(char x,char y){
  return x+y;
}

int subC(char x,char y){

  return x-y;
}
int mulC(char x,char y){
  return x*y;
}
int divC(char x,char y){

  return x/y;
}

int modC(char x,char y){

  return x%y;
}

//array of pointer to functions that receive two chars and returns
int
int  (*arrpf[])(char,char) = { addC ,subC,mulC,divC,modC};

int res;
char i;
void main() {

   for (i=0;i<5;i++){
     res = arrpf[i](10,20);
   }
}
```

Pointer Arithmetic

Pointer arithmetic in the *mikroC PRO for PIC* is limited to:

- assigning one pointer to another,
- comparing two pointers,
- comparing pointer to zero,
- adding/subtracting pointer and an integer value,
- subtracting two pointers.

The internal arithmetic performed on pointers depends on the memory specifier in force and the presence of any overriding pointer modifiers. When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects.

Arrays and pointers

Arrays and pointers are not completely independent types in the mikroC PRO for PIC. When the name of an array comes up in expression evaluation (except with operators `&` and `sizeof`), it is implicitly converted to the pointer pointing to array's first element. Due to this fact, arrays are not modifiable lvalues.

Brackets `[ ]` indicate array subscripts. The expression

```
id[exp]
```

is defined as

```
*((id) + (exp))
```

where either:
- `id` is a pointer and `exp` is an integer, or
- `id` is an integer and `exp` is a pointer.

The following statements are true:

```
&a[i]   =  a + i
 a[i]   =  *(a + i)
```

According to these guidelines, it can be written:

```
pa = &a[4];         // pa points to a[4]
x = *(pa + 3);      // x = a[7]
/* .. but: */
y = *pa + 3;        // y = a[4] + 3
```

Also the care should be taken when using operator precedence:

```
*pa++;              // Equal to *(pa++), increments the pointer
(*pa)++;            // Increments the pointed object!
```

The following examples are also valid, but better avoid this syntax as it can make the code really illegible:

```
(a + 1)[i] = 3;
// same as: *((a + 1) + i) = 3, i.e. a[i + 1] = 3

(i + 2)[a] = 0;
// same as: *((i + 2) + a) = 0, i.e. a[i + 2] = 0
```

Assignment and Comparison

The simple assignment operator (`=`) can be used to assign value of one pointer to another if they are of the same type. If they are of different types, you must use a typecast operator. Explicit type conversion is not necessary if one of the pointers is generic (of the `void` type).

Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

Two pointers pointing to the same array may be compared by using relational operators `==`, `!=`, `<`, `<=`, `>`, and `>=`. Results of these operations are the same as if they were used on subscript values of array elements in question:

```
int *pa = &a[4], *pb = &a[2];

if (pa == pb) {... /* won't be executed as 4 is not equal to 2 */ }
if (pa > pb)  {... /* will be executed as 4 is greater than 2 */ }
```

You can also compare pointers to zero value – testing in that way if the pointer actually points to anything. All pointers can be successfuly tested for equality or inequality to null:

```
if (pa == 0) { ... }
if (pb != 0) { ... }
```

Note: Comparing pointers pointing to different objects/arrays can be performed at programmer’s own responsibility — a precise overview of data’s physical storage is required.

Pointer Addition

You can use operators `+`, `++`, and `+=` to add an integral value to a pointer. The result of addition is defined only if the pointer points to an element of an array and if the result is a pointer pointing to the same array (or one element beyond it).

If a pointer is declared to point to `type`, adding an integral value `n` to the pointer increments the pointer value by `n * sizeof(type)` as long as the pointer remains within the legal range (first element to one beyond the last element). If `type` has a size of 10 bytes, then adding 5 to a pointer to `type` advances the pointer 50 bytes in memory. In case of the `type` type, the size of a step is one byte.

For example:

```
int a[10];           /* array a containing 10 elements of type int */
int *pa = &a[0];     /* pa is pointer to int, pointing to a[0] */
*(pa + 3) = 6;        /* pa+3 is a pointer pointing to a[3], so a[3]
now equals 6 */
pa++;                /* pa now points to the next element of array a:
a[1] */
```

There is no such element as "one past the last element", of course, but the pointer is allowed to assume such value. C "guarantees" that the result of addition is defined even when pointing to one element past array. If P points to the last array element, `P + 1` is legal, but `P + 2` is undefined.

This allows you to write loops which access the array elements in a sequence by means of incrementing pointer — in the last iteration you will have the pointer pointing to one element past the array, which is legal. However, applying an indirection operator (*) to a "pointer to one past the last element" leads to undefined behavior.

For example:

```
void f (some_type a[], int n) {
    /* function f handles elements of array a; */
    /* array a has n elements of type some_type */
  int i;
  some_type *p=&a[0];

  for ( i = 0; i < n; i++ ) {
          /* .. here we do something with *p .. */
    p++;  /* .. and with the last iteration p exceeds
                the last element of array a */
  }
  /* at this point, *p is undefined! */
}
```

Pointer Subtraction

Similar to addition, you can use operators `-,` `--` , and `-=` to subtract an integral value from a pointer.

Also, you may subtract two pointers. The difference will be equal to the distance between two pointed addresses, in bytes.

For example:

```
int a[10];
int *pi1 = &a[0];
int *pi2 = &a[4];
i = pi2 - pi1;       /* i equals 8 */
pi2 -= (i >> 1);     /* pi2 = pi2 - 4: pi2 now points to [0] */
```

Structures

A structure is a derived type usually representing a user-defined collection of named members (or components). These members can be of any type, either fundamental or derived (with some restrictions to be discussed later), in any sequence. In addition, a structure member can be a bit field.

Unlike arrays, structures are considered to be single objects. The *mikroC PRO for PIC* structure type lets you handle complex data structures almost as easily as single variables.

Note: the *mikroC PRO for PIC* does not support anonymous structures (ANSI divergence).

Structure Declaration and Initialization

Structures are declared using the keyword `struct`::

```
struct tag {member-declarator-list};
```

Here, `tag` is the name of a structure; `member-declarator-list` is a list of structure members, actually a list of variable declarations. Variables of structured type are declared the same as variables of any other type.

The member type cannot be the same as the struct type being currently declared. However, a member can be a pointer to the structure being declared, as in the following example:

```
struct mystruct {mystruct s;};    /* illegal! */
struct mystruct {mystruct *ps;}; /* OK */
```

Also, a structure can contain previously defined structure types when declaring an instance of declared structure. Here is an example:

```
/* Structure defining a dot: */
struct Dot {float x, y;};

/* Structure defining a circle: */
struct Circle {
  float r;
  struct Dot center;
} o1, o2;
/* declare variables o1 and o2 of Circle */
```

Note that the structure tag can be omitted, but then additional objects of this type cannot be declared elsewhere. For more information, see the Untagged Structures below.

Structure is initialized by assigning it a comma-delimited sequence of values within braces, similar to array. For example:

```
/* Referring to declarations from the example above: */

/* Declare and initialize dots p and q: */
struct Dot p = {1., 1.}, q = {3.7, -0.5};

/* Declare and initialize circle o1: */
struct Circle o1 = {1., {0., 0.}};  // radius is 1, center is at (0, 0)
```

Incomplete Declarations

Incomplete declarations are also known as forward declarations. A pointer to a structure type `A` can legally appear in the declaration of another structure `B` before `A` has been declared:

```
struct A;  // incomplete
struct B {struct A *pa;};
struct A {struct B *pb;};
```

The first appearance of `A` is called incomplete because there is no definition for it at that point. An incomplete declaration is allowed here, because the definition of `B` doesn't need the size of `A`.

Untagged Structures and Typedefs

If the structure tag is omitted, an `untagged structure` is created. The untagged structures can be used to declare the identifiers in the comma-delimited `member-declarator-list` to be of the given structure type (or derived from it), but additional objects of this type cannot be declared elsewhere.
It is possible to create a typedef while declaring a structure, with or without tag:

```
/* With tag: */
typedef struct mystruct { ... } Mystruct;
Mystruct s, *ps, arrs[10];  /* same as struct mystruct s, etc. */

/* Without tag: */
typedef struct { ... } Mystruct;
Mystruct s, *ps, arrs[10];
```

Usually, there is no need to use both `tag` and `typedef`: either can be used in structure type declarations.

Untagged structure and union members are ignored during initialization.

Note: See also Working with structures.

WORKING WITH STRUCTURES

Structures represent user-defined types. A set of rules regarding the application of structures is strictly defined.

Assignment

Variables of the same structured type may be assigned one to another by means of simple assignment operator (=). This will copy the entire contents of the variable to destination, regardless of the inner complexity of a given structure.
Note that two variables are of the same structured type only if they are both defined by the same instruction or using the same type identifier. For example:

```
/* a and b are of the same type: */
struct { int m1, m2;} a, b;

/* But c and d are _not_ of the same type although
  their structure descriptions are identical: */
struct { int m1, m2;} c;
struct { int m1, m2;} d;
```

Size of Structure

The size of the structure in memory can be retrieved by means of the operator `sizeof`. It is not necessary that the size of the structure is equal to the sum of its members' sizes. It is often greater due to certain limitations of memory storage.

Structures and Functions

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void);      /* func1() returns a structure */
mystruct *func2(void);    /* func2() returns pointer to structure */
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s;);        /* directly */
void func2(mystruct *sptr;);   /* via a pointer */
```

Structure Member Access

Structure and union members are accessed using the following two selection operators:

- `.` (period)
- `->` (right arrow)

The operator `.` is called the direct member selector and it is used to directly access one of the structure's members. Suppose that the object `s` is of the struct type `S` and `m` is a member identifier of the type `M` declared in `s`, then the expression

```
s.m    // direct access to member m
```

is of the type `M`, and represents the member object `m` in `S`.

The operator `->` is called the indirect (or pointer) member selector. Suppose that the object `s` is of the struct type `S` and `ps` is a pointer to `s`. Then if m is a member identifier of the type `M` declared in `s`, the expression

```
ps->m    // indirect access to member m;
          // identical to (*ps).m
```

is of the type `M`, and represents the member object `m` in `s`. The expression `ps->m` is a convenient shorthand for `(*ps).m`

For example:

```
struct mystruct {
  int i;
  char str[21];
  double d;
} s, *sptr = &s;

...

s.i = 3;              // assign to the i member of mystruct s
sptr -> d = 1.23;    // assign to the d member of mystruct s
```

The expression `s.m` is lvalue, providing that `s` is lvalue and m is not an array type. The expression `sptr->m` is an lvalue unless `m` is an array type.

Accessing Nested Structures

If the structure `B` contains a field whose type is the structure `A`, the members of `A` can be accessed by two applications of the member selectors:

```
struct A {
  int j; double x;
};
struct B {
  int i; struct A aa; double d;
} s, *sptr;

...

s.i = 3;            // assign 3 to the i member of B
s.aa.j = 2;         // assign 2 to the j member of A
sptr->d = 1.23;     // assign 1.23 to the d member of B
sptr->aa.x = 3.14;  // assign 3.14 to x member of A
```

Structure Uniqueness

Each structure declaration introduces a unique structure type, so that in

```
struct A {
  int i,j; double d;
} aa, aaa;

struct B {
  int i,j; double d;
} bb;
```

the objects `aa` and `aaa` are both of the type struct `A`, but the objects `aa` and `bb` are of different structure types. Structures can be assigned only if the source and destination have the same type:

```
aa = aaa;    /* OK: same type, member by member assignment */
aa = bb;     /* ILLEGAL: different types */

/* but you can assign member by member: */
aa.i = bb.i;
aa.j = bb.j;
aa.d = bb.d;
```

Unions

Union types are derived types sharing many of syntactic and functional features of structure types. The key difference is that a union members share the same memory space.

Note: The *mikroC PRO for PIC* does not support anonymous unions (ANSI divergence).

Unions Declaration

Unions have the same declaration as structures, with the keyword `union` used instead of `struct`

```
union tag {  member-declarator-list };
```

Unlike structures' members, the value of only one of union's members can be stored at any time. Here is a simple example:

```
union myunion {   // union tag is 'myunion'
  int i;
  double d;
  char ch;
} mu, *pm;
```

The identifier `mu`, of the type `myunion`, can be used to hold a 2-byte `int`, 4-byte `double` or single-byte `char`, but only one of them at a certain moment. The identifier `pm` is a pointer to union `myunion`.

Size of Union

The size of a union is the size of its largest member. In our previous example, both `sizeof(union myunion)` and `sizeof(mu)` return 4, but 2 bytes are unused (padded) when `mu` holds the `int` object, and 3 bytes are unused when `mu` holds `char`.

Union Member Access

Union members can be accessed with the structure member selectors (. and ->), be careful when doing this:

```
/* Referring to declarations from the example above: */
pm = &mu;
mu.d = 4.016;
tmp = mu.d;  // OK: mu.d = 4.016
```

```
tmp = mu.i;  // peculiar result

pm->i = 3;
tmp = mu.i;  // OK: mu.i = 3
```

The third line is legal, since `mu.i` is an integral type. However, the bit pattern in `mu.i` corresponds to parts of the previously assigned `double`. As such, it probably won't provide an useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

Bit Fields

Bit fields are specified numbers of bits that may or may not have an associated identifier. Bit fields offer a way of subdividing structures into named parts of user-defined sizes.

Structures and unions can contain bit fields that can be up to 16 bits.

You cannot take the address of a bit field.

Note: If you need to handle specific bits of 8-bit variables (`char` and `unsigned short`) or registers, you don't need to declare bit fields. Much more elegant solution is to use the *mikroC PRO for PIC*'s intrinsic ability for individual bit access — see Accessing Individual Bits for more information.

Bit Fields Declaration

Bit fields can be declared only in structures and unions. Declare a structure normally and assign individual fields like this (fields need to be `unsigned`):

```
struct tag {
  unsigned bitfield-declarator-list;
}
```

Here, `tag` is an optional name of the structure; `bitfield-declarator-list` is a list of bit fields. Each component identifer requires a colon and its width in bits to be explicitly specified. Total width of all components cannot exceed two bytes (16 bits).

As an object, bit fields structure takes two bytes. Individual fields are packed within two bytes from right to left. In `bitfield-declarator-list`, you can omit identifier(s) to create an artificial “padding”, thus skipping irrelevant bits.

For example, if there is a need to manipulate only bits 2–4 of a register as one block, create a structure like this:

```
struct {
  unsigned  : 2,  // Skip bits 0 and 1, no identifier here
     mybits : 3;  // Relevant bits 2, 3 and 4
                   // Bits 5, 6 and 7 are implicitly left out
} myreg;
```

Here is an example:

```
typedef struct {
  lo_nibble  : 4;
  hi_nibble  : 4;
  high_byte  : 8;} myunsigned;
```

which declares the structured type `myunsigned` containing three components: `lo_nibble` (bits 3..0), `hi_nibble` (bits 7..4) and `high_byte` (bits 15..8).

Bit Fields Access

Bit fields can be accessed in the same way as the structure members. Use direct and indirect member selector (. and `->`). For example, we could work with our previously declared `myunsigned` like this:

```
// This example writes low byte of bit field of myunsigned type to
PORT0:
myunsigned Value_For_PORT0;

void main() {
  ...
  Value_For_PORT0.lo_nibble  = 7;
  Value_For_PORT0.hi_nibble  = 0x0C;
  P0 = *(char *) (void *)&Value_For_PORT0;
      // typecasting :
        // 1. address of structure to pointer to void
        // 2. pointer to void to pointer to char
        // 3. dereferencing to obtain the value
}
```

Type Conversions

The *mikroC PRO for PIC* is a strictly typed language, with each operator, statement and function demanding appropriately typed operands/arguments. However, we often have to use objects of “mismatching” types in expressions. In that case, *type conversion* is needed.

Conversion of object of one type means that object's type is changed into another type. The *mikroC PRO for PIC* defines a set of standard conversions for built-in types, provided by compiler when necessary. For more information, refer to the Standard Conversions.

Conversion is required in the following situations:

- if a statement requires an expression of particular type (according to language definition), and we use an expression of different type,
- if an operator requires an operand of particular type, and we use an operand of different type,
- if a function requires a formal parameter of particular type, and we pass it an object of different type,
- if an expression following the keyword `return` does not match the declared function return type,
- if intializing an object (in declaration) with an object of different type.

In these situations, compiler will provide an automatic implicit conversion of types, without any programmer's interference. Also, the programmer can demand conversion explicitly by means of the *typecast* operator. For more information, refer to the Explicit Typecasting.

Standard Conversions

When using arithmetic expression, such as `a + b`, where `a` and `b` are of different arithmetic types, the *mikroC PRO for PIC* performs implicit type conversions before the expression is evaluated. These standard conversions include promotions of “lower” types to “higher” types in the interests of accuracy and consistency.

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

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

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

Details:

Here are the steps the *mikroC PRO for PIC* uses to convert the operands in an arithmetic expression:
First, any small integral types are converted according to the following rules:

1. `char` converts to `int`
2. `signed char` converts to `int`, with the same value
3. `short` converts to `int`, with the same value, sign-extended
4. `unsigned short` converts to `int`, with the same value, zero-filled
5. `enum` converts to `int`, with the same value

After this, any two values associated with an operator are either `int` (including the `long` and `unsigned` modifiers) or `float` (equivalent with `double` and `long double` in the mikroC PRO for PIC).

1. If either operand is `float`, the other operand is converted to `float`.
2. Otherwise, if either operand is `unsigned long`, the other operand is converted to `unsigned long`.
3. Otherwise, if either operand is `long`, then the other operand is converted to `long`.
4. Otherwise, if either operand is `unsigned`, then the other operand is converted to `unsigned`.
5. Otherwise, both operands are `int`.

The result of the expression is the same type as that of the two operands.
Here are several examples of implicit conversion:

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

Pointer Conversion

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast `type`* will convert a pointer to type “pointer to `type`”.

Explicit Type Concersions (Typecasting)

In most situations, compiler will provide an automatic implicit conversion of types where needed, without any user's interference. Also, the user can explicitly convert an operand to another type using the prefix unary *typecast* operator:

```
(type) object
```

This will convert `object` to a specified `type`. Parentheses are mandatory.

For example:

```
/* Let's have two variables of char type: */
char a, b;

/* Following line will coerce a to unsigned int: */
(unsigned int) a;

/* Following line will coerce a to double,
   then coerce b to double automatically,
   resulting in double type value: */
(double) a + b;    // equivalent to ((double) a) + b;
```

Declarations

A declaration introduces one or several names to a program – it informs the compiler what the name represents, what its type is, what operations are allowed with it, etc. This section reviews concepts related to declarations: declarations, definitions, declaration specifiers, and initialization.

The range of objects that can be declared includes:

- Variables
- Constants
- Functions
- Types
- Structure, union and enumeration tags
- Structure members
- Union members
- Arrays of other types
- Statement labels
- Preprocessor macros

Declarations and Definitions

Defining declarations, also known as *definitions*, beside introducing the name of an object, also establish the creation (where and when) of an object; that is, the allocation of physical memory and its possible initialization. Referencing declarations, or just declarations, simply make their identifiers and types known to the compiler.

Here is an overview. Declaration is also a definition, except if:

- it declares a function without specifying its body
- it has the `extern` specifier, and has no initializator or body (in case of func.)
- it is the `typedef` declaration

There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

For example:

```
/* Here is a nondefining declaration of function max; */
/* it merely informs compiler that max is a function */
int max();

/* Here is a definition of function max: */
int max(int x, int y) {
  return (x >= y) ? x : y;
}

/* Definition of variable i: */
int i;

/* Following line is an error, i is already defined! */
int i;
```

Declarations and Declarators

The declaration contains specifier(s) followed by one or more identifiers (declarators). The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is terminated by a semicolon.

Declarations of variable identifiers have the following pattern:

```
storage-class [type-qualifier] type var1 [=init1], var2 [=init2], ... ;
```

where `var1, var2`,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of `type`; if omitted, `type` defaults to `int`. The specifier `storage-class` can take the values `extern, static, register`, or the default `auto`. Optional `type-qualifier` can take values `const` or `volatile`. For more details, refer to Storage Classes and Type Qualifiers.

For example:

```
/* Create 3 integer variables called x, y, and z
   and initialize x and y to the values 1 and 2, respectively: */
int x = 1, y = 2, z;    // z remains uninitialized

/* Create a floating-point variable q with static modifier,
   and initialize it to 0.25: */
static float q = .25;
```

These are all defining declarations; storage is allocated and any optional initializers are applied.

Linkage

An executable program is usually created by compiling several independent *translation units*, then linking the resulting object files with preexisting libraries. A term translation unit refers to a source code file together with any included files, but without the source lines omitted by conditional preprocessor directives. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope.

The *linkage* is a process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of two linkage attributes, closely related to their scope: external linkage or internal linkage. These attributes are determined by the placement and format of your declarations, together with an explicit (or implicit by default) use of the storage class specifier `static` or `extern`.

Each instance of a particular identifier with external linkage represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with internal linkage represents the same object or function within one file only.

Linkage Rules

Local names have internal linkage; the same identifier can be used in different files to signify different objects. Global names have external linkage; identifier signifies the same object throughout all program files.
If the same identifier appears with both internal and external linkage within the same file, the identifier will have internal linkage.

Internal Linkage Rules

1. names having file scope, explicitly declared as `static`, have internal linkage
2. names having file scope, explicitly declared as `const` and not explicitly declared as `extern`, have internal linkage
3. `typedef` names have internal linkage
4. enumeration constants have internal linkage

External Linkage Rules

1. names having file scope, that do not comply to any of previously stated internal linkage rules, have external linkage

The storage class specifiers `auto` and `register` cannot appear in an external declaration. No more than one external definition can be given for each identifier in a translation unit declared with internal linkage. An external definition is an external declaration that defines an object or a function and also allocates a storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of `sizeof`), then exactly one external definition of that identifier must be somewhere in the entire program.

Storage Classes

Associating identifiers with objects requires each identifier to have at least two attributes: storage class and type (sometimes referred to as data type). The *mikroC PRO for PIC* compiler deduces these attributes from implicit or explicit declarations in the source code.

A storage class dictates the location (data segment, register, heap, or stack) of object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). A storage class can be established by the syntax of a declaration, by its placement in the source code, or by both of these factors:

```
storage-class type identifier
```

The storage class specifiers in the *mikroC PRO for PIC* are:

- auto
- register
- static
- extern

Auto

The `auto` modifer is used to define that a local variable has a local duration. This is the default for local variables and is rarely used. `auto` can not be used with globals. See also Functions.

Register

At the moment the modifier `register` technically has no special meaning. The *mikroC PRO for PIC* compiler simply ignores requests for register allocation.

Static

A global name declared with the `static` specifier has internal linkage, meaning that it is local for a given file. See Linkage for more information.

A local name declared with the `static` specifier has static duration. Use `static` with a local variable to preserve the last value between successive calls to that function. See Duration for more information.

Extern

A name declared with the `extern` specifier has external linkage, unless it has been previously declared as having internal linkage. A declaration is not a definition if it has the `extern` specifier and is not initialized. The keyword `extern` is optional for a function prototype.

Use the `extern` modifier to indicate that the actual storage and initial value of the variable, or body of the function, is defined in a separate source code module. Functions declared with `extern` are visible throughout all source files in the program, unless the function is redefined as `static`.

See Linkage for more information.

Type Qualifiers

The type qualifiers `const` and `volatile` are optional in declarations and do not actually affect the type of declared object.

Qualifiers Const

The qualifier `const` implies that a declared object will not change its value during runtime. In declarations with the `const` qualifier all objects need to be initialized.

The *mikroC PRO for PIC* treats objects declared with the `const` qualifier the same as literals or preprocessor constants. If the user tries to change an object declared with the `const` qualifier compiler will report an error.

For example:

```
const double PI = 3.14159;
```

Qualifier Volatile

The qualifier `volatile` implies that a variable may change its value during runtime independently from the program. Use the volatile modifier to indicate that a variable can be changed by a background routine, an interrupt routine, or I/O port. Declaring an object to be volatile warns the compiler not to make assumptions concerning the value of an object while evaluating expressions in which it occurs because the value could be changed at any moment.

Typedef Specifier

The specifier `typedef` introduces a synonym for a specified type. The typedef declarations are used to construct shorter or more convenient names for types already defined by the language or declared by the user.
The specifier `typedef` stands first in the declaration:

```
typedef <type_definition> synonym;
```

The `typedef` keyword assigns synonym to `<type_definition>`. The `synonym` needs to be a valid identifier.
A declaration starting with the `typedef` specifier does not introduce an object or a function of a given type, but rather a new name for a given type. In other words, the typedef declaration is identical to a "normal" declaration, but instead of objects, it declares types. It is a common practice to name custom type identifiers with starting capital letter — this is not required by the *mikroC PRO for PIC*. For example:

```
/* Let's declare a synonym for "unsigned long int" */
typedef unsigned long int Distance;

/* Now, synonym "Distance" can be used as type identifier: */
Distance i; // declare variable i of unsigned long int
```

In the `typedef` declaration, as in any other declaration, several types can be declared at once. For example:

```
typedef int  *Pti, Array[10];
```

Here, `Pti` is a synonym for type "pointer to `int`", and `Array` is a synonym for type "array of 10 `int` elements".

asm Declarations

The *mikroC PRO for PIC* allows embedding assembly in the source code by means of the asm declaration. The declarations `_asm` and `_asm` are also allowed in the *mikroC PRO for PIC* and have the same meaning. Note that numerals cannnot be used as absolute addresses for SFR or GPR variables in assembly instructions. Symbolic names may be used instead (listing will display these names as well as addresses).

Assembly instructions can be grouped by the `asm` keyword (or `_`, or `_asm`):

```
asm {
  block of assembly instructions
}
```

There are two ways to embeding single assembly instruction to C code:

```
asm assembly instruction;
```

and

```
asm assembly instruction
```

Note: semicolon and LF are terminating asm scope for single assembly instructions. This is the reason why the following syntax is not asm block:

```
asm
  {
  block of assembly instructions
  }
```

This code will be interpreted as single empty asm line followed by C compound statement.

The *mikroC PRO for PIC* comments (both single-line and multi-line) are allowed in embedded assembly code.

if you have a global variable "g_var", that is of type long (i.e. 4 bytes), you are to access it like this:

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

If you want to know details about asm syntax supported by *mikroC PRO for PIC* it is recomended to study `asm` and lst files generated by compiler. It is also recomended to check "Include source lines in output files" checkbox in Output settings

Note: Compiler doesn't expect memory banks to be changed inside the assembly code. If the user wants to do this, then he must restore the previous bank selection.

Related topics: *mikroC PRO for PIC* specifcs

Initialization

The initial value of a declared object can be set at the time of declaration (*initialization*). A part of the declaration which specifies the initialization is called *initializer*.

Initializers for globals and `static` objects must be constants or constant expressions. The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of an object is that of the expression; the same constraints for type and conversions as for simple assignments are applied to initializations too.

For example:

```
int i = 1;
char *s = "hello";
struct complex c = {0.1, -0.2};
// where 'complex' is a structure (float, float)
```

For structures or unions with automatic storage duration, the initializer must be one of the following:

- An initializer list.
- A single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

For example:

```
struct dot {int x; int y; } m = {30, 40};
```

For more information, refer to Structures and Unions.

Also, you can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For more information, refer to Arrays.

Automatic Initialization

The *mikroC PRO for PIC* does not provide automatic initialization for objects. Uninitialized globals and objects with static duration will take random values from memory.

FUNCTIONS

Functions are central to C programming. Functions are usually defined as subprograms which return a value based on a number of input parameters. Return value of the function can be used in expressions – technically, function call is considered to be an expression like any other.

C allows a function to create results other than its return value, referred to as *side effects*. Often, the function return value is not used at all, depending on the side effects. These functions are equivalent to *procedures* of other programming languages, such as Pascal. C does not distinguish between procedure and function – functions play both roles.

Each program must have a single external function named `main` marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions have external linkage by default and are normally accessible from any file in the program. This can be restricted by using the `static` storage class specifier in function declaration (see Storage Classes and Linkage).

Note: Check the PIC Specifics for more information on functions' limitations on the PIC compliant MCUs.

Function Declaration

Functions are declared in user's source files or made available by linking precompiled libraries. The declaration syntax of the function is:

```
type function_name(parameter-declarator-list);
```

The `function_name` must be a valid identifier. This name is used to call the function; see Function Calls for more information.

`type` represents the type of function result, and can be of any standard or user-defined type. For functions that do not return value the `void` type should be used. The type can be omitted in global function declarations, and function will assume the `int` type by default.

Function type can also be a pointer. For example, `float*` means that a function result is a pointer to float. The generic pointer `void*` is also allowed.

The function cannot return an array or another function.

Within parentheses, `parameter-declarator-list` is a list of formal arguments that function takes. These declarators specify the type of each function parameter. The compiler uses this information to check validity of function calls. If the list is empty, a function does not take any arguments. Also, if the list is `void`, a function also does not take any arguments; note that this is the only case when `void` can be used as an argument's type.

Unlike variable declaration, each argument in the list needs its own type specifier and possible qualifier `const` or `volatile`.

Function Prototype

A function can be defined only once in the program, but can be declared several times, assuming that the declarations are compatible. When declaring a function, the formal argument's identifier does not have to be specified, but its type does.

This kind of declaration, commonly known as the *function prototype*, allows better control over argument number, type checking and type conversions. The name of a parameter in function prototype has its scope limited to the prototype. This allows one parameter identifier to have different name in different declarations of the same function:

```
/* Here are two prototypes of the same function: */

int test(const char*)    /* declares function test */
int test(const char*p)   /* declares the same function test */
```

Function prototypes are very useful in documenting code. For example, the function `Cf_Init` takes two parameters: Control Port and Data Port. The question is, which is which? The function prototype:

```
void Cf_Init(char *ctrlport, char *dataport);
```

makes it clear. If a header file contains function prototypes, the user can read that file to get the information needed for writing programs that call these functions. If a prototype parameter includes an identifier, then the indentifier is only used for error checking.

Function Definition

Function definition consists of its declaration and *function body*. The `function body` is technically a block – a sequence of local definitions and statements enclosed within braces `{}`. All variables declared within function body are local to the function, i.e. they have function scope.

The function itself can be defined only within the file scope, which means that function declarations cannot be nested.

To return the function result, use the `return` statement. The statement return in functions of the `void` type cannot have a parameter – in fact, the return statement can be omitted altogether if it is the last statement in the function body.

Here is a sample function definition:

```
/* function max returns greater one of its 2 arguments: */

int max(int x, int y) {
  return (x>=y) ? x : y;
}
```

Here is a sample function which depends on side effects rather than return value:

```
/* function converts Descartes coordinates (x,y) to polar (r,fi): */
#include <math.h>

void polar(double x, double y, double *r, double *fi) {
  *r = sqrt(x * x + y * y);
  *fi = (x == 0 && y == 0) ? 0 : atan2(y, x);
  return; /* this line can be omitted */
}
```

Function Reentrancy

Functions reentrancy is allowed if the function has no parameters and local variables, or if the local variables are placed in the Rx space. Remember that the PIC has stack and memory limitations which can varies greatly between MCUs.

Function Calls and Argument Conversion

Function Calls

A function is called with actual arguments placed in the same sequence as their matching formal parameters. Use the function-call operator ():

```
function_name(expression_1, ... , expression_n)
```

Each `expression` in the function call is an *actual argument*. Number and types of actual arguments should match those of formal function parameters. If types do not match, implicit type conversions rules will be applied. Actual arguments can be of any complexity, but order of their evaluation is not specified.

Upon function call, all formal parameters are created as local objects initialized by the 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 expression of the `return` statement. This means that the function call as an operand in complex expression is treated as a function result.

If the function has no result (type `void`) or the result is not needed, then the function call can be written as a self-contained expression.

In C, scalar arguments are always passed to the function by value. The function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine. A scalar object can be passed by the address if a formal parameter is declared as a pointer. The pointed object can be accessed by using the indirection operator * .

```
// For example, Soft_Uart_Read takes the pointer to error variable,
// so it can change the value of an actual argument:
Soft_Uart_Read(&error);

// The following code would be wrong; you would pass the value
// of error variable to the function:
Soft_Uart_Read(error);
```

Argument Conversions

If a function prototype has not been previously declared, the *mikroC PRO for PIC* converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard Conversions. If a function prototype is in scope, the *mikroC PRO for PIC* converts the passed argument to the type of the declared parameter according to the same conversion rules as in assignment statements.

If a prototype is present, the number of arguments must match. The types need to be compatible only to the extent that an assignment can legally convert them. The user can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Note: If the function prototype does not match the actual function definition, the *mikroC PRO for PIC* will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with the corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and actual definitions will be detected.

The compiler is also able to force arguments to change their type to a proper one. Consider the following code:

```
int limit = 32;
char ch = 'A';
long res;

// prototype
extern long func(long par1, long par2);

main() {
  ...
  res = func(limit, ch);  // function call
}
```

Since the program has the function prototype for `func`, it converts `limit` and `ch` to `long`, using the standard rules of assignment, before it places them on the stack for the call to `func`.

Without the function prototype, `limit` and `ch` would be placed on the stack as an integer and a character, respectively; in that case, the stack passed to `func` will not match size or content that `func` expects, which can cause problems.

Ellipsis ('...') Operator

The ellipsis ('...') consists of three successive periods with no whitespace intervening. An ellipsis can be used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types. For example:

```
void func (int n, char ch, ...);
```

This declaration indicates that func will be defined in such a way that calls must have at least two arguments, `int` and `char`, but can also have any number of additional arguments.

Example:

```
#include <stdarg.h>


int addvararg(char a1,...){
va_list ap;
char temp;
va_start(ap,a1);

while( temp = va_arg(ap,char))
  a1 += temp;
return a1;
}


int res;
void main() {


  res = addvararg(1,2,3,4,5,0);

  res = addvararg(1,2,3,4,5,6,7,8,9,10,0);

}
```

OPERATORS

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

- Arithmetic Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Reference/Indirect Operators
- Relational Operators
- Structure Member Selectors

- Comma Operator `,`
- Conditional Operator `? :`

- Array subscript operator `[]`
- Function call operator `()`

- `sizeof` Operator

- Preprocessor Operators `#` and `##`

Operators Presidence and Associativity

There are 15 precedence categories, some of them contain only one operator. Operators in the same category have equal precedence.

If duplicates of operators appear in the table, the first occurrence is unary and the second binary. Each category has an associativity rule: left-to-right (→), or right-to-left (←). In the absence of parentheses, these rules resolve a grouping of expressions with operators of equal precedence.

Precedence	Operands	Operators	Associativity
15	2	`()` `[]` `.` `->`	→
14	1	`!` `~` `++` `--` `+` `-` `*` `&` `(type)` **`sizeof`**	←
13	2	`*` `/` `%`	→
12	2	`+` `-`	→
11	2	`<<` `>>`	→
10	2	`<` `<=` `>` `>=`	→
9	2	`==` `!=`	→
8	2	`&`	→
7	2	`^`	→
6	2	`|`	→
5	2	`&&`	→
4	2	`||`	→
3	3	`?:`	←
2	2	`=` `*=` `/=` `%=` `+=` `-=` `&=` `^=` `|=` `<<=` `>>`	←
1	2	`,`	→

Arithmetic Operators

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results. The type `char` technically represents small integers, so the `char` variables can be used as operands in arithmetic operations.

All arithmetic operators associate from left to right.

Operator	Operation	Precedence
Binary Operators		
+	addition	12
-	subtraction	12
*	multiplication	13
/	division	13
%	modulus operator returns the remainder of integer division (cannot be used with floating points)	13
	Unary Operators	
+	unary plus does not affect the operand	14
-	unary minus changes the sign of the operand	14
++	increment adds one to the value of the operand. Postincrement adds one to the value of the operand after it evaluates; while preincrement adds one before it evaluates	14
--	decrement subtracts one from the value of the operand. Postdecrement subtracts one from the value of the operand after it evaluates; while predecrement subtracts one before it evaluates	14

Note: Operator * is context sensitive and can also represent the pointer reference operator.

Binary Arithmetic Operators

Division of two integers returns an integer, while remainder is simply truncated:

```
/* for example: */
7 / 4;           /* equals 1 */
7 * 3 / 4;       /* equals 5 */

/* but: */
7. * 3. / 4.;    /* equals 5.25 because we are working with floats */
```

Remainder operand `%` works only with integers; the sign of result is equal to the sign of the first operand:

```
/* for example: */
9 % 3;           /* equals 0 */
7 % 3;           /* equals 1 */
-7 % 3;          /* equals -1 */
```

Arithmetic operators can be used for manipulating characters:

```
'A' + 32;           /* equals 'a' (ASCII only) */
'G' - 'A' + 'a';    /* equals 'g' (both ASCII and EBCDIC) */
```

Unary Arithmetic Operators

Unary operators `++` and `--` are the only operators in C which can be either prefix (e.g. `++k`, `--k`) or postfix (e.g. `k++, k--`).

When used as prefix, operators `++` and `--` (preincrement and predecrement) add or subtract one from the value of the operand before the evaluation. When used as suffix, operators `++` and `--` (postincrement and postdecrement) add or subtract one from the value of the operand after the evaluation.

For example:

```
int j = 5;
j = ++k;      /* k = k + 1, j = k, which gives us j = 6, k = 6 */
```

but:

```
int j = 5;
j = k++;        /* j = k, k = k + 1, which gives us j = 5, k = 6 */
```

Relational Operators

Use relational operators to test equality or inequality of expressions. If an expression evaluates to be true, it returns 1; otherwise it returns 0.

All relational operators associate from left to right.

Relational Operators Overview

Operator	Operation	Precedence
==	equal	9
!=	not equal	9
>	greater than	10
<	less than	10
>=	greater than or equal	10
<=	less than or equal	10

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)) */
```

Do not forget that relational operators return either 0 or 1. Consider the following examples:

```
/* ok: */
5 > 7                  /* returns 0 */
10 <= 20               /* returns 1 */

/* this can be tricky: */
8 == 13 > 5            /* returns 0, as: 8 == (13 > 5)  ?  8 == 1
?  0 */
14 > 5 < 3            /* returns 1, as: (14 > 5) < 3  ?  1 < 3  ?
1 */
a < b < 5              /* returns 1, as: (a < b) < 5  ?  (0 or 1)
< 5  ?  1*/
```

Bitwise Operators

Use the bitwise operators to modify individual bits of numerical operands.

Bitwise operators associate from left to right. The only exception is the bitwise complement operator `~` which associates from right to left.

Bitwise Operators Overview

Operator	Operation	Precedence
&	bitwise AND; compares pairs of bits and returns 1 if both bits are 1, otherwise returns 0	8
|	bitwise (inclusive) OR; compares pairs of bits and returns 1 if either or both bits are 1, otherwise returns 0	6
^	bitwise exclusive OR (XOR); compares pairs of bits and returns 1 if the bits are complementary, otherwise returns 0	7
~	bitwise complement (unary); inverts each bit	14
<<	bitwise shift left; moves the bits to the left, discards the far left bit and assigns 0 to the far right bit.	11
>>	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the far left bit, otherwise sign extends	11

Logical Operations on Bit Level

&	**0**	**1**
0	0	0
1	0	1

|	**0**	**1**
0	0	1
1	1	1

^	**0**	**1**
0	0	1
1	1	0

~	**0**	**1**
	1	0

Bitwise operators `&`, `|` and `^` perform logical operations on the appropriate pairs of bits of their operands. Operator `~` complements each bit of its operand. For example:

```
0x1234 & 0x5678         /* equals 0x1230 */

/* because ..

0x1234 : 0001 0010 0011 0100
0x5678 : 0101 0110 0111 1000
----------------------------
  &    : 0001 0010 0011 0000

.. that is, 0x1230 */

/* Similarly: */

0x1234 | 0x5678;        /* equals 0x567C */
0x1234 ^ 0x5678;        /* equals 0x444C */
~ 0x1234;              /* equals 0xEDCB */
```

Note: Operator `&` can also be a pointer reference operator. Refer to Pointers for more information.

Bitwise Shift Operators

Binary operators `<<` and `>>` 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.

With shift left (`<<`), far left 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 2n if all discarded bits are zero. This is also true for signed operands if all discarded bits are equal to a sign bit.

```
000001 <<  5;     /* equals 000040 */
0x3801 <<  4;     /* equals 0x8010, overflow! */
```

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

```
0xFF56  >>  4;     /* equals 0xFFF5 */
0xFF56u >>  4;     /* equals 0x0FF5 */
```

Bitwise versus Logical

Do not forget of the principle difference between how bitwise and logical operators work. For example:

```
0222222 &  0555555;     /* equals 000000 */
0222222 && 0555555;     /* equals 1 */

~ 0x1234;               /* equals 0xEDCB */
! 0x1234;               /* equals 0 */
```

Logical Operators

Operands of logical operations are considered true or false, that is non-zero or zero. Logical operators always return 1 or 0. Operands in a logical expression must be of scalar type.

Logical operators `&&` and `||` associate from left to right. Logical negation operator `!` associates from right to left.

Logical Operators Overview

Operator	Operation	Precedence
&&	logical AND	5
||	logical OR	4
!	logical negation	14

Logical Operators

&&	0	x
0	0	0
x	0	1

||	0	x
0	0	1
x	1	1

!	0	x
	1	0

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

```
c >= '0' && c <= '9';     /* reads as: (c >= '0') && (c <= '9') */
a + 1 == b || ! f(x);     /* reads as: ((a + 1) == b) || (! (f(x))) */
```

Logical AND `&&` returns 1 only if both expressions evaluate to be nonzero, otherwise returns 0. If the first expression evaluates to false, the second expression will not be evaluated. For example:

```
a > b && c < d;      /* reads as  (a > b) && (c < d) */
/* if (a > b) is false (0), (c < d) will not be evaluated */
```

Logical OR `||` returns 1 if either of expression evaluates to be nonzero, otherwise returns 0. If the first expression evaluates to true, the second expression is not evaluated. For example:

```
a && b || c && d;  /* reads as: (a && b) || (c && d) */
/* if (a && b) is true (1), (c && d) will not be evaluated */
```

Logical Expressions and Side Effects

General rule regarding complex logical expressions is that the evaluation of consecutive logical operands stops at the very moment the final result is known. For example, if we have an expression `a && b && c` where `a` is false (0), then operands `b` and `c` will not be evaluated. This is very important if `b` and `c` are expressions, as their possible side effects will not take place!

Logical versus Bitwise

Be aware of the principle difference between how bitwise and logical operators work. For example:

```
0222222 &  0555555      /* equals 000000 */
0222222 && 0555555      /* equals 1 */

~ 0x1234                 /* equals 0xEDCB */
! 0x1234                 /* equals 0 */
```

Conditional Operator ? :

The conditional operator `? :` is the only ternary operator in C. Syntax of the conditional operator is:

```
expression1 ? expression2 : expression3
```

The `expression1` is evaluated first. If its value is true, then `expression2` evaluates and `expression3` is ignored. If expression1 evaluates to false, then `expression3` evaluates and `expression2` is ignored. The result will be a value of either `expression2` or `expression3` depending upon which of them evaluates.

Note: The fact that only one of these two expressions evaluates is very important if they are expected to produce side effects!

Conditional operator associates from right to left.
Here are a couple of practical examples:

```
/* Find max(a, b): */
max = ( a > b ) ? a : b;

/* Convert small letter to capital: */
/* (no parentheses are actually necessary) */
c = ( c >= 'a' && c <= 'z' ) ? ( c - 32 ) : c;
```

Conditional Operator Rules

`expression1` must be a scalar expression; `expression2` and `expression3` must obey one of the following rules:

1. Both expressions have to be of arithmetic type. `expression2` and `expression3` are subject to usual arithmetic conversions, which determines the resulting type.
2. Both expressions have to be of compatible struct or union types. The resulting type is a structure or union type of `expression2` and `expression3`.
3. Both expressions have to be of `void` type. The resulting type is `void`.
4. Both expressions have to be of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type qualified with all type qualifiers of the types pointed to by both expressions.
5. One expression is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all type qualifiers of the types pointed to by both expressions.
6. One expression is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of `void`. The resulting type is that of the non-pointer-to-`void` expression.

Assignment Operators

Unlike many other programming languages, C treats value assignment as operation (represented by an operator) rather than instruction.

Simple Assignment Operator

For a common value assignment, a simple assignment operator (=) is used:

```
expression1=expression2
```

The `expression1` is an object (memory location) to which the value of `expression2` is assigned. Operand `expression1` has to be lvalue and `expression2` can be any expression. The assignment expression itself is not lvalue.

If `expression1` and `expression2` are of different types, the result of the `expression2` will be converted to the type of `expression1`, if necessary. Refer to Type Conversions for more information.

Compound Assignment Operator

C allows more comlex assignments by means of compound assignment operators. The syntax of compound assignment operators is:

```
expression1 op = expression2
```

where op can be one of binary operators `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, or `>>`.

Thus, we have 10 different compound assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=` and `>>=`. All of them associate from right to left. Spaces separating compound operators (e.g. `+ =`) will generate an error.

Compound assignment has the same effect as

```
expression1 = expression1 op expression2
```

except the lvalue `expression1` is evaluated only once. For example, `expression1+= expression2` is the same as `expression1 = expression1 + expression2`.

Assignment Rules

For both simple and compound assignment, the operands `expression1` and `expression2` must obey one of the following rules:

1. `expression1` is of qualified or unqualified arithmetic type and `expression2` is of arithmetic type.
2. `expression1` has a qualified or unqualified version of structure or union type compatible with the type of `expression2`.
3. `expression1` and `expression2` are pointers to qualified or unqualified versions of compatible types and the type pointed to by left has all qualifiers of the type pointed to by right.
4. Either `expression1` or `expression2` is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void. The type pointed to by left has all qualifiers of the type pointed to by right.
5. `expression1` is a pointer and `expression2` is a null pointer constant.

Sizeof Operator

The prefix unary operator `sizeof` returns an integer constant that represents the size of memory space (in bytes) used by its operand (determined by its type, with some exceptions).
The operator `sizeof` can take either a type identifier or an unary expression as an operand. You cannot use `sizeof` with expressions of function type, incomplete types, parenthesized names of such types, or with lvalue that designates a bit field object.

Sizeof Applied to Expression

If applied to expression, the size of an operand is determined without evaluating the expression (and therefore without side effects). The result of the operation will be the size of the type of the expression’s result.

Sizeof Applied to Type

If applied to a type identifier, `sizeof` returns the size of the specified type. The unit for type size is `sizeof(char)` which is equivalent to one byte. The operation `sizeof(char)` gives the result 1, whether `char` is `signed` or `unsigned`.

Thus:

```
sizeof(char)             /* returns 1 */
sizeof(int)              /* returns 2 */
sizeof(unsigned long)    /* returns 4 */
sizeof(float)            /* returns 4 */
```

When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type):

```
int i, j, a[10];
...
j = sizeof(a[1]);   /* j = sizeof(int) = 2 */
i = sizeof(a);      /* i = 10*sizeof(int) = 20 */

/* To get the number of elements in an array: */
int num_elem = i/j;
```

If the operand is a parameter declared as array type or function type, `sizeof` gives the size of the pointer. When applied to structures and unions, `sizeof` gives the total number of bytes, including any padding. The operator `sizeof` cannot be applied to a function.

EXPRESSION

Expression is a sequence of operators, operands, and punctuators that specifies a computation. Formally, expressions are defined recursively: subexpressions can be nested without formal limit. However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.

In ANSI C, the *primary expressions* are: constant (also referred to as literal), identifier, and (`expression`), defined recursively.

Expressions are evaluated according to a certain conversion, grouping, associativity and precedence rules, which depends on the operators used, 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 the *mikroC PRO for PIC*.

Expressions can produce lvalue, rvalue, or no value. Expressions might cause side effects whether they produce a value or not.

Comma Expressions

One of the specifics of C is that it allows using of comma as a sequence operator to form so-called *comma expressions* or *sequences*. Comma expression is a comma-delimited list of expressions – it is formally treated as a single expression so it can be used in places where an expression is expected. The following sequence:

```
expression_1, expression_2;
```

results in the left-to-right evaluation of each `expression`, with the value and type of `expression_2` giving the result of the whole expression. Result of `expression_1` is discarded.

Binary operator comma (,) has the lowest precedence and associates from left to right, so that `a, b, c` is the same as `(a, b), c`. This allows writing sequences with any number of expressions:

```
expression_1, expression_2, ... expression_n;
```

which results in the left-to-right evaluation of each `expression`, with the value and type of `expression_n` giving the result of the whole expression. Results of other `expressions` are discarded, but their (possible) side-effect do occur.

For example:

```
result = ( a = 5, b /= 2, c++ );
/* returns preincremented value of variable c,
   but also intializes a, divides b by 2 and increments c */

result = ( x = 10, y = x + 3, x--, z -= x * 3 - --y );
/* returns computed value of variable z,
   and also computes x and y */
```

Note

Do not confuse comma operator (sequence operator) with comma punctuator which separates elements in a function argument list and initializator lists. To avoid ambiguity with commas in function argument and initializer lists, use parentheses. For example,

```
func(i, (j = 1, j + 4), k);
```

calls the function `func` with three arguments (i, 5, k), not four.

STATEMENTS

Statements specify a flow of control as the program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code.

Statements can be roughly divided into:

- Labeled Statements
- Expression Statements
- Selection Statements
- Iteration Statements (Loops)
- Jump Statements
- Compound Statements (Blocks)

Labeled Statements

Each statement in a program can be labeled. A label is an identifier added before the statement like this:

```
label_identifier: statement;
```

There is no special declaration of a label – it just "tags" the statement. `label_identifier` has a function scope and the same label cannot be redefined within the same function.

Labels have their own namespace: label identifier can match any other identifier in the program.

A statement can be labeled for two reasons:

1. The label identifier serves as a target for the unconditional goto statement,

2. The label identifier serves as a target for the switch statement. For this purpose, only `case` and `default` labeled statements are used:

```
case constant-expression : statement
default : statement
```

Expression Statements

Any expression followed by a semicolon forms an expression statement:

```
expression;
```

The *mikroC PRO for PIC* executes an expression statement by evaluating the `expression`. All side effects from this evaluation are completed before the next statement starts executing. Most of expression statements are assignment statements or function calls.

A `null statement` is a special case, consisting of a single semicolon (`;`). The null statement does nothing, and therefore is useful in situations where the *mikroC PRO for PIC* syntax expects a statement but the program does not need one. For example, a null statement is commonly used in "empty" loops:

```
for (; *q++ = *p++ ;);  /* body of this loop is a null statement */
```

Selection Statements

Selection or flow-control statements select one of alternative courses of action by testing certain values. There are two types of selection statements:

- if
- switch

If Statement

The `if` statement is used to implement a conditional statement. The syntax of the `if` statement is:

```
if (expression) statement1 [else statement2]
```

If `expression` evaluates to true, `statement1` executes. If `statement` is false, `statement2` executes. The expression must evaluate to an integral value; otherwise, the condition is ill-formed. Parentheses around the `expression` are mandatory.

The `else` keyword is optional, but no statements can come between `if` and `else`.

Nested If Statement

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) statement1
else if (expression2)
  if (expression3) statement2
  else statement3        /* this belongs to: if (expression3) */
else statement4          /* this belongs to: if (expression2) */
```

Note

`#if` and `#else` preprocessor statements (directives) look similar to `if` and `else` statements, but have very different effects. They control which source file lines are compiled and which are ignored.

Switch Statements

The `switch` statement is used to pass control to a specific program branch, based on a certain condition. The syntax of the `switch` statement is:

```
switch (expression) {
  case constant-expression_1 : statement_1;
    .
    .
    .
  case constant-expression_n : statement_n;
  [ default : statement;]
}
```

First, the `expression` (condition) is evaluated. The `switch` statement then compares it to all available constant-expressions following the keyword `case`. If a match is found, `switch` passes control to that matching `case` causing the `statement` following the match evaluates. Note that `constant-expressions` must evaluate to integer. It is not possible to have two same `constant expressions` evaluating to the same value.

Parentheses around `expression` are mandatory.

Upon finding a match, program flow continues normally: the following instructions will be executed in natural order regardless of the possible `case` label. If no case satisfies the condition, the `default` case evaluates (if the label `default` is specified).
For example, if a variable `i` has value between 1 and 3, the following switch would

always return it as 4:

```
switch (i) {
  case 1: i++;
  case 2: i++;
  case 3: i++;
}
```

To avoid evaluating any other cases and relinquish control from `switch`, each `case` should be terminated with break.

Here is a simple example with `switch`. Suppose we have a variable `phase` with only 3 different states (0, 1, or 2) and a corresponding function (event) for each of these states. This is how we could switch the code to the appopriate routine:

```
switch (phase) {
  case 0: Lo();  break;
  case 1: Mid(); break;
  case 2: Hi();  break;
  case: Message("Invalid state!");
}
```

Nested Switch

Conditional `switch` statements can be nested – labels `case` and `default` are then assigned to the innermost enclosing `switch` statement.

Iteration Statements (Loops)

Iteration statements allows to loop a set of statements. There are three forms of iteration statements in the *mikroC PRO for PIC*:

- while
- do
- for

While Statement

The `while` keyword is used to conditionally iterate a statement. The syntax of the `while` statement is:

```
while (expression) statement
```

The `statement` executes repeatedly until the value of `expression` is false. The test takes place before `statement` is executed. Thus, if `expression` evaluates to false on the first pass, the loop does not execute. Note that parentheses around `expression` are mandatory.

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

```
int s = 0, i = 0;
while (i < n) {
   s += a[ i]  *  b[ i] ;
   i++;
}
```

Note that body of the loop can be a null statement. For example:

```
while (*q++ = *p++);
```

Do Statement

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

```
do statement while (expression);
```

The `statement` is executed repeatedly as long as the value of `expression` remains non-zero. The `expression` is evaluated after each iteration, so the loop will execute statement at least once.

Parentheses around `expression` are mandatory.

Note that do is the only control structure in C which explicitly ends with semicolon (`;`). Other control structures end with `statement`, which means that they implicitly include a semicolon or closing brace.

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

```
s = 0; i = 0;
do {
   s += a[ i]  *  b[ i] ;
   i++;
}  while  ( i < n );
```

For Statement

The `for` statement implements an iterative loop. The syntax of the `for` statement is:

```
for ([init-expression]; [condition-expression]; [increment-expression]) statement
```

Before the first iteration of the loop, `init-expression` sets the starting variables for the loop. You cannot pass declarations in `init-expression`.

`condition-expression` is checked before the first entry into the block; `statement` is executed repeatedly until the value of `condition-expression` is false. After each iteration of the loop, increment-expression increments a loop counter. Consequently, `i++` is functionally the same as `++i`.

All expressions are optional. If `condition-expression` is left out, it is assumed to be always true. Thus, "empty" `for` statement is commonly used to create an endless loop in C:

```
for ( ; ; ) statement
```

The only way to break out of this loop is by means of the `break` statement.

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

```
for ( s = 0, i = 0; i < n; i++ ) s += a[i] * b[i];
```

There is another way to do this:

```
for ( s = 0, i = 0; i < n; s += a[i] * b[i], i++ );  /* valid, but ugly */
```

but it is considered a bad programming style. Although legal, calculating the sum should not be a part of the incrementing expression, because it is not in the service of loop routine. Note that null statement (;) is used for the loop body.

Jump Statements

The jump statement, when executed, transfers control unconditionally. There are four such statements in the *mikroC PRO for PIC*:

- break
- continue
- goto
- return

BREAK AND CONTINUE STATEMENTS

Break Statement

Sometimes it is necessary to stop the loop within its body. Use the `break` statement within loops to pass control to the first statement following the innermost `switch,` `for,` `while,` or `do` block.

`break` is commonly used in the `switch` statements to stop its execution upon the first positive match. For example:

```
switch (state) {
  case 0: Lo();  break;
  case 1: Mid(); break;
  case 2: Hi();  break;
  default: Message("Invalid state!");
}
```

Continue Statement

The `continue` statement within loops is used to “skip the cycle”. It passes control to the end of the innermost enclosing end brace belonging to a looping construct. At that point the loop continuation condition is re-evaluated. This means that `continue` demands the next iteration if the loop continuation condition is true.

Specifically, the `continue` statement within the loop will jump to the marked position as it is shown below:

```
while (..) {
...
if (val>0) continue;
...
// continue jumps
here
}
```

```
do {
...
if (val>0) continue;
...
// continue jumps
here
while (..);
```

```
for (..;..;..) {
...
if (val>0) continue;
...
// continue jumps
here
}
```

Goto Statement

The `goto` statement is used for unconditional jump to a local label — for more information on labels, refer to Labeled Statements. The syntax of the `goto` statement is:

```
goto label_identifier;
```

This will transfer control to the location of a local label specified by `label_identifier`. The `label_identifier` has to be a name of the label within the same function in which the `goto` statement is. The goto line can come before or after the label.

`goto` is used to break out from any level of nested control structures but it cannot be used to jump into block while skipping that block's initializations – for example, jumping into loop's body, etc.

The 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 the `goto` statement is breaking out from deeply nested control structures:

```
for (...) {
  for (...) {
  ...
    if (disaster) goto Error;
  ...
  }
}
 .
 .
 .
Error: /* error handling code */
```

Return Statement

The `return` statement is used to exit from the current function back to the calling routine, optionally returning a value. The syntax is:

```
return [expression];
```

This will evaluate `expression` and return the result. Returned value will be automatically converted to the expected function type, if needed. The `expression` is optional; if omitted, the function will return a random value from memory.

Note: The statement `return` in functions of the `void` type cannot have `expression` – in fact, the `return` statement can be omitted altogether if it is the last statement in the function body.

Compound Statements (Blocks)

The compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces { }. Syntactically, the block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within the block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth up to the limits of memory.

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

```
for (i = 0; i < n; i++ ) {
   int temp = a[ i];
   a[ i]  = b[ i];
   b[ i]  = temp;
}
```

Note that, unlike other statements, compound statements do not end with semicolon (`;`), i.e. there is never a semicolon following the closing brace.

PREPROCESSOR

Preprocessor is an integrated text processor which prepares the source code for compiling. Preprocessor allows:

- inserting text from a specifed file to a certain point in the code (see File Inclusion),
- replacing specific lexical symbols with other symbols (see Macros),
- conditional compiling which conditionally includes or omits parts of the code (see Conditional Compilation).

Note that preprocessor analyzes text at token level, not at individual character level. Preprocessor is controled by means of preprocessor directives and preprocessor operators.

Preprocessor Directives

Any line in the source code with a leading `#` is taken as a *preprocessing directive* (or *control line*), unless `#` is within a string literal, in a character constant, or embedded in a comment. The initial `#` can be preceded or followed by a whitespace (excluding new lines).

A *null directive* consists of a line containing the single character `#`. This line is always ignored.

Preprocessor directives are usually placed at the beginning of the source code, but they can legally appear at any point in a program. The *mikroC PRO for PIC* preprocessor detects preprocessor directives and parses the tokens embedded in them. A directive is in effect from its declaration to the end of the program file.

Here is one commonly used directive:

```
#include <math.h>
```

For more information on including files with the `#include` directive, refer to File Inclusion.

The *mikroC PRO for PIC* supports standard preprocessor directives:

```
#  (null directive)        #if
#define                    #ifdef
#elif                      #ifndef
#else                      #include
#endif                     #line
#error                     #undef
```

Note: For the time being only funcall pragma is supported.

Line Continuation with Backslash (\)

To break directive into multiple lines end the line with a backslash (`\`):

```
#define MACRO   This directive continues to \
                the following line.
```

Macros

Macros provide a mechanism for a token replacement, prior to compilation, with or without a set of formal, function-like parameters.

Defining Macros and Macro Expansions

The `#define` directive defines a macro:

```
#define macro_identifier <token_sequence>
```

Each occurrence of `macro_identifier` in the source code following this control line will be replaced in the original position with the possibly empty `token_sequence` (there are some exceptions, which are discussed later). Such replacements are known as macro expansions.`token_sequence` is sometimes called the body of a macro. An empty token sequence results in the removal of each affected macro identifier from the source code.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in a macro expansion.`token_sequence` terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

After each individual macro expansion, a further scan is made of the newly expanded text. This allows the possibility of using nested macros: the expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into something that looks like a preprocessing directive, such directive will not be recognized by the preprocessor. Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code will not be expanded.

A macro won't be expanded during its own expansion (so `#define MACRO MACRO` won't expand indefinitely).
Here is an example:

```
/* Here are some simple macros: */
#define ERR_MSG "Out of range!"
#define EVERLOOP for( ; ; )
/* which we could use like this: */
main() {
  EVERLOOP {
    ...
    if (error) { Lcd_Out_Cp(ERR_MSG); break; }
    ...
  }
}
```

Attempting to redefine an already defined macro identifier will result in a warning unless a new definition is exactly the same token-by-token definition as the existing one. The preferred strategy when definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if `BLOCK_SIZE` is currently defined; if `BLOCK_SIZE` is not currently defined, the middle line is invoked to define it.

Macros with Parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) <token_sequence>
```

Note that there can be no whitespace between `macro_identifier` and `“(”`. The optional `arg_list` is a sequence of identifiers separated by commas, like the argument list of a C function. Each comma-delimited identifier has the role of a formal argument or placeholder.

Such macros are called by writing

```
macro_identifier(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C “functions” are implemented as macros. However, there are some important semantic differences.

The optional `actual_arg_list` must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal arg_list of the `#define` line – there must be an actual argument for each formal argument. An error will be reported if the number of arguments in two lists is not the same.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in `actual_arg_list`. Like with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Here is a simple example:

```
/* A simple macro which returns greater of its 2 arguments: */
```

```
#define _MAX(A, B) ((A) > (B)) ? (A) : (B)

/* Let's call it: */
x = _MAX(a + b, c + d);

/* Preprocessor will transform the previous line into:
x = ((a + b) > (c + d)) ? (a + b) : (c + d) */
```

It is highly recommended to put parentheses around each argument in the macro body in order to avoid possible problems with operator precedence.

Undefining Macros

The `#undef` directive is used to undefine a macro.

```
#undef macro_identifier
```

The directive `#undef` detaches any previous token sequence from `macro_identifier`; the macro definition has been forgotten, and `macro_identifier` is undefined. No macro expansion occurs within the `#undef` lines.

The state of being defined or undefined is an important property of an identifier, regardless of the actual definition. The `#ifdef` and `#ifndef` conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with `#define`, using the same or different token sequence.

File Inclusion

The preprocessor directive `#include` pulls in header files (extension .h) into the source code. Do not rely on preprocessor to include source files (extension) — see Add/Remove Files from Project for more information.

The syntax of the `#include` directive has two formats:

```
#include <header_name>
#include "header_name"
```

The preprocessor removes the `#include` line and replaces it with the entire text of a header file at that point in the source code. The placement of `#include` can therefore influence the scope and duration of any identifiers in the included file.

The difference between these two formats lies in searching algorithm employed in trying to locate the include file.

If the `#include` directive is used with the `<header_name>` version, the search is made successively in each of the following locations, in this particular order:

1. the *mikroC PRO for PIC* installation folder › “include” folder
2. user's custom search paths

The `"header_name"` version specifies a user-supplied include file; the mikroC PRO for PIC will look for the header file in the following locations, in this particular order:

1. the project folder (folder which contains the project file `.mcppi`)
2. the mikroC PRO for PIC installation folder › “include” folder
3. user's custom search paths

Explicit Path

By placing an explicit path in `header_name`, only that directory will be searched. For example:

```
#include "C:\my_files\test.h"
```

Note

There is also a third version of the `#include` directive, rarely used, which assumes that neither `<` nor `"` appear as the first non-whitespace character following `#include`:

```
#include macro_identifier
```

It assumes that macro definition that will expand `macro identifier` into a valid delimited header name with either `<header_name>` or `"header_name"` formats exists.

Preprocessor Operators

The `#` (pound sign) is a preprocessor directive when it occurs as the first non-white-space character on a line. Also, `#` and `##` perform operator replacement and merging during the preprocessor scanning phase.

Operator

In C preprocessor, a character sequence enclosed by quotes is considered a token and its content is not analyzed. This means that macro names within quotes are not expanded.

If you need an actual argument (the exact sequence of characters within quotes) as a result of preprocessing, use the `#` operator in macro body. It can be placed in front of a formal macro argument in definition in order to convert the actual argument to a string after replacement.

For example, let’s have macro `LCD_PRINT` for printing variable name and value on Lcd:

```
#define LCD_PRINT(val) Lcd_Custom_Out_Cp(#val ": "); \
                       Lcd_Custom_Out_Cp(IntToStr(val));
```

Now, the following code,

```
LCD_PRINT(temp)
```

will be preprocessed to this:

```
 Lcd_Custom_Out_Cp("temp" ": "); Lcd_Custom_Out_Cp(IntToStr(temp));
```

Operator

Operator `##` is used for *token pasting*. Two tokens can be pasted(merged) together by placing `##` in between them (plus optional whitespace on either side). The preprocessor removes whitespace and `##`, combining the separate tokens into one new token. This is commonly used for constructing identifiers.

For example, see the definition of macro `SPLICE` for pasting two tokens into one identifier:

```
#define SPLICE(x,y) x ## _ ## y
```

Now, the call `SPLICE(cnt,2)` will expand to the identifier `cnt_2`.

Note

The *mikroC PRO for PIC* does not support the older nonportable method of token pasting using `(l/**/r)`.

Conditional Compilation

Conditional compilation directives are typically used to make source programs easy to change and easy to compile in different execution environments. The mikroC PRO for PIC supports conditional compilation by replacing the appropriate source-code lines with a blank line.

All conditional compilation directives must be completed in the source or include file in which they have begun.

Directives #if, #elif, #else and #endif

The conditional directives `#if`, `#elif`, `#else`, and `#endif` work very similar to the common C conditional statements. If the expression you write after `#if` has a nonzero value, the line group immediately following the `#if` directive is retained in the translation unit.

The syntax is:

```
#if constant_expression_1
<section_1>

[ #elif constant_expression_2
<section_2>]
  ...
[ #elif constant_expression_n
<section_n>]

[ #else
<final_section>]

#endif
```

Each `#if` directive in a source file must be matched by a closing `#endif` directive. Any number of `#elif` directives can appear between `#if` and `#endif` directives, but at most one `#else` directive is allowed. The `#else` directive, if present, must be the last directive before `#endif`.

`sections` can be any program text that has meaning to compiler or preprocessor. The preprocessor selects a single section by evaluating `constant_expression` following each `#if` or `#elif` directive until it finds a true (nonzero) constant expression. The constant expressions are subject to macro expansion.

If all occurrences of constant-expression are false, or if no `#elif` directives appear, the preprocessor selects the text block after the `#else` clause. If the `#else` clause is omitted and all instances of `constant_expression` in the `#if` block are false, no section is selected for further processing.

Any processed section can contain further conditional clauses, nested to any depth. Each nested `#else`, `#elif`, or `#endif` directive belongs to the closest preceding the `#if` directive.

The net result of the preceding scenario is that only one code `section` (possibly empty) will be compiled.

Directives #ifdef and #ifndef

The `#ifdef` and `#ifndef` directives can be used anywhere `#if` can be used and they can test whether an identifier is currently defined or not. The line

```
#ifdef identifier
```

has exactly the same effect as `#if 1` if `identifier` is currently defined, and the same effect as `#if 0` if `identifier` is currently undefined. The other directive, `#ifndef`, tests true for the “not-defined” condition, producing the opposite results.

The syntax thereafter follows that of `#if`, `#elif`, `#else`, and `#endif`.

An identifier defined as `NULL` is considered to be defined.

CHAPTER 7

mikroC PRO for PIC Libraries

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

Use Library manager to include *mikroC PRO for PIC* Libraries in you project.

Hardware PIC-specific Libraries

- ADC Library
- CAN Library
- CANSPI Library
- Compact Flash Library
- EEPROM Library
- Ethernet PIC18FxxJ60 Library
- Flash Memory Library
- Graphic LCD Library
- I^{2}C Library
- Keypad Library
- LCD Library
- Manchester Code Library
- Muliti Media Card Libray
- OneWire Library
- Port Expander Library
- PrintOut Library
- PS/2 Library
- PWM Library
- RS-485 Library
- Software I^{2}C 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
- UART Library
- USB HID Library

Standard ANSI C Libraries

- ANSI C Ctype Library
- ANSI C Math Library
- ANSI C Stdlib Library
- ANSI C String Library

Miscellaneous Libraries

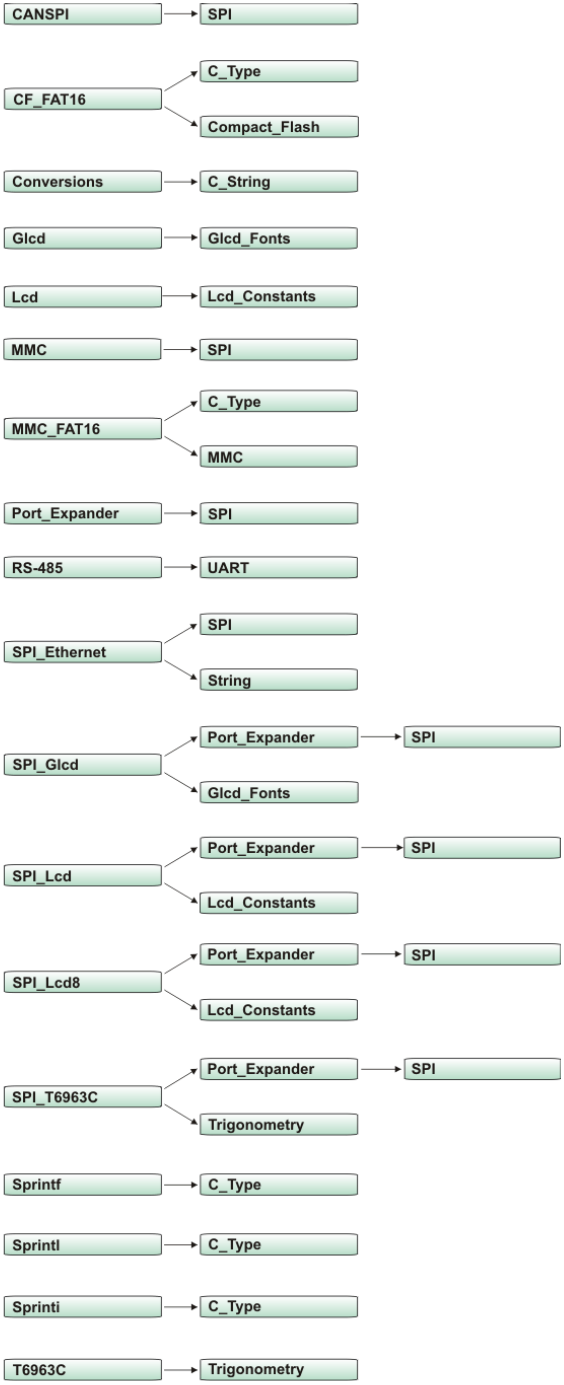
- Button Library
- Conversions Library
- Sprint Library
- Setjmp Library
- Time Library
- Trigonometry Library

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, PIC Libraries

HARDWARE LIBRARIES

- ADC Library
- CAN Library
- CANSPI Library
- Compact Flash Library
- EEPROM Library
- Ethernet PIC18FxxJ60 Library
- Flash Memory Library
- Graphic Lcd Library
- I^{2}C Library
- Keypad Library
- Lcd Library
- Manchester Code Library
- Multi Media Card Library
- OneWire Library
- Port Expander Library
- PrintOut Library
- PS/2 Library
- PWM Library
- RS-485 Library
- Software I^{2}C 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
- UART Library
- USB HID Library

ADC LIBRARY

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

ADC_Read

Prototype	`unsigned ADC_Read(unsigned short channel);`
Returns	10-bit unsigned value read from the specified channel.
Description	Initializes PIC's internal ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12TAD). 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.

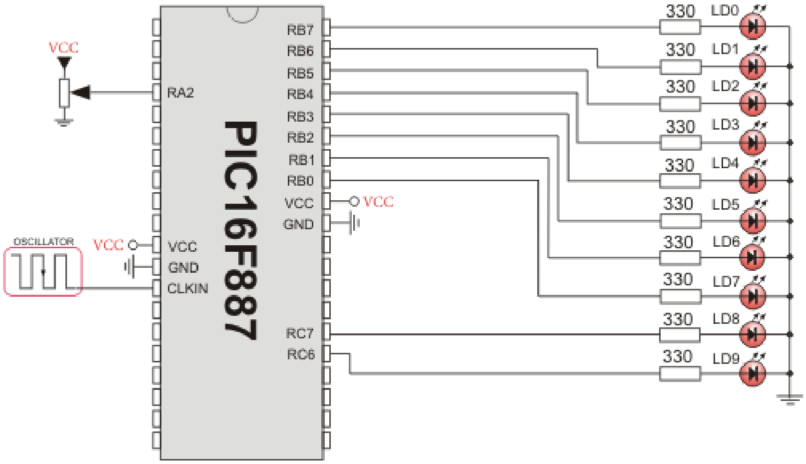
Example	`unsigned tmp;`
	`...`
	`tmp = ADC_Read(2);   // Read analog value from channel 2`

Library Example

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

```
unsigned int temp_res;

void main() {
  ANSEL  = 0x04;                 // Configure AN2 pin as analog
  TRISA  = 0xFF;                 // PORTA is input
  ANSELH = 0;                    // Configure other AN pins as digital I/O
  TRISC  = 0x3F;                 // Pins RC7, RC6 are outputs
  TRISB  = 0;                    // PORTB is output

  do {
    temp_res = ADC_Read(2);      // Get 10-bit results of AD conversion
    PORTB = temp_res;            // Send lower 8 bits to PORTB
     PORTC = temp_res >> 2;      // Send 2 most significant bits to RC7, RC6
  }  while(1);
}
```

HW Connection



ADC HW connection

CAN LIBRARY

mikroC PRO for PIC provides a library (driver) for working with the CAN module.

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

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

CAN supports two message formats:

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

Note: CAN Library is supported only by MCUs with the CAN module.

Note: Consult the CAN standard about CAN bus termination resistance.

Library Routines

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

Following routines are for the internal use by compiler only:

- RegsToCANID
- CANIDToRegs

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

CANSetOperationMode

Prototype	`void CANSetOperationMode(unsigned short mode, unsigned short wait_flag);`
Returns	Nothing.
Description	Sets CAN to requested mode, i.e. copies `mode` to CANSTAT. Parameter mode needs to be one of `CAN_OP_MODE` constants (see CAN constants). Parameter `wait_flag` needs to be either 0 or 0xFF: ■ If set to 0xFF, this is a blocking call – the function won't "return" until the requested mode is set. ■ If 0, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use `CANGetOperationMode` to verify correct operation mode before performing mode specific operation.
Requires	CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	`CANSetOperationMode(_CAN_MODE_CONFIG, 0xFF);`

CANGetOperationMode

Prototype	`unsigned short CANGetOperationMode();`
Returns	Current opmode.
Description	Function returns current operational mode of CAN module.
Requires	CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	`if (CANGetOperationMode() == _CAN_MODE_NORMAL) { ... };`

CANInitialize

Prototype	`void CANInitialize(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);`
Returns	Nothing.

Description

Initializes CAN. All pending transmissions are aborted. Sets all mask registers to 0 to allow all messages. The Config mode is internaly set by this function. Upon a execution of this function Normal mode is set. Filter registers are set according to flag value:

```
if (CAN_CONFIG_FLAGS & _CAN_CONFIG_VALID_XTD_MSG != 0)
  // Set all filters to XTD_MSG
else if (config & _CAN_CONFIG_VALID_STD_MSG != 0)
  // Set all filters to STD_MSG
else
  // Set half the filters to STD, and the rest to XTD_MSG
```

Parameters:

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

Requires	CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.

Example

```
init = _CAN_CONFIG_SAMPLE_THRICE &
        _CAN_CONFIG_PHSEG2_PRG_ON &
        _CAN_CONFIG_STD_MSG        &
        _CAN_CONFIG_DBL_BUFFER_ON &
        _CAN_CONFIG_VALID_XTD_MSG &
        _CAN_CONFIG_LINE_FILTER_OFF;
...
CANInitialize(1, 1, 3, 3, 1, init);   // initialize CAN
```

CANSetBoudRate

Prototype	`void CANSetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);`
Returns	Nothing.
Description	Sets CAN baud rate. Due to complexity of CAN protocol, you cannot simply force a bps value. Instead, use this function when CAN is in Config mode. Refer to datasheet for details. Parameters: ■ `SJW` as defined in 18XXX8 datasheet (1–4) ■ `BRP` as defined in 18XXX8 datasheet (1–64) ■ `PHSEG1` as defined in 18XXX8 datasheet (1–8) ■ `PHSEG2` as defined in 18XXX8 datasheet (1–8) ■ `PROPSEG` as defined in 18XXX8 datasheet (1–8) ■ `CAN_CONFIG_FLAGS` is formed from predefined constants (see CAN constants)
Requires	CAN must be in Config mode; otherwise the function will be ignored. CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.

Example

```
init = _CAN_CONFIG_SAMPLE_THRICE &
        _CAN_CONFIG_PHSEG2_PRG_ON &
        _CAN_CONFIG_STD_MSG        &
        _CAN_CONFIG_DBL_BUFFER_ON &
        _CAN_CONFIG_VALID_XTD_MSG &
        _CAN_CONFIG_LINE_FILTER_OFF;
...
CANSetBaudRate(1, 1, 3, 3, 1, init);
```

CANSetMask

Prototype	`void CANSetFilter(char CAN_FILTER, long value, char CAN_CONFIG_FLAGS);`
Returns	Nothing.
Description	Function sets mask for advanced filtering of messages. Given `value` is bit adjusted to appropriate buffer mask registers. Parameters: ■ `CAN_MASK` is one of predefined constant values (see CAN constants) ■ `value` is the mask register value ■ `CAN_CONFIG_FLAGS` selects type of message to filter, either `_CAN_CONFIG_XTD_MSG or _CAN_CONFIG_STD_MSG`
Requires	CAN must be in Config mode; otherwise the function will be ignored. CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.

Example	`// Set all mask bits to 1, i.e. all filtered bits are relevant:`
	`CANSetMask(_CAN_MASK_B1, -1, _CAN_CONFIG_XTD_MSG);`
	`// Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`Complement will do the trick and fill it up with ones.`

CANSetFilter

Prototype	`void CANSetFilter(char CAN_FILTER, long value, char CAN_CONFIG_FLAGS);`
Returns	Nothing.
Description	Function sets message filter. Given `value` is bit adjusted to appropriate buffer mask registers. Parameters: ■ `CAN_FILTER` is one of predefined constant values (see CAN constants) ■ `value` is the filter register value ■ `CAN_CONFIG_FLAGS` selects type of message to filter, either `_CAN_CONFIG_XTD_MSG or _CAN_CONFIG_STD_MSG`
Requires	CAN must be in Config mode; otherwise the function will be ignored. CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	`// Set id of filter B1_F1 to 3:` `CANSetFilter(_CAN_FILTER_B1_F1, 3, _CAN_CONFIG_XTD_MSG);`

CANRead

Prototype	`char CANRead(long *id, char *data, char *datalen, char *CAN_RX_MSG_FLAGS);`
Returns	Message from receive buffer or zero if no message found.
Description	Function reads message from receive buffer. If at least one full receive buffer is found, it is extracted and returned. If none found, function returns zero. Parameters: ■ `id` is message identifier ■ `data` is an array of bytes up to 8 bytes in length ■ `datalen` is data length, from 1–8. ■ `CAN_RX_MSG_FLAGS` is value formed from constants (see CAN constants)
Requires	CAN must be in mode in which receiving is possible. CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	

```
char rcv, rx, len, data[8];
long id;
// ...
rx = 0;
// ...
rcv = CANRead(id, data, len, rx);
```

CANWrite

Prototype	`unsigned short CANWrite(long id, char *data, char datalen, char CAN_TX_MSG_FLAGS);`
Returns	Returns zero if message cannot be queued (buffer full).
Description	If at least one empty transmit buffer is found, function sends message on queue for transmission. If buffer is full, function returns 0. Parameters: ■ `id` is CAN message identifier. Only 11 or 29 bits may be used depending on message type (standard or extended) ■ `data` is array of bytes up to 8 bytes in length ■ `datalen` is data length from 1–8 ■ `CAN_TX_MSG_FLAGS` is value formed from constants (see CAN constants)
Requires	CAN must be in Normal mode. CAN routines are currently supported only by P18XXX8 PIC MCUs. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	

```
char tx, data;
long id;
// ...
tx = _CAN_TX_PRIORITY_0 &
      _CAN_TX_XTD_FRAME;
// ...
CANWrite(id, data, 2, tx);
```

CAN Constants

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

CAN_OP_MODE

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

```
const char
     _CAN_MODE_BITS    = 0xE0,    // Use this to access opmode  bits
     _CAN_MODE_NORMAL  = 0x00,
     _CAN_MODE_SLEEP   = 0x20,
     _CAN_MODE_LOOP    = 0x40,
     _CAN_MODE_LISTEN  = 0x60,
     _CAN_MODE_CONFIG  = 0x80;
```

CAN_CONFIG_FLAGS

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

```
const char
     _CAN_CONFIG_DEFAULT          = 0xFF,    // 11111111

     _CAN_CONFIG_PHSEG2_PRG_BIT  = 0x01,
     _CAN_CONFIG_PHSEG2_PRG_ON   = 0xFF,    // XXXXXXX1
     _CAN_CONFIG_PHSEG2_PRG_OFF  = 0xFE,    // XXXXXXX0

     _CAN_CONFIG_LINE_FILTER_BIT = 0x02,
     _CAN_CONFIG_LINE_FILTER_ON  = 0xFF,    // XXXXXX1X
     _CAN_CONFIG_LINE_FILTER_OFF = 0xFD,    // XXXXXX0X

     _CAN_CONFIG_SAMPLE_BIT       = 0x04,
     _CAN_CONFIG_SAMPLE_ONCE      = 0xFF,    // XXXXX1XX
     _CAN_CONFIG_SAMPLE_THRICE    = 0xFB,    // XXXXX0XX

     _CAN_CONFIG_MSG_TYPE_BIT     = 0x08,
     _CAN_CONFIG_STD_MSG          = 0xFF,    // XXXX1XXX
     _CAN_CONFIG_XTD_MSG          = 0xF7,    // XXXX0XXX

     _CAN_CONFIG_DBL_BUFFER_BIT  = 0x10,
     _CAN_CONFIG_DBL_BUFFER_ON   = 0xFF,    // XXX1XXXX
     _CAN_CONFIG_DBL_BUFFER_OFF  = 0xEF,    // XXX0XXXX
```

```
     _CAN_CONFIG_MSG_BITS          = 0x60,
     _CAN_CONFIG_ALL_MSG           = 0xFF,    // X11XXXXX
     _CAN_CONFIG_VALID_XTD_MSG    = 0xDF,    // X10XXXXX
     _CAN_CONFIG_VALID_STD_MSG    = 0xBF,    // X01XXXXX
     _CAN_CONFIG_ALL_VALID_MSG    = 0x9F;    // X00XXXXX
```

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

```
init = _CAN_CONFIG_SAMPLE_THRICE &
        _CAN_CONFIG_PHSEG2_PRG_ON &
        _CAN_CONFIG_STD_MSG        &
        _CAN_CONFIG_DBL_BUFFER_ON &
        _CAN_CONFIG_VALID_XTD_MSG &
        _CAN_CONFIG_LINE_FILTER_OFF;
...
CANInitialize(1, 1, 3, 3, 1, init);   // initialize CAN
```

CAN_TX_MSG_FLAGS

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

```
const char
     _CAN_TX_PRIORITY_BITS = 0x03,
     _CAN_TX_PRIORITY_0     = 0xFC,    // XXXXXX00
     _CAN_TX_PRIORITY_1     = 0xFD,    // XXXXXX01
     _CAN_TX_PRIORITY_2     = 0xFE,    // XXXXXX10
     _CAN_TX_PRIORITY_3     = 0xFF,    // XXXXXX11

     _CAN_TX_FRAME_BIT     = 0x08,
     _CAN_TX_STD_FRAME     = 0xFF,     // XXXXX1XX
     _CAN_TX_XTD_FRAME     = 0xF7,     // XXXXX0XX

     _CAN_TX_RTR_BIT       = 0x40,
     _CAN_TX_NO_RTR_FRAME = 0xFF,     // X1XXXXXX
     _CAN_TX_RTR_FRAME     = 0xBF;     // X0XXXXXX
```

You may use bitwise AND (`&`) to adjust the appropriate flags. For example:

```
// form value to be used with CANSendMessage:
send_config = _CAN_TX_PRIORITY_0 &
               _CAN_TX_XTD_FRAME   &
               _CAN_TX_NO_RTR_FRAME;
...
CANSendMessage(id, data, 1, send_config);
```

CAN_RX_MSG_FLAGS

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

```
const char
     _CAN_RX_FILTER_BITS  = 0x07,  // Use this to access filter bits
     _CAN_RX_FILTER_1      = 0x00,
     _CAN_RX_FILTER_2      = 0x01,
     _CAN_RX_FILTER_3      = 0x02,
     _CAN_RX_FILTER_4      = 0x03,
     _CAN_RX_FILTER_5      = 0x04,
     _CAN_RX_FILTER_6      = 0x05,
     _CAN_RX_OVERFLOW      = 0x08,  // Set if Overflowed else cleared
     _CAN_RX_INVALID_MSG  = 0x10,  // Set if invalid else cleared
     _CAN_RX_XTD_FRAME     = 0x20,  // Set if XTD message else cleared
     _CAN_RX_RTR_FRAME     = 0x40,  // Set if RTR message else cleared
     _CAN_RX_DBL_BUFFERED = 0x80;  // Set if this message was hard
                                          ware double-buffered
```

You may use bitwise AND (`&`) to adjust the appropriate flags. For example:

```
if (MsgFlag & _CAN_RX_OVERFLOW != 0) {
  ...
  // Receiver overflow has occurred.
  // We have lost our previous message.
}
```

CAN_MASK

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

```
#const char
     _CAN_MASK_B1 = 0,
     _CAN_MASK_B2 = 1;
```

CAN_FILTER

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

```
const char
     _CAN_FILTER_B1_F1 = 0,
     _CAN_FILTER_B1_F2 = 1,
     _CAN_FILTER_B2_F1 = 2,
     _CAN_FILTER_B2_F2 = 3,
```

```
    _CAN_FILTER_B2_F3 = 4,
    _CAN_FILTER_B2_F4 = 5;
```

Library Example

This is a simple demonstration of CAN 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 CAN node:

```
unsigned char Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags; // can flags
unsigned char Rx_Data_Len;                // received data length in bytes
char RxTx_Data[8];                        // can rx/tx data buffer
char Msg_Rcvd;                            // reception flag
const long ID_1st = 12111, ID_2nd = 3;    // node IDs
long Rx_ID;

void main() {

  PORTC = 0;                               // clear PORTC
  TRISC = 0;                               // set PORTC as output

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

  Can_Send_Flags = _CAN_TX_PRIORITY_0 &    // form value to be used
                    _CAN_TX_XTD_FRAME &    // with CANWrite
                    _CAN_TX_NO_RTR_FRAME;

  Can_Init_Flags = _CAN_CONFIG_SAMPLE_THRICE &     // form value to be used
                    _CAN_CONFIG_PHSEG2_PRG_ON &    // with CANInit
                    _CAN_CONFIG_XTD_MSG &
                    _CAN_CONFIG_DBL_BUFFER_ON &
                    _CAN_CONFIG_VALID_XTD_MSG;

  CANInitialize(1,3,3,3,1,Can_Init_Flags);      // Initialize CAN module
  CANSetOperationMode(_CAN_MODE_CONFIG,0xFF);   // set CONFIGURATION mode
  CANSetMask(_CAN_MASK_B1,-1,_CAN_CONFIG_XTD_MSG); // set all mask1 bits to
   ones
  CANSetMask(_CAN_MASK_B2,-1,_CAN_CONFIG_XTD_MSG); // set all mask2 bits to
   ones
  CANSetFilter(_CAN_FILTER_B2_F4,ID_2nd,_CAN_CONFIG_XTD_MSG);// set id of
   filter B2_F4 to 2nd node ID

  CANSetOperationMode(_CAN_MODE_NORMAL,0xFF);     // set NORMAL mode

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

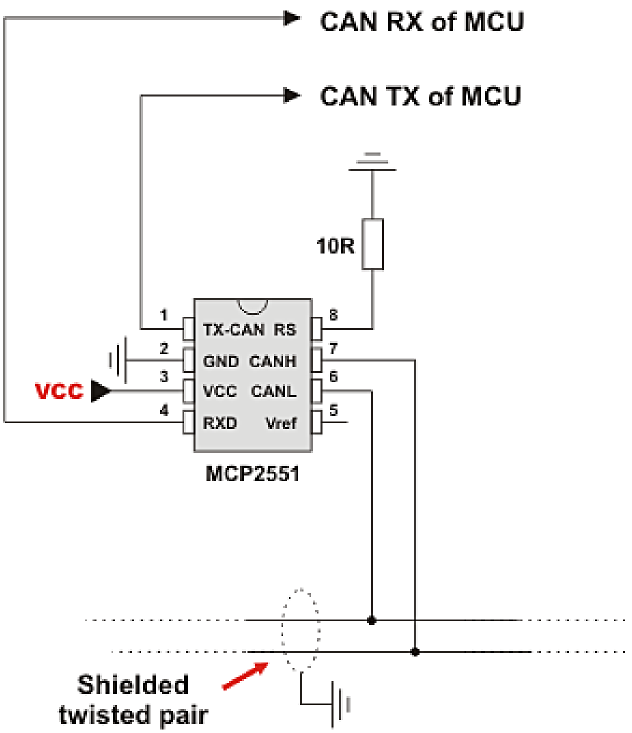
```
  CANWrite(ID_1st, RxTx_Data, 1, Can_Send_Flags); // send initial message

  while(1) {                                  // endless loop
   Msg_Rcvd = CANRead(&Rx_ID , RxTx_Data , &Rx_Data_Len, &Can_Rcv_Flags); //
receive message
    if ((Rx_ID == ID_2nd) && Msg_Rcvd) {  // if message received check id
      PORTC = RxTx_Data[0];   // id correct, output data at PORTC
      RxTx_Data[0]++;        // increment received data
Delay_ms(10);
 CANWrite(ID_1st, RxTx_Data, 1, Can_Send_Flags); // send incremented data back
    }
  }
}
```

Code for the second CAN node:

```
unsigned char Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags; // can
flags
unsigned char Rx_Data_Len;       // received data length in bytes
char RxTx_Data[8];               // can rx/tx data buffer
char Msg_Rcvd;                   // reception flag
const long ID_1st = 12111, ID_2nd = 3;  // node IDs
long Rx_ID;

void main() {

  PORTC = 0;                     // clear PORTC
  TRISC = 0;                     // set PORTC as output

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

  Can_Send_Flags = _CAN_TX_PRIORITY_0 &  // form value to be used
                    _CAN_TX_XTD_FRAME &  // with CANWrite
                    _CAN_TX_NO_RTR_FRAME;

  Can_Init_Flags = _CAN_CONFIG_SAMPLE_THRICE & // form value to be used
                    _CAN_CONFIG_PHSEG2_PRG_ON & // with CANInit
                    _CAN_CONFIG_XTD_MSG &
                    _CAN_CONFIG_DBL_BUFFER_ON &
                    _CAN_CONFIG_VALID_XTD_MSG &
                    _CAN_CONFIG_LINE_FILTER_OFF;

  CANInitialize(1,3,3,3,1,Can_Init_Flags); // initialize external CAN module
  CANSetOperationMode(_CAN_MODE_CONFIG,0xFF); // set CONFIGURATION mode
  CANSetMask(_CAN_MASK_B1,-1,_CAN_CONFIG_XTD_MSG); // set all mask1
bits to ones
```

```
   CANSetMask(_CAN_MASK_B2,-1,_CAN_CONFIG_XTD_MSG); // set all mask2
bits to ones
   CANSetFilter(_CAN_FILTER_B2_F3,ID_1st,_CAN_CONFIG_XTD_MSG);// set
id of filter B2_F3 to 1st node ID

  CANSetOperationMode(_CAN_MODE_NORMAL,0xFF); // set NORMAL mode

  while (1) {                                     // endless loop
        Msg_Rcvd = CANRead(&Rx_ID , RxTx_Data , &Rx_Data_Len,
&Can_Rcv_Flags); // receive message
  if ((Rx_ID == ID_1st) && Msg_Rcvd) { // if message received check id
       PORTC = RxTx_Data[ 0] ;  // id correct, output data at PORTC
       RxTx_Data[ 0] ++;        // increment received data
      CANWrite(ID_2nd, RxTx_Data, 1, Can_Send_Flags); // send incre-
mented data back
    }
  }
```

HW Connection



Example of interfacing CAN transceiver with MCU and bus

CANSPI LIBRARY

The SPI module is available with a number of the PIC compliant MCUs. The mikroC PRO for PIC 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 slower than “real” CAN.
- The library uses the SPI module for communication. User must initialize SPI module before using the SPI Graphic Lcd Library.
  For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine.
- CANSPI module refers to mikroElektronika's CANSPI Add-on board connected to SPI module of MCU.

External dependecies of CANSPI Library

The following variables must be defined in all projects using CANSPI Library:	Description:	Example:
`extern sfr sbit CanSpi_CS;`	Chip Select line.	`sbit CanSpi_CS at RC0_bit;`
`extern sfr sbit CanSpi_Rst;`	Reset line.	`sbit CanSpi_Rst at RC2_bit;`
`extern sfr sbit CanSpi_CS_Direction;`	Direction of the Chip Select pin.	`sbit CanSpi_CS_Direction at TRISC0_bit;`
`extern sfr sbit CanSpi_Rst_Direction;`	Direction of the Reset pin.	`sbit CanSpi_Rst_Direction at TRISC2_bit;`

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	`void CANSPISetOperationMode(char mode, char WAIT);`
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 operation 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	`char CANSPIGetOperationMode();`
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	see below

Example

```
// check whether the CANSPI module is in Normal mode and if it
is do something.
if (CANSPIGetOperationMode() == _CANSPI_MODE_NORMAL) {
  ...
}
```

CANSPIInitialize

Prototype	`void CANSPIInitialize( char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CANSPI_CONFIG_FLAGS);`
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 `CANSPI_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_Direction:` Direction of the Chip Select pin - `CanSpi_Rst_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
sbit CanSpi_CS  at  RC0_bit;
sbit CanSpi_CS_Direction  at  TRISC0_bit;
sbit CanSpi_Rst at  RC2_bit;
sbit CanSpi_Rst_Direction at  TRISC2_bit;
// End CANSPI module connections

// initialize the CANSPI module with the appropriate baud rate
and message acceptance flags along with the sampling rules
char CanSPi_Init_Flags;
  ...
  CanSPi_Init_Flags = _CANSPI_CONFIG_SAMPLE_THRICE &  // form
value to be used
                     _CANSPI_CONFIG_PHSEG2_PRG_ON &   // with
CANSPIInitialize
                     _CANSPI_CONFIG_XTD_MSG &
                     _CANSPI_CONFIG_DBL_BUFFER_ON &
                     _CANSPI_CONFIG_VALID_XTD_MSG;
  ...
  SPI1_Init();         // initialize SPI module
  CANSPIInitialize(1,3,3,3,1,CanSpi_Init_Flags);   // initialize
external CANSPI module
```

CANSPISetBaudRate

Prototype	`void CANSPISetBaudRate( char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CANSPI_CONFIG_FLAGS);`
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 `CANSPI_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
char canspi_config_flags;
...
CANSPISetOperationMode(CANSPI_MODE_CONFIG,0xFF); // set CONFIGU-
RATION mode (CANSPI module mast be in config mode for baud rate
settings)
canspi_config_flags = _CANSPI_CONFIG_SAMPLE_THRICE &
                      _CANSPI_CONFIG_PHSEG2_PRG_ON &
                      _CANSPI_CONFIG_STD_MSG        &
                      _CANSPI_CONFIG_DBL_BUFFER_ON &
                      _CANSPI_CONFIG_VALID_XTD_MSG &
                      _CANSPI_CONFIG_LINE_FILTER_OFF;
CANSPISetBaudRate(1, 1, 3, 3, 1, canspi_config_flags);
```

CANSPISetMask

Prototype	`void CANSPISetMask(char CANSPI_MASK, long val, char CANSPI_CONFIG_FLAGS);`
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` costants (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
& _CANSPI_CONFIG_XTD_MSG);
```

CANSPISetFilter

Prototype	`void CANSPISetFilter(char CANSPI_FILTER, long val, char CANSPI_CONFIG_FLAGS);`
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	`char CANSPIRead(long *id, char *rd_data, char *data_len, char *CANSPI_RX_MSG_FLAGS);`
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.
char msg_rcvd, rx_flags, data_len;
char data[8];
long msg_id;
...
CANSPISetOperationMode(CA_NSPI_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, data, data_len, rx_flags)) {
   ...
}
```

CANSPIWrite

Prototype	`char CANSPIWrite(long id, char *wr_data, char data_len, char CANSPI_TX_MSG_FLAGS);`
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
char tx_flags;
char data[ 8] ;
long msg_id;
...
CANSPISetOperationMode(_CANSPI_MODE_NORMAL,0xFF);
// set NORMAL mode (CANSPI must be in mode in which transmission
is possible)

tx_flags = _CANSPI_TX_PRIORITY_0 & _CANSPI_TX_XTD_FRAME;
// set message flags
CANSPIWrite(msg_id, 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 char
    _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 char
    _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          = 0xFF,   // 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 &
        _CANSPI_CONFIG_PHSEG2_PRG_ON &
        _CANSPI_CONFIG_STD_MSG       &
        _CANSPI_CONFIG_DBL_BUFFER_ON &
        _CANSPI_CONFIG_VALID_XTD_MSG &
        _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 char
    _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 &
              _CANSPI_TX_XTD_FRAME  &
              _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 char
    _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 hard
       ware double-buffered
```

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

```
if (MsgFlag & _CANSPI_RX_OVERFLOW != 0) {
  ...
  // Receiver overflow has occurred.
  // We have lost our previous message.
}
```

CANSPI_MASK

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

```
const char
     _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 char
     _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:

```
unsigned char Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags; // can
flags
unsigned char Rx_Data_Len;      // received data length in bytes
char RxTx_Data[8];              // can rx/tx data buffer
char Msg_Rcvd;                  // reception flag
const long ID_1st = 12111, ID_2nd = 3;   // node IDs
long Rx_ID;

// CANSPI module connections
sbit CanSpi_CS            at  RC0_bit;
sbit CanSpi_CS_Direction  at  TRISC0_bit;
sbit CanSpi_Rst           at  RC2_bit;
sbit CanSpi_Rst_Direction at  TRISC2_bit;
// End CANSPI module connections

void main() {

  ANSEL  = 0;              // Configure AN pins as digital I/O
  ANSELH = 0;

  PORTB = 0;                // clear PORTB
  TRISB = 0;                // set PORTB as output

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

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

  Can_Init_Flags = _CANSPI_CONFIG_SAMPLE_THRICE & // Form value to be used
                  _CANSPI_CONFIG_PHSEG2_PRG_ON & //  with CANSPIInit
                    _CANSPI_CONFIG_XTD_MSG &
                    _CANSPI_CONFIG_DBL_BUFFER_ON &
  _CANSPI_CONFIG_VALID_XTD_MSG;
```

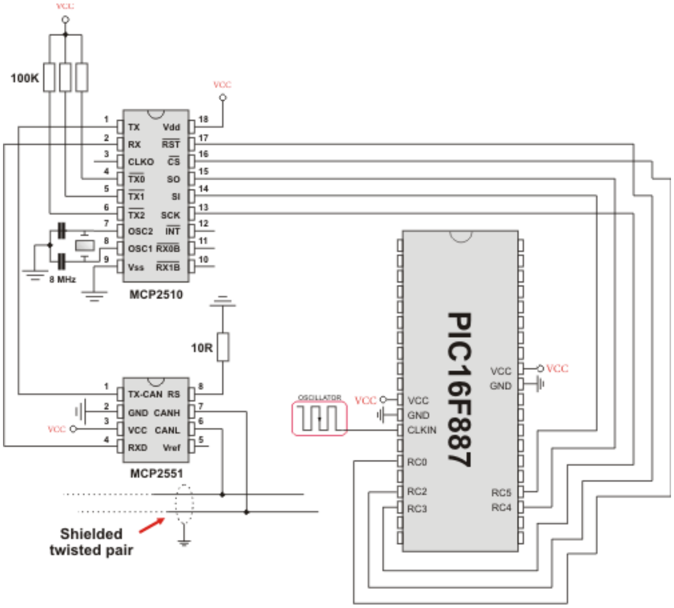
```
  SPI1_Init();                // initialize SPI1 module

   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,ID_2nd,_CANSPI_CONFIG_XTD_MSG);
// set id of filter B2_F4 to 2nd node ID

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

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

  CANSPIWrite(ID_1st, RxTx_Data, 1, Can_Send_Flags); // send initial
message

  while(1) {                 // endless loop
      Msg_Rcvd = CANSPIRead(&Rx_ID , RxTx_Data , &Rx_Data_Len,
&Can_Rcv_Flags);// receive message
    if ((Rx_ID == ID_2nd) && Msg_Rcvd) {
// if message received check id
      PORTB = RxTx_Data[0];
// id correct, output data at PORTC
      RxTx_Data[0]++;
// increment received data
      Delay_ms(10);
      CANSPIWrite(ID_1st, RxTx_Data, 1, Can_Send_Flags);
// send incremented data back
    }
  }
}
```

Code for the second CANSPI node:

```
unsigned char Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags; // can
flags
unsigned char Rx_Data_Len;        // received data length in bytes
char RxTx_Data[8];                // can rx/tx data buffer
char Msg_Rcvd;                    // reception flag
const long ID_1st = 12111, ID_2nd = 3;      // node IDs
long Rx_ID;

// CANSPI module connections
sbit CanSpi_CS            at  RC0_bit;
sbit CanSpi_CS_Direction  at  TRISC0_bit;
sbit CanSpi_Rst           at  RC2_bit;
sbit CanSpi_Rst_Direction at  TRISC2_bit;
// End CANSPI module connections

void main() {

  ANSEL  = 0;                     // Configure AN pins as digital I/O
  ANSELH = 0;

  PORTB = 0;                      // clear PORTB
  TRISB = 0;                      // set PORTB as output

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

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

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

  SPI1_Init();   // initialize SPI1 module

  CANSPIInitialize(1,3,3,3,1,Can_Init_Flags); // initialize external
CANSPI module
  CANSPISetOperationMode(_CANSPI_MODE_CONFIG,0xFF); // set CONFIGU-
RATION 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,ID_1st,_CANSPI_CONFIG_XTD_MSG);
// set id of filter B2_F3 to 1st node ID

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

  while (1) {                            // endless loop
       Msg_Rcvd = CANSPIRead(&Rx_ID , RxTx_Data , &Rx_Data_Len,
&Can_Rcv_Flags); // receive message
    if ((Rx_ID == ID_1st) && Msg_Rcvd) {  // if message received check id
       PORTB = RxTx_Data[ 0] ;  // id correct, output data at PORTC
       RxTx_Data[ 0] ++;        // increment received data
         CANSPIWrite(ID_2nd, RxTx_Data, 1, Can_Send_Flags); // send
incremented data back
    }
  }
}
```

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 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.

The following variables must be defined in all projects using Compact Flash Library:	Description:	Example:
`extern sfr char CF_Data_Port;`	Compact Flash Data Port.	`char CF_Data_Port at PORTD;`
`extern sfr sbit CF_RDY;`	Ready signal line.	`sbit CF_RDY at RB7_bit;`
`extern sfr sbit CF_WE;`	Write Enable signal line.	`sbit CF_WE at RB6_bit;`
`extern sfr sbit CF_OE;`	Output Enable signal line.	`sbit CF_OE at RB5_bit;`
`extern sfr sbit CF_CD1;`	Chip Detect signal line.	`sbit CF_CD1 at RB4_bit;`
`extern sfr sbit CF_CE1;`	Chip Enable signal line.	`sbit CF_CE1 at RB3_bit;`
`extern sfr sbit CF_A2;`	Address pin 2.	`sbit CF_A2 at RB2_bit;`
`extern sfr sbit CF_A1;`	Address pin 1.	`sbit CF_A1 at RB1_bit;`
`extern sfr sbit CF_A0;`	Address pin 0.	`sbit CF_A0 at RB0_bit;`
`extern sfr sbit CF_RDY_direction;`	Direction of the Ready pin.	`sbit CF_RDY_direction at TRISB7_bit;`
`extern sfr sbit CF_WE_direction;`	Direction of the Write Enable pin.	`sbit CF_WE_direction at TRISB6_bit;`
`extern sfr sbit CF_OE_direction;`	Direction of the Output Enable pin.	`sbit CF_OE_direction at TRISB5_bit;`
`extern sfr sbit CF_CD1_direction;`	Direction of the Chip Detect pin.	`sbit CF_CD1_direction at TRISB4_bit;`
`extern sfr sbit CF_CE1_direction;`	Direction of the Chip Enable pin.	`sbit CF_CE1_direction at TRISB3_bit;`
`extern sfr sbit CF_A2_direction;`	Direction of the Address 2 pin.	`sbit CF_A2_direction at TRISB2_bit;`
`extern sfr sbit CF_A1_direction;`	Direction of the Address 1 pin.	`sbit CF_A1_direction at TRISB1_bit;`
`extern sfr sbit CF_A0_direction;`	Direction of the Address 0 pin.	`sbit CF_A0_direction at TRISB0_bit;`

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

The following routine is for the internal use by compiler only:

- Cf_Issue_ID_Command

Cf_Init

Prototype	`void 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 - `CF_A1` : Address pin 1 - `CF_A0` : Address pin 0 - `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
char Cf_Data_Port at PORTD;

sbit CF_RDY at RB7_bit;
sbit CF_WE  at RB6_bit;
sbit CF_OE  at RB5_bit;
sbit CF_CD1 at RB4_bit;
sbit CF_CE1 at RB3_bit;
sbit CF_A2  at RB2_bit;
sbit CF_A1  at RB1_bit;
sbit CF_A0  at RB0_bit;

sbit CF_RDY_direction at TRISB7_bit;
sbit CF_WE_direction  at TRISB6_bit;
sbit CF_OE_direction  at TRISB5_bit;
sbit CF_CD1_direction at TRISB4_bit;
sbit CF_CE1_direction at TRISB3_bit;
sbit CF_A2_direction  at TRISB2_bit;
sbit CF_A1_direction  at TRISB1_bit;
sbit CF_A0_direction  at TRISB0_bit;
// end of compact flash pinout
...
Cf_Init();       // initialize CF
```

Cf_Detect

Prototype	`unsigned short Cf_Detect(void);`
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:
do
  asm nop;
while (!Cf_Detect());
```

Cf_Enable

Prototype	`void Cf_Enable(void);`
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	`void Cf_Disable(void);`
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	`void Cf_Read_Init(unsigned long address, unsigned short sector_count);`
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	`unsigned short Cf_Read_Byte(void);`
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:`
	`char data;`
	`...`
	`data = Cf_Read_Byte();`

Cf_Write_Init

Prototype	`void Cf_Write_Init(unsigned long address, unsigned short sectcnt);`
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	`void Cf_Write_Byte(unsigned short data_);`
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 contents 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	`char data = 0xAA;`
	`...`
	`Cf_Write_Byte(data);`

Cf_Read_Sector

Prototype	`void Cf_Read_Sector(unsigned long sector_number, unsigned short *buffer);`
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`
	`unsigned short data[512];`
	`...`
	`Cf_Read_Sector(22, data);`

Cf_Write_Sector

Prototype	`void Cf_Write_Sector(unsigned long sector_number, unsigned short *buffer);`
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`
	`unsigned short data[512];`
	`...`
	`Cf_Write_Sector(22, data);`

Cf_Fat_Init

Prototype	`unsigned short Cf_Fat_Init();`
Returns	- `0` - if CF card was detected and successfuly 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 necessary data needed by the library.
Requires	Nothing.

Example	`// Init the FAT library`
	`if (!Cf_Fat_Init()) {     // Init the FAT library`
	`...`
	`}`

Cf_Fat_QuickFormat

Prototype	`unsigned char Cf_Fat_QuickFormat(char *cf_fat_label);`
Returns	- `0` - if CF card was detected, successfuly 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 null string is passed, the volume will not be labeled. **Note**: This routine can be used instead or in conjunction with 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(&cf_fat_label)) {`
	`...`
	`}`

Cf_Fat_Assign

<table>
<tr><td>**Prototype**</td><td>`unsigned short Cf_Fat_Assign(char *filename, char file_cre_attr);`</td></tr>
<tr><td>**Returns**</td><td>`- 0` if file does not exist and no new file is created.
`- 1` if file already exists or file does not exist but a new file is created.</td></tr>
<tr><td>**Description**</td><td>Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied over the assigned file.

Parameters:
- `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 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_cre_attr`: 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>
Note: Long File Names (LFN) are not supported.</td></tr>
<tr><td>**Requires**</td><td>CF card and CF library must be initialized for file operations. See Cf_Fat_Init.</td></tr>
<tr><td>**Example**</td><td>`// create file with archive attributes if it does not already exist`
`Cf_Fat_Assign("MIKRO007.TXT",0xA0);`</td></tr>
</table>

Cf_Fat_Reset

Prototype	`void Cf_Fat_Reset(unsigned long *size);`
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	`unsigned long size;`
	`...`
	`Cf_Fat_Reset(size);`

Cf_Fat_Read

Prototype	`void Cf_Fat_Read(unsigned short *bdata);`
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	`char character;`
	`...`
	`Cf_Fat_Read(&character);`

Cf_Fat_Rewrite

Prototype	`void 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	`void 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	`void 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	`void Cf_Fat_Write(char *fdata, unsigned data_len);`
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	`char file_contents[42];`
	`...`
	`Cf_Fat_Write(file_contents, 42); // write data to the assigned file`

Cf_Fat_Set_File_Date

Prototype	`void Cf_Fat_Set_File_Date(unsigned int year, unsigned short month, unsigned short day, unsigned short hours, unsigned short mins, unsigned short seconds);`
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_Set_File_Date

Prototype	`void Cf_Fat_Get_File_Date(unsigned int *year, unsigned short *month, unsigned short *day, unsigned short *hours, unsigned short *mins);`
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	`unsigned year;`
	`char month, day, hours, mins;`
	`...`
	`Cf_Fat_Get_File_Date(&year, &month, &day, &hours, &mins);`

Cf_Fat_Set_File_Size

Prototype	`unsigned long Cf_Fat_Get_File_Size();`
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	`unsigned long my_file_size;`
	`...`
	`my_file_size = Cf_Fat_Get_File_Size();`

Cf_Fat_Get_Swap_File

<table>
<tr><td>Prototype</td><td><code>unsigned long Cf_Fat_Get_Swap_File(unsigned long sectors_cnt, char *filename, char file_attr);</code></td></tr>
<tr><td>Returns</td><td>- 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.
- <code>0</code> otherwise</td></tr>
<tr><td>Description</td><td>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. Swap file can be considered as a "window" on the media where the user can freely write/read data. It's main purpose in the mikroC'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:
- <code>sectors_cnt</code>: number of consecutive sectors that user wants the swap file to have.
- <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 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 chsaracter 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_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>Not used</td></tr>
</table>
Note: Long File Names (LFN) are not supported.</td></tr>
</table>

Requires	CF card and CF library must be initialized for file operations. See Cf_Fat_Init.
Example	See code below.

```
//--------------- 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 sector over UART
unsigned long size;
...
size = Cf_Fat_Get_Swap_File(1000, "mikroE.txt", 0x20);
if (size) {
  UART_Write(0xAA);
  UART_Write(Lo(size));
  UART_Write(Hi(size));
  UART_Write(Higher(size));
  UART_Write(Highest(size));
  UART_Write(0xAA);
}
```

Library Example

The following example demonstrates various aspects of the Cf_Fat16 library: Creation of new file and writing down to it; Opening existing file and re-writing it (writing from start-of-file); Opening existing file and appending data to it (writing from end-of-file); Opening a file and reading data from it (sending it to USART terminal); Creating and modifying several files at once;

```
// set compact flash pinout
char Cf_Data_Port at PORTD;

sbit CF_RDY at RB7_bit;
sbit CF_WE  at RB6_bit;
sbit CF_OE  at RB5_bit;
sbit CF_CD1 at RB4_bit;
sbit CF_CE1 at RB3_bit;
sbit CF_A2  at RB2_bit;
sbit CF_A1  at RB1_bit;
sbit CF_A0  at RB0_bit;

sbit CF_RDY_direction at TRISB7_bit;
sbit CF_WE_direction  at TRISB6_bit;
sbit CF_OE_direction  at TRISB5_bit;
sbit CF_CD1_direction at TRISB4_bit;
sbit CF_CE1_direction at TRISB3_bit;
sbit CF_A2_direction  at TRISB2_bit;
sbit CF_A1_direction  at TRISB1_bit;
sbit CF_A0_direction  at TRISB0_bit;
// end of cf pinout

const LINE_LEN = 39;
char err_txt[20]          = "FAT16 not found";
char file_contents[LINE_LEN] = "XX CF FAT16 library by Anton Rieckertn";
```

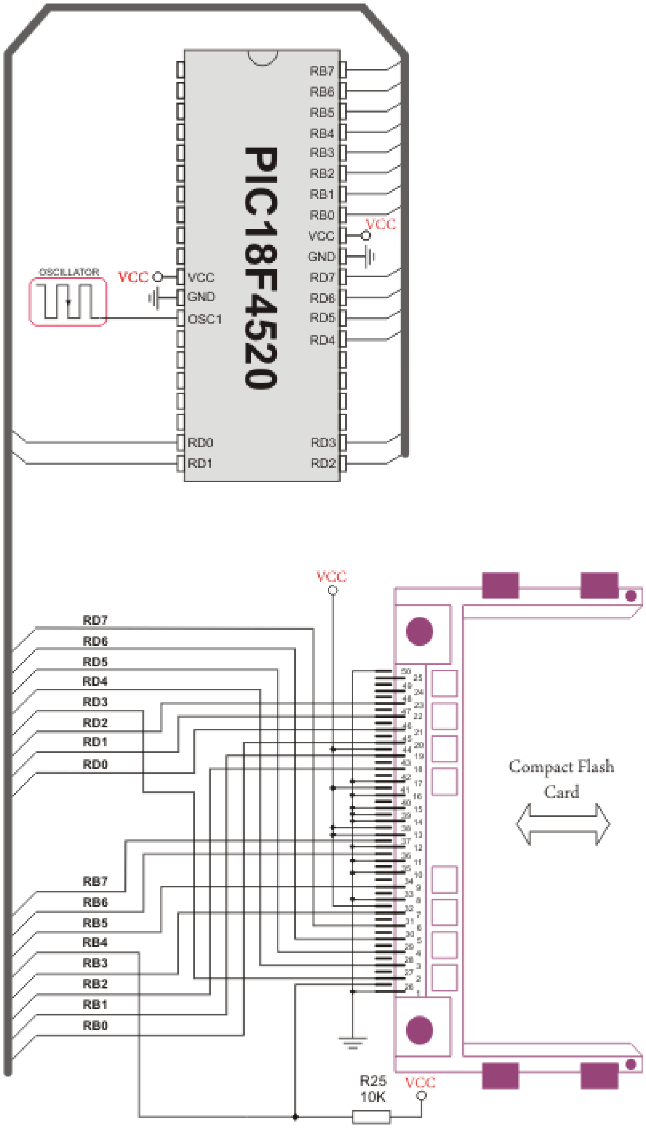
```
char             filename[ 14] = "MIKRO00x.TXT";  // File names
unsigned short loop, loop2;
unsigned long  i, size;
char             Buffer[ 512] ;

// UART1 write text and new line (carriage return + line feed)
void UART1_Write_Line(char *uart_text) {
  UART1_Write_Text(uart_text);
  UART1_Write(13);
  UART1_Write(10);
}

// Creates new file and writes some data to it
void M_Create_New_File() {
  filename[ 7] = 'A';
  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; loop <= 99; loop++) {
    UART1_Write('.');
    file_contents[ 0] = loop / 10 + 48;
    file_contents[ 1] = loop % 10 + 48;
     Cf_Fat_Write(file_contents, LINE_LEN-1); // write data to the
assigned file
  }
}

// Creates many new files and writes data to them
void M_Create_Multiple_Files() {
  for(loop2 = 'B'; loop2 <= 'Z'; loop2++) {
    UART1_Write(loop2);             // signal the progress
    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; loop <= 44; loop++) {
      file_contents[ 0] = loop / 10 + 48;
      file_contents[ 1] = loop % 10 + 48;
      Cf_Fat_Write(file_contents, LINE_LEN-1);  // write data to the
assigned file
    }
  }
}

// Opens an existing file and rewrites it
void M_Open_File_Rewrite() {
  filename[ 7] = 'C';
  Cf_Fat_Assign(&filename, 0);
  Cf_Fat_Rewrite();
  for(loop = 1; loop <= 55; loop++) {
```

```
     file_contents[0] = loop / 10 + 65;
     file_contents[1] = loop % 10 + 65;
     Cf_Fat_Write(file_contents, LINE_LEN-1);   // write data to the
assigned file
  }
}

// Opens an existing file and appends data to it
//               (and alters the date/time stamp)
void M_Open_File_Append() {
   filename[7] = 'B';
   Cf_Fat_Assign(&filename, 0);
   Cf_Fat_Set_File_Date(2005,6,21,10,35,0);
   Cf_Fat_Append();            // Prepare file for append
   Cf_Fat_Write(" for mikroElektronika 2005n", 27); // Write data to
assigned file
}

// Opens an existing file, reads data from it and puts it to UART
void M_Open_File_Read() {
  char character;

  filename[7] = 'B';
  Cf_Fat_Assign(&filename, 0);
  Cf_Fat_Reset(&size);// To read file, procedure returns size of file
  for (i = 1; i <= size; i++) {
    Cf_Fat_Read(&character);
    UART1_Write(character);     // Write data to UART
  }
}

// Deletes a file. If file doesn't exist, it will first be created
// and then deleted.
void M_Delete_File() {
  filename[7] = 'F';
  Cf_Fat_Assign(filename, 0);
  Cf_Fat_Delete();
}

// Tests whether file exists, and if so sends its creation date
// and file size via UART
void M_Test_File_Exist() {
  unsigned long  fsize;
  unsigned int   year;
  unsigned short month, day, hour, minute;
  unsigned char  outstr[12];
  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)) {
    //--- file has been found - get its date
    Cf_Fat_Get_File_Date(&year, &month, &day, &hour, &minute);
    WordToStr(year, outstr);
    UART1_Write_Text(outstr);
    ByteToStr(month, outstr);
    UART1_Write_Text(outstr);
    WordToStr(day, outstr);
    UART1_Write_Text(outstr);
    WordToStr(hour, outstr);
    UART1_Write_Text(outstr);
    WordToStr(minute, outstr);
    UART1_Write_Text(outstr);
    //--- get file size
    fsize = Cf_Fat_Get_File_Size();
    LongToStr((signed long)fsize, outstr);
    UART1_Write_Line(outstr);
  }
  else {
    //--- file was not found - signal it
    UART1_Write(0x55);
    Delay_ms(1000);
    UART1_Write(0x55);
  }
}
// Tries to create a swap file, whose size will be at least 100
// sectors (see Help for details)
void M_Create_Swap_File() {
  unsigned int i;

  for(i=0; i<512; i++)
    Buffer[ i]  = i;

  size = Cf_Fat_Get_Swap_File(5000, "mikroE.txt", 0x20); // see help
on this function for details

  if (size) {
    LongToStr((signed long)size, err_txt);
    UART1_Write_Line(err_txt);

    for(i=0; i<5000; i++) {
      Cf_Write_Sector(size++, Buffer);
      UART1_Write('.');
    }
  }
}
// Main. Uncomment the function(s) to test the desired operation(s)
void main() {
  #define COMPLETE_EXAMPLE       // comment this line to make sim-
pler/smaller example
```

```
  ADCON1  |= 0x0F;               // Configure AN pins as digital
  CMCON   |= 7;                  // Turn off comparators

  // Initialize UART1 module
  UART1_Init(19200);
  Delay_ms(10);

  UART1_Write_Line("PIC-Started"); // PIC present report

  // use fat16 quick format instead of init routine if a formatting
is needed
  if (Cf_Fat_Init() == 0) {
    Delay_ms(2000); // wait for a while until the card is stabilized
                    //   period depends on used CF card
    //--- Test start
    UART1_Write_Line("Test Start.");
    //--- Test routines. Uncomment them one-by-one to test certain
features
    M_Create_New_File();
    #ifdef COMPLETE_EXAMPLE
      M_Create_Multiple_Files();
      M_Open_File_Rewrite();
      M_Open_File_Append();
      M_Open_File_Read();
      M_Delete_File();
      M_Test_File_Exist();
      M_Create_Swap_File();
    #endif
    UART1_Write_Line("Test End.");

  }
  else {
    UART1_Write_Line(err_txt); // Note: Cf_Fat_Init tries to initial-
ize a card more than once.
                               // If card is not present, initializa-
tion may last longer (depending on clock speed)
  }

}
```

HW Connection



Pin diagram of CF memory card

EEPROM LIBRARY

EEPROM data memory is available with a number of PIC MCUs. *mikroC PRO for PIC* includes library for comfortable work with EEPROM.

Library Routines

- Eeprom_Read
- Eeprom_Write

EEPROM_Read

Prototype	`unsigned short EEPROM_Read(unsigned int address);`
Returns	Returns byte from specified address.
Description	Reads `data` from specified `address`. Parameter address is of integer type, which means it supports MCUs with more than 256 bytes of EEPROM.
Requires	Requires EEPROM module. Ensure minimum 20ms delay between successive use of routines `EEPROM_Write` and `EEPROM_Read`. Although PIC will write the correct value, `EEPROM_Read` might return an undefined result.

Example	`unsigned short take;`
	`...`
	`take = EEPROM_Read(0x3F);`

EEPROM_Write

Prototype	`void EEPROM_Write(unsigned int address, unsigned short data);`
Returns	Nothing.
Description	Writes `data` to specified `address`. Parameter address is of integer type, which means it supports MCUs with more than 256 bytes of EEPROM. Be aware that all interrupts will be disabled during execution of `EEPROM_Write` routine (GIE bit of INTCON register will be cleared). Routine will restore previous state of this bit on exit.
Requires	Requires EEPROM module. Ensure minimum 20ms delay between successive use of routines `EEPROM_Write` and `EEPROM_Read`. Although PIC will write the correct value, `EEPROM_Read` might return an undefined result.
Example	`EEPROM_Write(0x32, 19);`

Library Example

The example demonstrates use of EEPROM Library.

```
char ii;                      // loop variable

void main(){
  ANSEL =  0;                 // Configure AN pins as digital I/O
  ANSELH = 0;

  PORTB = 0;
  PORTC = 0;
  PORTD = 0;

  TRISB = 0;
  TRISC = 0;
  TRISD = 0;

  for(ii = 0; ii < 32; ii++)        // Fill data buffer
    EEPROM_Write(0x80+ii, ii);      // Write data to address 0x80+ii

  EEPROM_Write(0x02,0xAA);          // Write some data at address 2
  EEPROM_Write(0x50,0x55);          // Write some data at address 0150

  Delay_ms(1000);                   // Blink PORTB and PORTC diodes
  PORTB = 0xFF;                     // to indicate reading start
  PORTC = 0xFF;
  Delay_ms(1000);
  PORTB = 0x00;
  PORTC = 0x00;
  Delay_ms(1000);

  PORTB = EEPROM_Read(0x02);        // Read data from address 2 and
display it on PORTB
  PORTC = EEPROM_Read(0x50);        // Read data from address 0x50 and
display it on PORTC

  Delay_ms(1000);

  for(ii = 0; ii < 32; ii++) {  // Read 32 bytes block from address 0x80
    PORTD = EEPROM_Read(0x80+ii); // and display data on PORTD
    Delay_ms(250);
    }
}
```

ETHERNET PIC18FXXJ60 LIBRARY

`PIC18FxxJ60` family of microcontrollers feature an embedded Ethernet controller module. This is a complete connectivity solution, including full implementations of both Media Access Control (MAC) and Physical Layer transceiver (PHY) modules. Two pulse transformers and a few passive components are all that are required to connect the microcontroller directly to an Ethernet network.

The Ethernet module meets all of the IEEE 802.3 specifications for 10-BaseT connectivity to a twisted-pair network. 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. Provisions are also made for two LED outputs to indicate link and network activity

This library provides the posibility to easily utilize ethernet feature of the above mentioned MCUs.

Ethernet PIC18FxxJ60 library supports:

- IPv4 protocol.
- ARP requests.
- ICMP echo requests.
- UDP requests.
- TCP requests (no stack, no packet reconstruction).
- ARP client with cache.
- DNS client.
- UDP client.
- DHCP client.
- packet fragmentation is NOT supported.

Note: Global library variable `Ethernet_userTimerSec` is used to keep track of time for all client implementations (ARP, DNS, UDP and DHCP). It is user responsibility to increment this variable each second in it's code if any of the clients is used.

Note: For advanced users there are header files (`"eth_j60LibDef.h"` and `"eth_j60LibPrivate.h"`) in Uses\P18 folder of the compiler with description of all routines and global variables, relevant to the user, implemented in the Ethernet PIC18FxxJ60 Library.

Library Routines

- Ethernet_Init
- Ethernet_Enable
- Ethernet_Disable
- Ethernet_doPacket
- Ethernet_putByte
- Ethernet_putBytes
- Ethernet_putString
- Ethernet_putConstString
- Ethernet_putConstBytes
- Ethernet_getByte
- Ethernet_getBytes
- Ethernet_UserTCP
- Ethernet_UserUDP
- Ethernet_getIpAddress
- Ethernet_getGwIpAddress
- Ethernet_getDnsIpAddress
- Ethernet_getIpMask
- Ethernet_confNetwork
- Ethernet_arpResolve
- Ethernet_sendUDP
- Ethernet_dnsResolve
- Ethernet_initDHCP
- Ethernet_doDHCPLeaseTime
- Ethernet_renewDHCP

Ethernet_Init

Prototype	`void Ethernet_Init(unsigned char *mac, unsigned char *ip, unsigned char fullDuplex);`
Returns	Nothing.
Description	This is MAC module routine. It initializes Ethernet controller. This function is internaly splited into 2 parts to help linker when coming short of memory. Ethernet 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. - half duplex loopback disabled. - LED configuration: default (LEDA-link status, LEDB-link activity). 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). **Note**: If a DHCP server is to be used, IP address should be set to `0.0.0.0`.
Requires	Nothing.
Example	

```
#define Ethernet_HALFDUPLEX      0
#define Ethernet_FULLDUPLEX      1

unsigned char myMacAddr[6]  = {0x00, 0x14, 0xA5, 0x76, 0x19, 0x3f} ; // my MAC address
unsigned char myIpAddr = {192, 168,   1, 60 };   // my IP addr

Ethernet_Init(myMacAddr, myIpAddr, Ethernet_FULLDUPLEX);
```

Ethernet_Enable

<table>
<tr><td>Prototype</td><td><code>void Ethernet_Enable(unsigned char enFlt);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</td><td>This is MAC module routine. This routine enables appropriate network traffic on the MCU's internal Ethernet 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.

Parameters:
<code>- enFlt</code>: 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><code>_Ethernet_BROADCAST</code></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><code>_Ethernet_MULTICAST</code></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><code>_Ethernet_CRC</code></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><code>_Ethernet_UNICAST</code></td></tr>
</table>

Note: Advance filtering available in the MCU's internal Ethernet module such as <code>Pattern Match</code>, <code>Magic Packet</code> and <code>Hash Table</code> 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.
Note: 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 MCU's internal Ethernet module. The MCU's internal Ethernet module should be properly cofigured by the means of Ethernet_Init routine.</td></tr>
</table>

Requires	Ethernet module has to be initialized. See Ethernet_Init.
Example	`Ethernet_Enable(_Ethernet_CRC | _Ethernet_UNICAST); // enable CRC checking and Unicast traffic`

Ethernet_Disable

<table>
<tr><td>Prototype</td><td><code>void Ethernet_Enable(unsigned char enFlt);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</td><td>
<p>This is MAC module routine. This routine disables appropriate network traffic on the MCU's internal Ethernet 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:
<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>0</td><td>0x01</td><td>MAC Broadcast traffic/receive filter flag. When set, MAC broadcast traffic will be disabled.</td><td><code>_Ethernet_BROADCAST</code></td></tr>
<tr><td>1</td><td>0x02</td><td>MAC Multicast traffic/receive filter flag. When set, MAC multicast traffic will be disabled.</td><td><code>_Ethernet_MULTICAST</code></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, CRC check will be disabled and packets with invalid CRC field will be accepted.</td><td><code>_Ethernet_CRC</code></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 disabled.</td><td><code>_Ethernet_UNICAST</code></td></tr>
</table>
<p>Note: Advance filtering available in the MCU's internal Ethernet module such as <code>Pattern Match</code>, <code>Magic Packet</code> and <code>Hash Table</code> can not be disabled by this routine.
Note: 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 MCU's internal Ethernet module. The MCU's internal Ethernet module should be properly cofigured by the means of Ethernet_Init routine.</p>
</td></tr>
</table>

Requires	Ethernet module has to be initialized. See Ethernet_Init..
Example	`Ethernet_Disable(_Ethernet_CRC | _Ethernet_UNICAST); // disable CRC checking and Unicast traffic`

Ethernet_doPacket

Prototype	`unsigned char Ethernet_doPacket();`
Returns	- `0` - upon successful packet processing (zero packets received or received packet processed successfuly). - `1` - upon reception error or receive buffer corruption. Ethernet 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 Ethernet_UserTCP function is called for further processing. - upon UDP request the Ethernet_UserUDP function is called for further processing. **Note**: `Ethernet_doPacket` must be called as often as possible in user's code.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example

```
if (Ethernet_doPacket() == 0) {  // process received packets
   ...
}
```

Ethernet_putByte

Prototype	`void Ethernet_putByte(unsigned char v);`
Returns	Nothing.
Description	This is MAC module routine. It stores one byte to address pointed by the current Ethernet controller's write pointer (`EWRPT`). Parameters: - `v`: value to store
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`char data;`
	`...`
	`Ethernet_putByte(data); // put an byte into Ethernet controller's buffer`

Ethernet_putBytes

Prototype	`void Ethernet_putBytes(unsigned char *ptr, unsigned char n);`
Returns	Nothing.
Description	This is MAC module routine. It stores requested number of bytes into Ethernet controller's RAM starting from current Ethernet controller's write pointer (`EWRPT`) location. Parameters: - `ptr`: RAM buffer containing bytes to be written into Ethernet controller's RAM. - `n`: number of bytes to be written.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`char *buffer =  "mikroElektronika";`
	`...`
	`Ethernet_putBytes(buffer, 16); // put an RAM array into Ethernet controller's buffer`

Ethernet_putConstBytes

Prototype	`void Ethernet_putConstBytes(const unsigned char *ptr, unsigned char n);`
Returns	Nothing.
Description	This is MAC module routine. It stores requested number of const bytes into Ethernet controller's RAM starting from current Ethernet controller's write pointer (`EWRPT`) location. Parameters: - `ptr`: const buffer containing bytes to be written into Ethernet controller's RAM. - `n`: number of bytes to be written.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`const char *buffer =  "mikroElektronika";`
	`...`
	`Ethernet_putConstBytes(buffer, 16); // put a const array into Ethernet controller's buffer`

Ethernet_putString

Prototype	`unsigned int Ethernet_putString(unsigned char *ptr);`
Returns	Number of bytes written into Ethernet controller's RAM.
Description	This is MAC module routine. It stores whole string (excluding null termination) into Ethernet controller's RAM starting from current Ethernet controller's write pointer (`EWRPT`) location. Parameters: - `ptr`: string to be written into Ethernet controller's RAM.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`char *buffer =  "mikroElektronika";`
	`...`
	`Ethernet_putString(buffer); // put a RAM string into Ethernet controller's buffer`

Ethernet_putConstString

Prototype	`unsigned int Ethernet_putConstString(const unsigned char *ptr);`
Returns	Number of bytes written into Ethernet controller's RAM.
Description	This is MAC module routine. It stores whole const string (excluding null termination) into Ethernet controller's RAM starting from current Ethernet controller's write pointer (`EWRPT`) location. Parameters: - `ptr`: const string to be written into Ethernet controller's RAM.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`const char *buffer =  "mikroElektronika";`
	`...`
	`Ethernet_putConstString(buffer); // put a const string into Ethernet controller's buffer`

Ethernet_getByte

Prototype	`unsigned char Ethernet_getByte();`
Returns	Byte read from Ethernet controller's RAM.
Description	This is MAC module routine. It fetches a byte from address pointed to by current Ethernet controller's read pointer (`ERDPT`).
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`char buffer;`
	`...`
	`buffer = Ethernet_getByte(); // read a byte from Ethernet controller's buffer`

Ethernet_getBytes

Prototype	`void Ethernet_getBytes(unsigned char *ptr, unsigned int addr, unsigned char n);`
Returns	Nothing.
Description	This is MAC module routine. It fetches equested number of bytes from Ethernet controller's RAM starting from given address. If value of 0xFFFF is passed as the address parameter, the reading will start from current Ethernet controller's read pointer (ERDPT) location. Parameters: - `ptr`: buffer for storing bytes read from Ethernet controller's RAM. - `addr`: Ethernet controller's RAM start address. Valid values: 0..8192. - `n`: number of bytes to be read.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`char buffer[16];`
	`...`
	`Ethernet_getBytes(buffer, 0x100, 16); // read 16 bytes, starting from address 0x100`

Ethernet_UserTCP

Prototype	`unsigned int Ethernet_UserTCP(unsigned char *remoteHost, unsigned int remotePort, unsigned int localPort, unsigned int reqLength);`
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 Ethernet_get routines. The user puts data in the transmit buffer by using some of the 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 Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.

Ethernet_UserUDP

Prototype	`unsigned int Ethernet_UserUDP(unsigned char *remoteHost, unsigned int remotePort, unsigned int destPort, unsigned int reqLength);`
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 Ethernet_get routines. The user puts data in the transmit buffer by using some of the 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 Ethernet_Init.
Example	This function is internally called by the library and should not be called by the user's code.

Ethernet_getIpAddress

Prototype	`unsigned char * Ethernet_getIpAddress();`
Returns	Ponter to the global variable holding IP address.
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP address. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own IP address buffer. These locations should not be altered by the user in any case!
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char ipAddr[4];  // user IP address buffer`
	`...`
	`memcpy(ipAddr, Ethernet_getIpAddress(), 4); // fetch IP address`

Ethernet_getGwIpAddress

Prototype	`unsigned char * Ethernet_getGwIpAddress();`
Returns	Ponter to the global variable holding gateway IP address.
Description	This routine should be used when DHCP server is present on the network to fetch assigned gateway IP address. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own gateway IP address buffer. These locations should not be altered by the user in any case!
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char gwIpAddr[4];  // user gateway IP address buffer`
	`...`
	`memcpy(gwIpAddr, Ethernet_getGwIpAddress(), 4); // fetch gateway IP address`

Ethernet_getDnsIpAddress();

Prototype	`unsigned char * Ethernet_getDnsIpAddress`
Returns	Ponter to the global variable holding DNS IP address.
Description	This routine should be used when DHCP server is present on the network to fetch assigned DNS IP address. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own DNS IP address buffer. These locations should not be altered by the user in any case!
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char dnsIpAddr[4];  // user DNS IP address buffer`
	`...`
	`memcpy(dnsIpAddr, Ethernet_getDnsIpAddress(), 4); // fetch DNS server address`

Ethernet_getIpMask

Prototype	`unsigned char * Ethernet_getIpMask()`
Returns	Ponter to the global variable holding IP subnet mask.
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP subnet mask. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own IP subnet mask buffer. These locations should not be altered by the user in any case!
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char IpMask[4];   // user IP subnet mask buffer`
	`...`
	`memcpy(IpMask, Ethernet_getIpMask(), 4); // fetch IP subnet mask`

Ethernet_confNetwork

Prototype	`void Ethernet_confNetwork(char *ipMask, char *gwIpAddr, char *dnsIpAddr);`
Returns	Nothing.
Description	Configures network parameters (IP subnet mask, gateway IP address, DNS IP address) when DHCP is not used. Parameters: - `ipMask`: IP subnet mask. - `gwIpAddr` gateway IP address. - `dnsIpAddr`: DNS IP address. **Note**: The above mentioned network parameters should be set by this routine only if DHCP module is not used. Otherwise DHCP will override these settings.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char ipMask[4]    = {255, 255, 255,  0 };   // network mask (for example : 255.255.255.0)`
	`unsigned char gwIpAddr[4]  = {192, 168,   1,  1 };   // gateway (router) IP address`
	`unsigned char dnsIpAddr[4] = {192, 168,   1,  1 };   // DNS server IP address`
	`...`
	`Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr); // set network configuration parameters`

Ethernet_arpResolve

Prototype	`unsigned char *Ethernet_arpResolve(unsigned char *ip, unsigned char tmax);`
Returns	- MAC address behind the IP address - the requested IP address was resolved. - `0` - otherwise.
Description	This is ARP module routine. It sends an ARP request for given IP address and waits for ARP reply. If the requested IP address was resolved, an ARP cash entry is used for storing the configuration. ARP cash can store up to 3 entries. For ARP cash structure refer to `"eth_j60LibDef.h"` header file in the compiler's Uses/P18 folder. Parameters: - `ip`: IP address to be resolved. - `tmax`: time in seconds to wait for an reply. **Note**: The Ethernet services are not stopped while this routine waits for ARP reply. The incoming packets will be processed normaly during this time.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char IpAddr[4]  = {192, 168,   1,  1 };  // IP address`
	`...`
	`Ethernet_arpResolve(IpAddr, 5); // get MAC address behind the above IP address, wait 5 secs for the response`

Ethernet_sendUDP

Prototype	`unsigned char Ethernet_sendUDP(unsigned char *destIP, unsigned int sourcePort, unsigned int destPort, unsigned char *pkt, unsigned int pktLen);`
Returns	- `1` - UDP packet was sent successfully. - `0` - otherwise.
Description	This is UDP module routine. It sends an UDP packet on the network. Parameters: - `destIP`: remote host IP address. - `sourcePort`: local UDP source port number. - `destPort`: destination UDP port number. - `pkt`: packet to transmit. - `pktLen`: length in bytes of packet to transmit.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char IpAddr[4]  = {192, 168,   1,  1 };  // remote IP address`
	`...`
	`Ethernet_sendUDP(IpAddr, 10001, 10001, "Hello", 5); // send Hello message to the above IP address, from UDP port 10001 to UDP port 10001`

Ethernet_dnsResolve

Prototype	`unsigned char *Ethernet_dnsResolve(unsigned char *host, unsigned char tmax);`
Returns	- pointer to the location holding the IP address - the requested host name was resolved. `- 0` - otherwise.
Description	This is DNS module routine. It sends an DNS request for given host name and waits for DNS reply. If the requested host name was resolved, it's IP address is stored in library global variable and a pointer containing this address is returned by the routine. UDP port 53 is used as DNS port. Parameters: `- host`: host name to be resolved. `- tmax`: time in seconds to wait for an reply. **Note**: The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own resolved host IP address buffer. These locations should not be altered by the user in any case!
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`unsigned char * remoteHostIpAddr[ 4] ; // user host IP address buffer`
	`...`
	`// SNTP server:`
	`// Zurich, Switzerland: Integrated Systems Lab, Swiss Fed. Inst. of Technology`
	`// 129.132.2.21: swisstime.ethz.ch`
	`// Service Area: Switzerland and Europe`
	`memcpy(remoteHostIpAddr, Ethernet_dnsResolve("swisstime.ethz.ch", 5), 4);`

Ethernet_initDHCL

Prototype	`unsigned char Ethernet_initDHCP(unsigned char tmax);`
Returns	- `1` - network parameters were obtained successfuly. - `0` - otherwise.
Description	This is DHCP module routine. It sends an DHCP request for network parameters (IP, gateway, DNS addresses and IP subnet mask) and waits for DHCP reply. If the requested parameters were obtained successfuly, their values are stored into the library global variables. These parameters can be fetched by using appropriate library IP get routines: - Ethernet_getIpAddress - fetch IP address. - Ethernet_getGwIpAddress - fetch gateway IP address. - Ethernet_getDnsIpAddress - fetch DNS IP address. - Ethernet_getIpMask - fetch IP subnet mask. UDP port `68` is used as DHCP client port and UDP port `67` is used as DHCP server port. Parameters: - `tmax`: time in seconds to wait for an reply. **Note**: The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. **Note**: When DHCP module is used, global library variable `Ethernet_userTimerSec` is used to keep track of time. It is user responsibility to increment this variable each second in it's code.
Requires	Ethernet module has to be initialized. See Ethernet_Init.

Example	`...`
	`Ethernet_initDHCP(5); // get network configuration from DHCP server, wait 5 sec for the response`
	`...`

Ethernet_doDHCPLeaseTime

Prototype	`unsigned char Ethernet_doDHCPLeaseTime();`
Returns	- `0` - lease time has not expired yet. - `1` - lease time has expired, it's time to renew it.
Description	This is DHCP module routine. It takes care of IP address lease time by decrementing the global lease time library counter. When this time expires, it's time to contact DHCP server and renew the lease.
Requires	Ethernet module has to be initialized. See Ethernet_Init.
Example	

```
while(1) {
   ...
   if(Ethernet_doDHCPLeaseTime())
      ... // it's time to renew the IP address lease
}
```

Ethernet_renewDHCP

Prototype	`unsigned char Ethernet_renewDHCP(unsigned char tmax);`
Returns	- `0` - upon success (lease time was renewed). - `1` - otherwise (renewal request timed out).
Description	This is DHCP module routine. It sends IP address lease time renewal request to DHCP server. Parameters: - `tmax`: time in seconds to wait for an reply.
Requires	Ethernet module has to be initialized. See Ethernet_Init.
Example	

```
while(1) {
   ...
   if(Ethernet_doDHCPLeaseTime())
      Ethernet_renewDHCP(5); // it's time to renew the IP address lease,
with 5 secs for a reply
   ...
}
```

Library Example

This code shows how to use the PIC18FxxJ60 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 RD0 to RD7 bit and return HTML main page
  all other requests return also HTML main page.

```
#define _Ethernet_HALFDUPLEX      0
#define Ethernet_FULLDUPLEX      1

/*************************************************************
 * ROM constant strings
 */
const unsigned char httpHeader[] = "HTTP/1.1 200 OKnContent-type: "
;  // HTTP header
const unsigned char httpMimeTypeHTML[] = "text/htmlnn" ;
// HTML MIME type
const unsigned char httpMimeTypeScript[] = "text/plainnn" ;
// TEXT MIME type
unsigned char httpMethod[] = "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 char    *indexPage = // Change the IP address of the page to
be refreshed
"<meta http-equiv="refresh" content="3;url=http://192.168.20.60">
<HTML><HEAD></HEAD><BODY>
<h1>PIC18FxxJ60 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>ADC</th></tr>
<tr><td>AN2</td><td><script>document.write(AN2)</script></td></tr>
<tr><td>AN3</td><td><script>document.write(AN3)</script></td></tr>
```

```
</table></td><td><table border=1 style="font-size:20px ;font-family:
terminal ;">
<tr><th colspan=2>PORTB</th></tr>
<script>
var str,i;
str="";
for(i=0;i<8;i++)
{str+="<tr><td bgcolor=pink>BUTTON #"+i+"</td>";
if(PORTB&(1<<i)){str+="<td bgcolor=red>ON";}
else {str+="<td bgcolor=#cccccc>OFF";}
str+="</td></tr>";}
document.write(str) ;
</script>
" ;

const char   *indexPage2 =  "</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<3;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>
" ;

/***********************************
 * RAM variables
 */
unsigned char   myMacAddr[6]  = {0x00, 0x14, 0xA5, 0x76, 0x19, 0x3f} ;
// my MAC address
unsigned  char      myIpAddr[4]     =  {192,   168,     20,   60  } ;
// my IP address
unsigned char    gwIpAddr[4]   = {192, 168,   20,   6 } ;    // gateway
(router) IP address
unsigned char   ipMask[4]    = {255, 255, 255,  0 } ;  // network mask
(for example : 255.255.255.0)
unsigned  char       dnsIpAddr[4]   =  {192,   168,     20,     1  } ;
// DNS server IP address

unsigned char   getRequest[15] ;    // HTTP request buffer
unsigned char   dyna[30] ;      // buffer for dynamic response
unsigned long   httpCounter = 0;   // counter of HTTP requests
```

```
/***************************************
 *  functions
 */

/*
 *  put the constant string pointed to by s to the Ethernet con-
troller's transmit buffer.
 */
/*unsigned int    putConstString(const char *s)
        {
        unsigned int ctr = 0;

        while(*s)
                {
                Ethernet_putByte(*s++);
                ctr++;
                }
        return(ctr);
        }*/
/*
 * it will be much faster to use library Ethernet_putConstString rou-
tine
 *  instead of putConstString routine above. However, the code will
be a little
 * bit bigger. User should choose between size and speed and pick the
implementation that
 * suites him best. If you choose to go with the putConstString def-
inition above
 *  the #define line below should be commented out.
 *
 */
#define putConstString  Ethernet_putConstString

/*
 * put the string pointed to by s to the Ethernet controller's trans-
mit buffer
 */
/*unsigned int    putString(char *s)
        {
        unsigned int ctr = 0;

        while(*s)
                {
                Ethernet_putByte(*s++);

                ctr++;
                }
        return(ctr);
        }*/
/*
```

```
 * it will be much faster to use library Ethernet_putString routine
 * instead of putString routine above. However, the code will be a
little
 * bit bigger. User should choose between size and speed and pick the
implementation that
 * suites him best. If you choose to go with the putString defini-
tion above
 * the #define line below should be commented out.
 *
 */
#define putString  Ethernet_putString

/*
 * this function is called by the library
 * the user accesses to the HTTP request by successive calls to
Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to
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
 *
 */
unsigned int    Ethernet_UserTCP(unsigned char *remoteHost, unsigned
int remotePort, unsigned int localPort, unsigned int reqLength)
        {
        unsigned int    len = 0;          // my reply length
        unsigned char   i;                // general purpose char

        if(localPort != 80)//I listen only to web request on port 80
                {
                return(0);
                }

        // get 10 first bytes only of the request, the rest does not
matter here
        for(i = 0; i < 10; i++)
                {
                getRequest[i] = Ethernet_getByte();
                }
        getRequest[10] = 0;

         if(memcmp(getRequest, httpMethod, 5))// only GET method is
supported here
                {
                return(0);
                }
```

```
         httpCounter++;             // one more request done

        if(getRequest[ 5]  == 's')  // if request path name starts with
s, store dynamic data in transmit buffer
                 {
                  // the text string replied by this request can be
interpreted as javascript statements
                 // by browsers

                 len = putConstString(httpHeader);   // HTTP header
                 len += putConstString(httpMimeTypeScript); // with
text MIME type

                 // add AN2 value to reply
                 IntToStr(ADC_Read(2), dyna);
                 len += putConstString("var AN2=");
                 len += putString(dyna);
                 len += putConstString(";");

                 // add AN3 value to reply
                 IntToStr(ADC_Read(3), dyna);
                 len += putConstString("var AN3=");
                 len += putString(dyna);
                 len += putConstString(";");

                 // add PORTB value (buttons) to reply
                 len += putConstString("var PORTB=");
                 IntToStr(PORTB, dyna);
                 len += putString(dyna);
                 len += putConstString(";");

                 // add PORTD value (LEDs) to reply
                 len += putConstString("var PORTD=");
                 IntToStr(PORTD, dyna);
                 len += putString(dyna);
                 len += putConstString(";");

                 // add HTTP requests counter to reply
                 IntToStr(httpCounter, dyna);
                 len += putConstString("var REQ=");
                 len += putString(dyna);
                 len += putConstString(";");
                 }
        else if(getRequest[ 5]  == 't') // if request path name starts
with t, toggle PORTD (LED) bit number that comes after
                 {
                 unsigned char   bitMask = 0; // for bit mask
                if(isdigit(getRequest[ 6] )) // if 0 <= bit number <=
9, bits 8 & 9 does not exist but does not matter
                          {
```

```
    bitMask = getRequest[ 6] - '0'; // convert ASCII to integer
                bitMask = 1 << bitMask; // create bit mask
                PORTD ^= bitMask; // toggle PORTD with xor operator
                              }
                  }

         if(len == 0)                  // what do to by default
                  {
     len =  putConstString(httpHeader);   // HTTP header
     len += putConstString(httpMimeTypeHTML); // with HTML MIME type
     len += putConstString(indexPage); // HTML page first part
     len += putConstString(indexPage2);// HTML page second part
                  }

          return(len); // return to the library with the number of
bytes to transmit
        }

/*
 * this function is called by the library
  * the user accesses to the UDP request by successive calls to
Ethernet_getByte()
 * the user puts data in the transmit buffer by successive calls to
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
 *
 */
unsigned int    Ethernet_UserUDP(unsigned char *remoteHost, unsigned
int remotePort, unsigned int destPort, unsigned int reqLength)
        {
        unsigned int    len;     // my reply length

       // reply is made of the remote host IP address in human read-
able format
        ByteToStr(remoteHost[ 0] , dyna);// first IP address byte
        dyna[ 3]  = '.';
        ByteToStr(remoteHost[ 1] , dyna + 4);      // second
        dyna[ 7]  = '.';
        ByteToStr(remoteHost[ 2] , dyna + 8);      // third
        dyna[ 11]  = '.';
        ByteToStr(remoteHost[ 3] , dyna + 12);      // fourth

        dyna[ 15]  = ':';                       // add separator

        // then remote host port number
        WordToStr(remotePort, dyna + 16);
```

```
        dyna[21] = '[';
        WordToStr(destPort, dyna + 22);
        dyna[27] = ']';
        dyna[28] = 0;

         // the total length of the request is the length of the
dynamic string plus the text of the request
        len = 28 + reqLength;

        // puts the dynamic string into the transmit buffer
        Ethernet_putBytes(dyna, 28);

         // then puts the request string converted into upper char
into the transmit buffer
        while(reqLength--)
                {
                Ethernet_putByte(toupper(Ethernet_getByte()));
                }

        return(len);   // back to the library with the length of the
UDP reply
        }

/*
 * main entry
 */
void    main()
        {
        ADCON1 = 0x0B; // ADC convertors will be used with AN2 and AN3
        CMCON  = 0x07;          // turn off comparators

        PORTA = 0;
        TRISA = 0xfc;           // set PORTA as input for ADC
                      // except RA0 and RA1 which will be used as
                                // ethernet's LEDA and LEDB

        PORTB = 0;
        TRISB = 0xff;           // set PORTB as input for buttons

        PORTD = 0;
        TRISD = 0;              // set PORTD as output

        /*
         * Initialize Ethernet controller
         */
        Ethernet_Init(myMacAddr, myIpAddr, Ethernet_FULLDUPLEX);

       //dhcp will not be used here, so use preconfigured addresses
        Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr);
```

```
while(1)                            // do forever
        {
   /*
    * if necessary, test the return value to get error code
   */
 Ethernet_doPacket(); // process incoming Ethernet packets

/*
 * add your stuff here if needed
 * Ethernet_doPacket() must be called as often as possible
 * otherwise packets could be lost
 */
        }
}
```

FLASH MEMORY LIBRARY

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

Note: Due to the P16/P18 family flash specifics, flash library is MCU dependent. Since the P18 family differ significantlly in number of bytes that can be erased and/or written to specific MCUs, the appropirate suffix is added to the names of functions in order to make it easier to use them. Flash memory operations are MCU dependent :

1. **Read** operation supported. For this group of MCU's only read function is imple mented.
2. **Read** and **Write** operations supported (write is executed as erase-and-write). For this group of MCU's read and write functions are implemented. Note that write operation which is executed as erase-and-write, may write less bytes than it erases.
3. **Read**, **Write** and **Erase** operations supported. For this group of MCU's read, write and erase functions are implemented. Further more, flash memory block has to be erased prior to writting (write operation is not executed as erase-and-write).

Please refer to MCU datasheet before using flash library.

Library Routines

- FLASH_Read
- FLASH_Read_N_Bytes
- 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

Prototype	`// for PIC16`
	`unsigned FLASH_Read(unsigned address);`
	
	`// for PIC18`
	`unsigned short FLASH_Read(long address);`

Returns	Returns data byte from Flash memory.
Description	Reads data from the specified address in Flash memory.
Requires	Nothing.

Example	`// for PIC18`
	`unsigned short tmp;`
	`...`
	`tmp = FLASH_Read(0x0D00);`
	`...`

FLASH_Read_N_Bytes

Prototype	`void FLASH_Read_N_Bytes(long address, char* data_, unsigned int N);`
Returns	Nothing.
Description	Reads N data from the specified `address` in Flash memory to varibale pointed by `data`
Requires	Nothing.
Example	`FLASH_Read_N(0x0D00,data_buffer,sizeof(data_buffer));`

FLASH_Write

Prototype

```
// for PIC16
void FLASH_Write(unsigned address, unsigned int* data);
// for PIC18
void FLASH_Write_8(long address, char* data);
void FLASH_Write_16(long address, char* data);
void FLASH_Write_32(long address, char* data);
void FLASH_Write_64(long address, char* data);
```

Returns	Nothing.
Description	Writes block of data to Flash memory. Block size is MCU dependent. **P16**: This function may erase memory segment before writing block of data to it (MCU dependent). Furthermore, memory segment which will be erased may be greater than the size of the data block that will be written (MCU dependent). Therefore it is recommended to write as many bytes as you erase. FLASH_Write writes 4 flash memory locations in a row, so it needs to be called as many times as it is necessary to meet the size of the data block that will be written. **P18**: This function does not perform erase prior to write.
Requires	Flash memory that will be written may have to be erased before this function is called (MCU dependent). Refer to MCU datasheet for details.

Example

Write consecutive values in 64 consecutive locations, starting from 0x0D00:

```
unsigned short toWrite[ 64];
...
// initialize array:
for (i = 0; i < 64; i++)
  toWrite[ i] = i;

// write contents of the array to the address 0x0D00:
FLASH_Write_64(0x0D00, toWrite);
```

FLASH_Erase

Prototype	`// for PIC16`
	`void FLASH_Erase(unsigned address);`
	`// for PIC18`
	`void FLASH_Erase_64(long address);`
	`void FLASH_Erase_1024(long address);`

Returns	Nothing.
Description	Erases memory block starting from a given address. For P16 familly is implemented only for those MCU's whose flash memory does not support erase-and-write operations (refer to datasheet for details).
Requires	Nothing.
Example	Erase 64 byte memory memory block, starting from address 0x0D00: `FLASH_Erase_64(0x0D00);`

FLASH_Erase_Write

Prototype	`// for PIC18`
	`void FLASH_Erase_Write_64(long address, char* data);`
	`void FLASH_Erase_Write_1024(long address, char* data);`

Returns	None.
Description	Erase then write memory block starting from a given address.
Requires	Nothing.

Example	`char toWrite[64];`
	`int i;`
	`...`
	`// initialize array:`
	`for(i=0; i<64; i++) toWrite[i]=i;`
	`// erase block of memory at address 0x0D00 then write contents of the array to the address 0x0D00:`
	`FLASH_Erase_Write_64(0x0D00, toWrite);`

Library Example

The example demonstrates simple write to the flash memory for PIC16F887, then reads the data and displays it on PORTB and PORTC.

```
char i = 0;
unsigned int addr, data_, dataAR[4][4] = {{ 0x3FAA+0,  0x3FAA+1,
0x3FAA+2,  0x3FAA+3},
                                         { 0x3FAA+4,  0x3FAA+5,
0x3FAA+6,  0x3FAA+7},
                                         { 0x3FAA+8,  0x3FAA+9,
0x3FAA+10, 0x3FAA+11},
                                         {0x3FAA+12, 0x3FAA+13,
0x3FAA+14, 0x3FAA+15}};

void main() {
  ANSEL  = 0;                // Configure AN pins as digital I/O
  ANSELH = 0;
  PORTB = 0;                           // Initial PORTB value
  TRISB = 0;                           // Set PORTB as output
  PORTC = 0;                           // Initial PORTC value
  TRISC = 0;                           // Set PORTC as output
  Delay_ms(500);

  // All block writes
  // to program memory are done as 16-word erase by
  // eight-word write operations. The write operation is
  // edge-aligned and cannot occur across boundaries.
  // Therefore it is recommended to perform flash writes in 16-word
chunks.
  // That is why lower 4 bits of start address [3:0] must be zero.
  // Since FLASH_Write routine performs writes in 4-word chunks,
  // we need to call it 4 times in a row.
  addr = 0x0430;    // starting Flash address, valid for P16F887
  for (i = 0; i < 4; i++){     // Write some data to Flash
    Delay_ms(100);
    FLASH_Write(addr+i*4, dataAR[i]);
  }
  Delay_ms(500);

  addr = 0x0430;
  for (i = 0; i < 16; i++){
    data_ = FLASH_Read(addr++);  // P16's FLASH is 14-bit wide, so
    Delay_us(10);               //  two MSB's will always be '00'
    PORTB = data_;              // Display data on PORTB LS Byte
    PORTC = data_ >> 8;         // and PORTC MS Byte
    Delay_ms(500);
  }
}
```

GRAPHIC LCD LIBRARY

The *mikroC PRO for PIC* 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 Graphic LCD Library:	**Description:**	**Example:**
`extern sfr char GLCD_DataPort;`	Glcd Data Port.	`char GLCD_DataPort at PORTD;`
`extern sfr sbit GLCD_CS1;`	Chip Select 1 line.	`sbit GLCD_CS1 at RB0_bit;`
`extern sfr sbit GLCD_CS2;`	Chip Select 2 line.	`sbit GLCD_CS2 at RB1_bit;`
`extern sfr sbit GLCD_RS;`	Register select line.	`sbit GLCD_RS at RB2_bit;`
`extern sfr sbit GLCD_RW;`	Read/Write line.	`sbit GLCD_RW at RB3_bit;`
`extern sfr sbit GLCD_EN;`	Enable line.	`sbit GLCD_EN at RB4_bit;`
`extern sfr sbit GLCD_RST;`	Reset line.	`sbit GLCD_RST at RB5_bit;`
`extern sfr sbit GLCD_CS1_Direction;`	Direction of the Chip Select 1 pin.	`sbit GLCD_CS1_Direction at TRISB0_bit;`
`extern sfr sbit GLCD_CS2_Direction;`	Direction of the Chip Select 2 pin.	`sbit GLCD_CS2_Direction at TRISB1_bit;`
`extern sfr sbit GLCD_RS_Direction;`	Direction of the Register select pin.	`sbit GLCD_RS_Direction at TRISB2_bit;`
`extern sfr sbit GLCD_RW_Direction;`	Direction of the Read/Write pin.	`sbit GLCD_RW_Direction at TRISB3_bit;`
`extern sfr sbit GLCD_EN_Direction;`	Direction of the Enable pin.	`sbit GLCD_EN_Direction at TRISB4_bit;`
`extern sfr sbit GLCD_RST_Direction;`	Direction of the Reset pin.	`sbit GLCD_RST_Direction at TRISB5_bit;`

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

Prototype	`void Glcd_Init();`
Returns	Nothing.
Description	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>).
Requires	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 must be defined before using this function.

Example

```
// glcd pinout settings
char GLCD_DataPort at PORTD;

sbit GLCD_CS1 at RB0_bit;
sbit GLCD_CS2 at RB1_bit;
sbit GLCD_RS  at RB2_bit;
sbit GLCD_RW  at RB3_bit;
sbit GLCD_EN at RB4_bit;
sbit GLCD_RST at RB5_bit;

sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction at TRISB2_bit;
sbit GLCD_RW_Direction at TRISB3_bit;
sbit GLCD_EN_Direction at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;
...
ANSEL = 0;
ANSELH = 0;
Glcd_Init();
```

Glcd_Set_Side

Prototype	`void Glcd_Set_Side(unsigned short x_pos);`
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	`void Glcd_Set_X(unsigned short x_pos);`
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	`void Glcd_Set_Page(unsigned short page);`
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	`unsigned short Glcd_Read_Data();`
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	`unsigned short data;`
	`...`
	`data = Glcd_Read_Data();`

Glcd_Write_Data

Prototype	`void Glcd_Write_Data(unsigned short ddata);`
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	`unsigned short data;`
	`...`
	`Glcd_Write_Data(data);`

Glcd_Fill

Prototype	`void Glcd_Fill(unsigned short pattern);`
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	`void Glcd_Dot(unsigned short x_pos, unsigned short y_pos,` `unsigned short color);`
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	`void Glcd_Line(int x_start, int y_start, int x_end, int y_end,` `unsigned short color);`
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	`void Glcd_V_Line(unsigned short y_start, unsigned short y_end, unsigned short x_pos, unsigned short color);`
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	`void Glcd_H_Line(unsigned short x_start, unsigned short x_end, unsigned short y_pos, unsigned short color);`
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	`void Glcd_Rectangle(unsigned short x_upper_left, unsigned short y_upper_left, unsigned short x_bottom_right, unsigned short y_bottom_right, unsigned short color);`
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	`void Glcd_Box(unsigned short x_upper_left, unsigned short y_upper_left, unsigned short x_bottom_right, unsigned short y_bottom_right, unsigned short color);`
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	`void Glcd_Circle(int x_center, int y_center, int radius, unsigned short color);`
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	`void Glcd_Set_Font(const char *activeFont, unsigned short aFontWidth, unsigned short aFontHeight, unsigned int aFontOffs);`
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 mikroC PRO for PIC character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroC PRO for PIC 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" 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	`void Glcd_Write_Char(unsigned short chr, unsigned short x_pos, unsigned short page_num, unsigned short color);`
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	`void Glcd_Write_Text(char *text, unsigned short x_pos, unsigned short page_num, unsigned short color);`
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	`void Glcd_Image(code const unsigned short *image);`
Returns	Nothing.
Description	Displays bitmap on GLCD. Parameters: - `image`: image to be displayed. Bitmap array must be located in code memory. Use the *mikroC PRO for PIC* 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

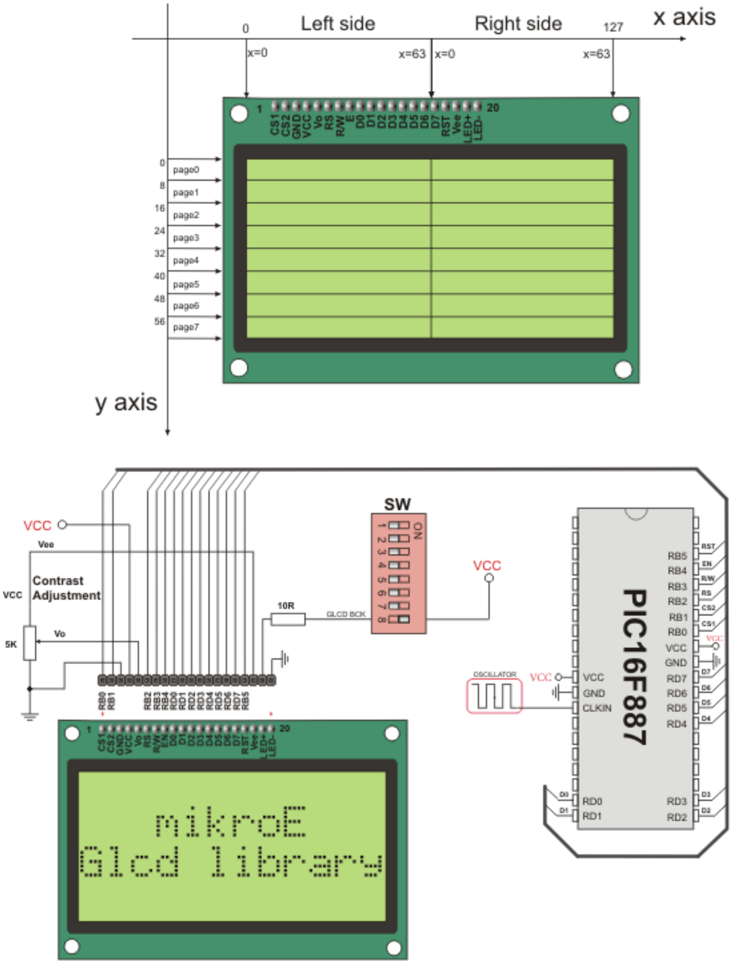
The following example demonstrates routines of the Glcd library: initialization, clear(pattern fill), image displaying, drawing lines, circles, boxes and rectangles, text displaying and handling.

```
//Declarations---------------------------------------------------------
------------
const code char truck_bmp[1024];
//--------------------------------------------------------------------end-
declarations

// Glcd module connections
char GLCD_DataPort at PORTD;

sbit GLCD_CS1 at RB0_bit;
sbit GLCD_CS2 at RB1_bit;
sbit GLCD_RS  at RB2_bit;
sbit GLCD_RW  at RB3_bit;
sbit GLCD_EN  at RB4_bit;
sbit GLCD_RST at RB5_bit;

sbit GLCD_CS1_Direction at TRISB0_bit;
sbit GLCD_CS2_Direction at TRISB1_bit;
sbit GLCD_RS_Direction  at TRISB2_bit;
sbit GLCD_RW_Direction  at TRISB3_bit;
sbit GLCD_EN_Direction  at TRISB4_bit;
sbit GLCD_RST_Direction at TRISB5_bit;
// End Glcd module connections

void delay2S(){                      // 2 seconds delay function
  Delay_ms(2000);
}

void main() {
  unsigned short ii;
  char *someText;

     #define COMPLETE_EXAMPLE   // comment this line to make
simpler/smaller example
  ANSEL  = 0;                  // Configure AN pins as digital
  ANSELH = 0;
  C1ON_bit = 0;                 // Disable comparators
  C2ON_bit = 0;

  Glcd_Init();                                  // Initialize GLCD
  Glcd_Fill(0x00);                              // Clear GLCD

  while(1) {
```

```
    #ifdef COMPLETE_EXAMPLE
      Glcd_Image(truck_bmp);            // Draw image
      delay2S(); delay2S();
    #endif

    Glcd_Fill(0x00);                                // Clear GLCD

    Glcd_Box(62,40,124,56,1);                      // Draw box
    Glcd_Rectangle(5,5,84,35,1);                   // Draw rectangle
    Glcd_Line(0, 0, 127, 63, 1);                   // Draw line
    delay2S();

    for(ii = 5; ii < 60; ii+=5 ){  // Draw horizontal and vertical lines
      Delay_ms(250);
      Glcd_V_Line(2, 54, ii, 1);
      Glcd_H_Line(2, 120, ii, 1);
    }

    delay2S();

    Glcd_Fill(0x00);                                // Clear GLCD
    #ifdef COMPLETE_EXAMPLE
        Glcd_Set_Font(Character8x7, 8, 7, 32);// Choose font, see
__Lib_GLCDFonts.c in Uses folder
    #endif
    Glcd_Write_Text("mikroE", 1, 7, 2);      // Write string

    for (ii = 1; ii <= 10; ii++)                  // Draw circles
      Glcd_Circle(63,32, 3*ii, 1);
    delay2S();

    Glcd_Box(12,20, 70,57, 2);                     // Draw box
    delay2S();

    #ifdef COMPLETE_EXAMPLE
      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(System3x5, 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();
    #endif
  }
}
```

HW Connection



Glcd HW connection

I^{2}C LIBRARY

I^{2}C full master MSSP module is available with a number of PIC MCU models. *mikroC PRO for PIC* provides library which supports the master I^{2}C mode.

Note: Some MCUs have multiple I^{2}C modules. In order to use the desired I^{2}C library routine, simply change the number 1 in the prototype with the appropriate module number, i.e. `I2C1_Init(100000);`

Library Routines

- I2C1_Init
- I2C1_Start
- I2C1_Repeated_Start
- I2C1_Is_Idle
- I2C1_Rd
- I2C1_Wr
- I2C1_Stop

I2C1_Init

Prototype	`void I2C1_Init(unsigned long clock);`
Returns	Nothing.
Description	Initializes I^{2}C with desired `clock` (refer to device data sheet for correct values in respect with Fosc). Needs to be called before using other functions of I^{2}C 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. **Note**: Calculation of the I^{2}C clock value is carried out by the compiler, as it would produce a relatively large code if performed on the libary level. Therefore, compiler needs to know the value of the parameter in the compile time. That is why this parameter needs to be a constant, and not a variable.
Example	`I2C1_Init(100000);`

I2C1_Start

Prototype	`unsigned short I2C1_Start(void);`
Returns	If there is no error, function returns 0.
Description	Determines if I^2C bus is free and issues START signal.
Requires	I^2C must be configured before using this function. See I2C1_Init.
Example	`I2C1_Start();`

I2C1_Repeated_Start

Prototype	`void I2C1_Repeated_Start(void);`
Returns	Nothing.
Description	Issues repeated START signal.
Requires	I^2C must be configured before using this function. See I2C1_Init.
Example	`I2C1_Repeated_Start();`

I2C1_Is_Idle

Prototype	`unsigned short I2C1_Is_Idle(void);`
Returns	Returns 1 if I^2C bus is free, otherwise returns 0.
Description	Tests if I^2C bus is free.
Requires	I^2C must be configured before using this function. See I2C1_Init.
Example	`if (I2C1_Is_Idle()) {...}`

I2C1_Rd

Prototype	`unsigned short I2C1_Rd(unsigned short ack);`
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	I^2C must be configured before using this function. See I2C1_Init. Also, START signal needs to be issued in order to use this function. See I2C1_Start.

Example	Read data and send not acknowledge signal:
	`unsigned short take;`
	`...`
	`take = I2C1_Rd(0);`

I2C1_Wr

Prototype	`unsigned short I2C1_Wr(unsigned short data_);`
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter data) via I^{2}C bus.
Requires	I^{2}C must be configured before using this function. See I2C1_Init. Also, START signal needs to be issued in order to use this function. See I2C1_Start.
Example	`I2C1_Write(0xA3);`

I2C1_Stop

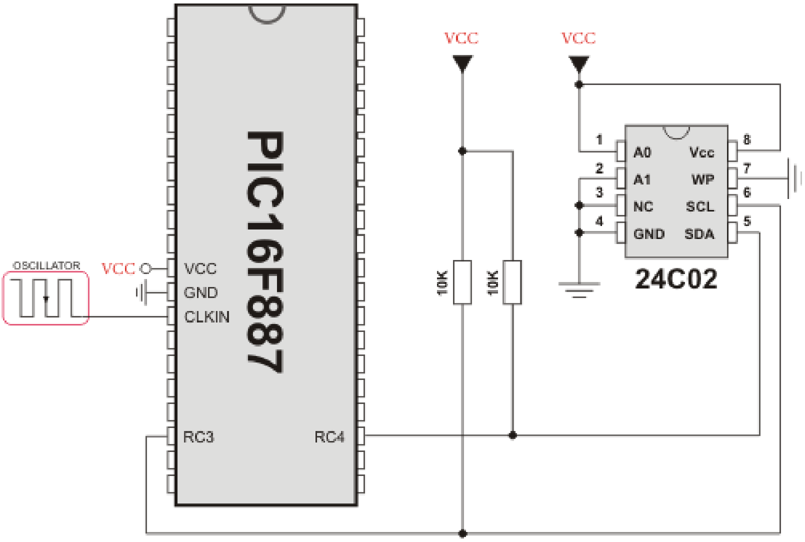
Prototype	`void I2C1_Stop(void);`
Returns	Nothing.
Description	Issues STOP signal.
Requires	I^{2}C must be configured before using this function. See I2C1_Init.
Example	`I2C1_Stop();`

Library Example

This code demonstrates use of I¸C library. PIC MCU is connected (SCL, SDA pins) to 24c02 EEPROM. Program sends data to EEPROM (data is written at address 2). Then, we read data via I¸C from EEPROM and send its value to PORTB, to check if the cycle was successful (see the figure below how to interface 24c02 to PIC).

```
void main(){
  ANSEL  = 0;              // Configure AN pins as digital I/O
  ANSELH = 0;
  PORTB = 0;
  TRISB = 0;               // Configure PORTB as output

  I2C1_Init(100000);       // initialize I2C communication
  I2C1_Start();            // issue I2C start signal
  I2C1_Wr(0xA2);           // send byte via I2C  (device address + W)
  I2C1_Wr(2);              // send byte (address of EEPROM location)
  I2C1_Wr(0xF0);           // send data (data to be written)
  I2C1_Stop();             // issue I2C stop signal

  Delay_100ms();

  I2C1_Start();            // issue I2C start signal
  I2C1_Wr(0xA2);           // send byte via I2C  (device address + W)
  I2C1_Wr(2);              // send byte (data address)
  I2C1_Repeated_Start();  // issue I2C signal repeated start
  I2C1_Wr(0xA3);           // send byte (device address + R)
  PORTB = I2C1_Rd(0u);     // Read the data (NO acknowledge)
  I2C1_Stop();             // issue I2C stop signal
}
```

HW Connection



Interfacing 24c02 to PIC via I^2C

KEYPAD LIBRARY

The *mikroC PRO for PIC* 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.

External dependencies of Keypad Library

The following variable must be defined in all projects using Keypad Library:	**Description:**	**Example:**
`extern sfr char keypadPort;`	Keypad Port.	`char keypadPort at PORTD;`

Library Routines

- Keypad_Init
- Keypad_Key_Press
- Keypad_Key_Click

Keypad_Init

Prototype	`void Keypad_Init(void);`
Returns	Nothing.
Description	Initializes port for working with keypad.
Requires	Global variable: - `keypadPort` - Keypad port must be defined before using this function.
Example	

```
// Keypad module connections
char keypadPort at PORTD;
// End of keypad module connections
...
Keypad_Init();
```

Keypad_Key_Press

Prototype	`char Keypad_Key_Press(void);`
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	`char kp;`
	`...`
	`kp = Keypad_Key_Press();`

Keypad_Key_Click

Prototype	`char Keypad_Key_Click(void);`
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	`char kp;`
	`...`
	`kp = Keypad_Key_Click();`

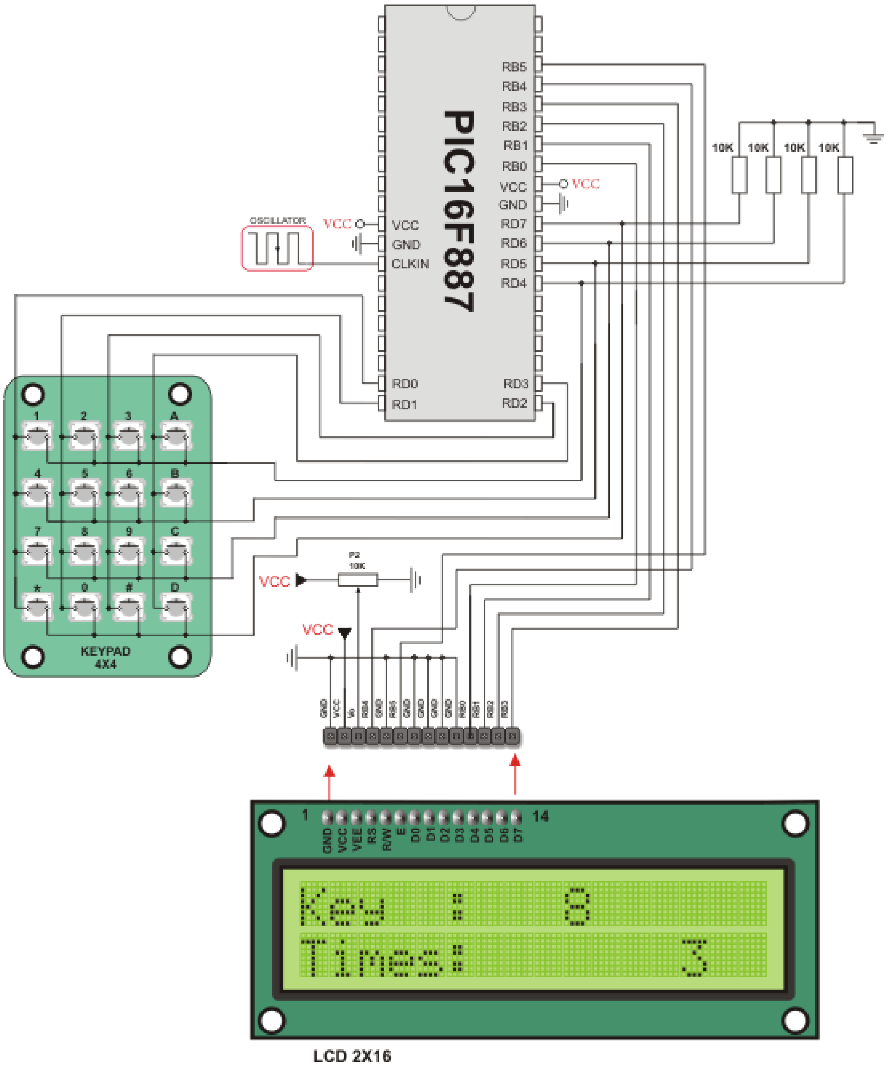
Library Example

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.

```
unsigned short kp, cnt, oldstate = 0;
char txt[ 6] ;

// Keypad module connections
char  keypadPort at PORTD;
// End Keypad module connections

// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

void main() {
  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, "1");
  Lcd_Out(1, 1, "Key  :");           // Write message text on Lcd
  Lcd_Out(2, 1, "Times:");

  do {
    kp = 0;                          // Reset key code variable

    // Wait for key to be pressed and released
    do
    //kp = Keypad_Key_Press();       // Store key code in kp variable
      kp = Keypad_Key_Click();      // Store key code in kp variable
    while (!kp);
```

```
    // Prepare value for output, transform key to it's ASCII value
     switch (kp) {
        //case 10: kp = 42; break;  // '*' // Uncomment this block for
keypad4x3
        //case 11: kp = 48; break;  // '0'
        //case 12: kp = 35; break;  // '#'
        //default: kp += 48;

      case  1: kp = 49; break; // 1 // Uncomment this block for keypad4x4
        case  2: kp = 50; break; // 2
        case  3: kp = 51; break; // 3
        case  4: kp = 65; break; // A
        case  5: kp = 52; break; // 4
        case  6: kp = 53; break; // 5
        case  7: kp = 54; break; // 6
        case  8: kp = 66; break; // B
        case  9: kp = 55; break; // 7
        case 10: kp = 56; break; // 8
        case 11: kp = 57; break; // 9
        case 12: kp = 67; break; // C
        case 13: kp = 42; break; // *
        case 14: kp = 48; break; // 0
        case 15: kp = 35; break; // #
        case 16: kp = 68; break; // D

     }

     if (kp != oldstate) {  // Pressed key differs from previous
        cnt = 1;
        oldstate = kp;
        }
     else {                 // Pressed key is same as previous
        cnt++;
        }

     Lcd_Chr(1, 10, kp);    // Print key ASCII value on Lcd

     if (cnt == 255) {      // If counter varialble overflow
        cnt = 0;
        Lcd_Out(2, 10, "   ");
        }

     WordToStr(cnt, txt);   // Transform counter value to string
     Lcd_Out(2, 10, txt);   // Display counter value on Lcd
  } while (1);
}
```

HW Connection



4x4 Keypad connection scheme

LCD LIBRARY

The *mikroC PRO for PIC* 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:
`extern sfr sbit LCD_RS:`	Register Select line.	`sbit LCD_RS at RB4_bit;`
`extern sfr sbit LCD_EN:`	Enable line.	`sbit LCD_EN at RB5_bit;`
`extern sfr sbit LCD_D7;`	Data 7 line.	`sbit LCD_D7 at RB3_bit;`
`extern sfr sbit LCD_D6;`	Data 6 line.	`sbit LCD_D6 at RB2_bit;`
`extern sfr sbit LCD_D5;`	Data 5 line.	`sbit LCD_D5 at RB1_bit;`
`extern sfr sbit LCD_D4;`	Data 4 line.	`sbit LCD_D4 at RB0_bit;`
`extern sfr sbit LCD_RS_Direction;`	Register Select direction pin.	`sbit LCD_RS_Direction at TRISB4_bit;`
`extern sfr sbit LCD_EN_Direction;`	Enable direction pin.	`sbit LCD_EN_Direction at TRISB5_bit;`
`extern sfr sbit LCD_D7_Direction;`	Data 7 direction pin.	`sbit LCD_D7_Direction at TRISB3_bit;`
`extern sfr sbit LCD_D6_Direction;`	Data 6 direction pin.	`sbit LCD_D6_Direction at TRISB2_bit;`
`extern sfr sbit LCD_D5_Direction;`	Data 5 direction pin.	`sbit LCD_D5_Direction at TRISB1_bit;`
`extern sfr sbit LCD_D4_Direction;`	Data 4 direction pin.	`sbit LCD_D4_Direction at TRISB0_bit;`

Library Routines

- Lcd_Init
- Lcd_Out
- Lcd_Out_Cp
- Lcd_Chr
- Lcd_Chr_Cp
- Lcd_Cmd

Lcd_Init

Prototype	`void 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 pinout settings
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D7 at RB3_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D4 at RB0_bit;
// Pin direction
sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D7_Direction at TRISB3_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D4_Direction at TRISB0_bit;
...
Lcd_Init();
```

Lcd_Out

Prototype	`void Lcd_Out(char row, char column, char *text);`
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	`void Lcd_Out_CP(char *text);`
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	`void Lcd_Chr(char row, char column, char out_char);`
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	`void Lcd_Chr_CP(char out_char);`
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	`void Lcd_Cmd(char out_char);`
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

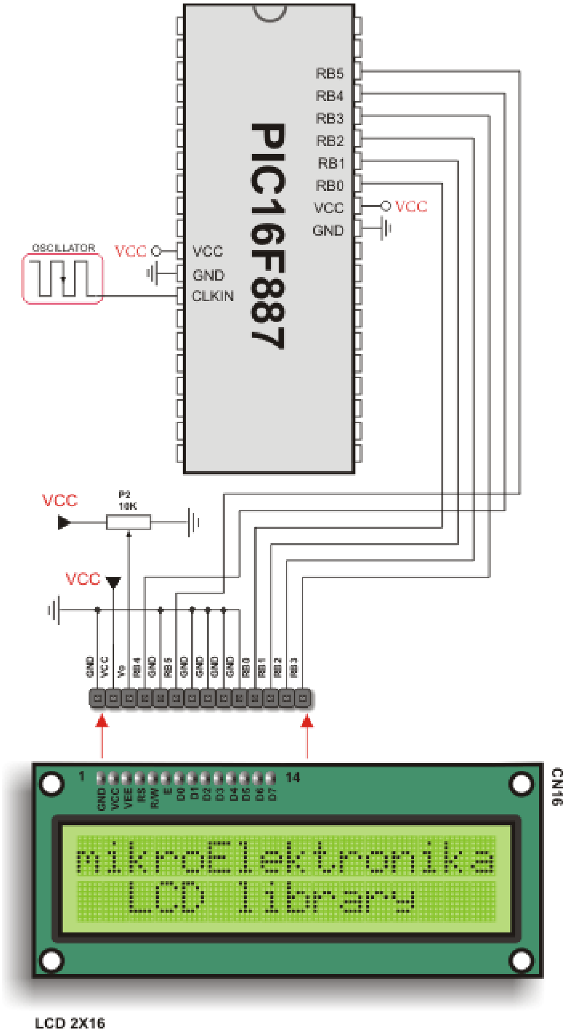
Library Example

The following code demonstrates usage of the Lcd Library routines:

```
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

char txt1[] = "mikroElektronika";
char txt2[] = "EasyPIC5";
char txt3[] = "Lcd4bit";
char txt4[] = "example";

char i;                          // Loop variable

void Move_Delay() {              // Function used for text moving
  Delay_ms(500);                 // You can change the moving speed here
}

void main(){
  TRISB = 0;
  PORTB = 0xFF;
  TRISB = 0xff;
  ANSEL  = 0;                    // Configure AN pins as digital I/O
  ANSELH = 0;
  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(2,5,txt2);                   // Write text in second row
```

```
    Delay_ms(2000);

    // Moving text
    for(i=0; i<4; i++) {      // Move text to the right 4 times
      Lcd_Cmd(_LCD_SHIFT_RIGHT);
      Move_Delay();
    }

    while(1) {                           // Endless loop
      for(i=0; i<8; i++) {          // Move text to the left 7 times
        Lcd_Cmd(_LCD_SHIFT_LEFT);
        Move_Delay();
      }

      for(i=0; i<8; i++) {          // Move text to the right 7 times
        Lcd_Cmd(_LCD_SHIFT_RIGHT);
        Move_Delay();
      }
    }
  }
```

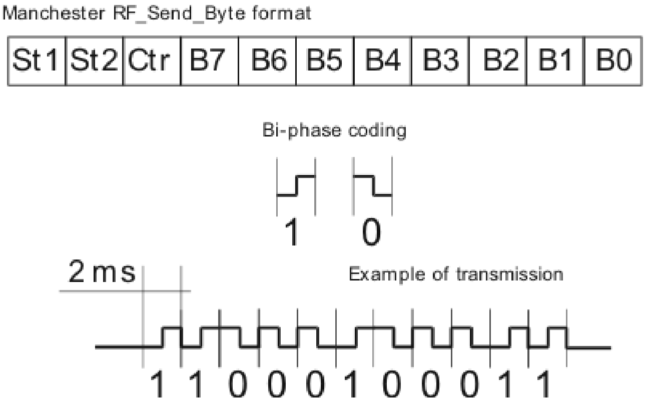
HW connection



LCD HW connection

MANCHESTER CODE LIBRARY

The *mikroC PRO for PIC* 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).



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 Manchester Code Library:	Description:	Example:
`extern sfr sbit MANRXPIN;`	Receive line.	`sbit MANRXPIN at RC0_bit;`
`extern sfr sbit MANTXPIN;`	Transmit line.	`sbit MANTXPIN at RC1_bit;`
`extern sfr sbit MANRXPIN_Direction;`	Direction of the Receive pin.	`sbit MANRXPIN_Direction at TRISC0_bit;`
`extern sfr sbit MANTXPIN_Direction;`	Direction of the Transmit pin.	`sbit MANTXPIN_Direction at TRISC1_bit;`

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	`unsigned int Man_Receive_Init();`
Returns	- `0` - if initialization and synchronization were successful. - `1` - upon unsuccessful synchronization. - `255` - upon user abort.
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
sbit MANRXPIN at RC0_bit;
sbit MANRXPIN_Direction at TRISC0_bit;
...
Man_Receive_Init();
```

Man_Receive

Prototype	`unsigned char Man_Receive(unsigned char *error);`
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	see below

```
unsigned char data = 0, error = 0;
...
data = Man_Receive(&error);
if (error)
  { /* error handling */ }
```

Man_Send_Init

Prototype	`void Man_Send_Init();`
Returns	Nothing.
Description	The function configures Transmitter pin.
Requires	Global variables: `MANTXPIN:` Transmit line `MANTXPIN_Direction:` Direction of the transmit pin must be defined before using this function.
Example	see below

```
// Initialize Transmitter:
sbit MANTXPIN at RC1_bit;
sbit MANTXPIN_Direction at TRISC1_bit;
...
Man_Send_Init();
```

Man_Send

Prototype	`void Man_Send(unsigned char tr_data);`
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	`unsigned char msg;`
	`...`
	`Man_Send(msg);`

Man_Synchro

Prototype	`unsigned char Man_Synchro();`
Returns	- `255` - 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	`unsigned int man__half_bit_len;`
	`...`
	`man__half_bit_len = Man_Synchro();`

Man_Break

Prototype	`void 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

```
char data1, error, counter = 0;
void interrupt {

  if (INTCON.T0IF) {
    if (counter >= 20) {
      Man_Break();
      counter = 0;                // reset counter
    }
    else
      counter++;                  // increment counter

    INTCON.T0IF = 0;   // Clear Timer0 overflow interrupt flag

    }
}
void main() {

  OPTION_REG = 0x04;         // TMR0 prescaler set to 1:32

  ...

  Man_Receive_Init();

  ...

  // try Man_Receive with blocking prevention mechanism
  INTCON.GIE = 1;                 // Global interrupt enable
  INTCON.T0IE = 1;                // Enable Timer0 overflow
interrupt
  data1 = Man_Receive(&error);
  INTCON.GIE = 0;                 // Global interrupt disable

  ...

}
```

Library Example

The following code is code for the Manchester receiver, it shows how to use the Manchester Library for receiving data:

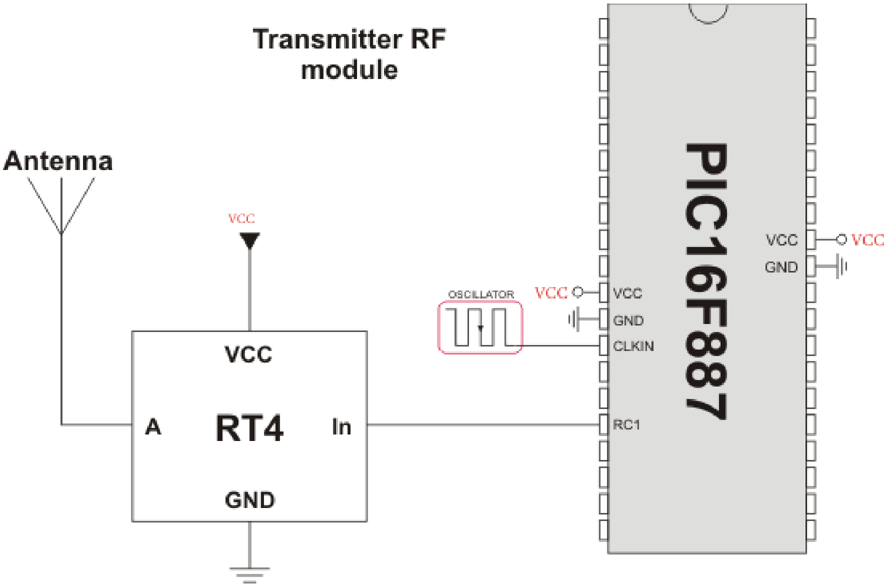
```
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections


// Manchester module connections
sbit MANRXPIN at RC0_bit;
sbit MANRXPIN_Direction at TRISC0_bit;
sbit MANTXPIN at RC1_bit;
sbit MANTXPIN_Direction at TRISC1_bit;
// End Manchester module connections

char error, ErrorCount, temp;

void main() {
  ErrorCount = 0;
  ANSEL  = 0;              // Configure AN pins as digital I/O
  ANSELH = 0;
  TRISC.F5 = 0;
  Lcd_Init();              // Initialize LCD
  Lcd_Cmd(_LCD_CLEAR);     // Clear LCD display

  Man_Receive_Init();      // Initialize Receiver

  while (1) {              // Endless loop

      Lcd_Cmd(_LCD_FIRST_ROW);          // Move cursor to the 1st row

      while (1) {                       // Wait for the "start" byte
        temp = Man_Receive(&error);     // Attempt byte receive
        if (temp == 0x0B)// "Start" byte, see Transmitter example
          break;         // We got the starting sequence
```

```
          if (error)          // Exit so we do not loop forever
            break;
          }

        do
          {
            temp = Man_Receive(&error); // Attempt byte receive
            if (error) {                   // If error occured
              Lcd_Chr_CP('?');        // Write question mark on LCD
              ErrorCount++;           // Update error counter
              if (ErrorCount > 20) {  // In case of multiple errors
                temp = Man_Synchro(); // Try to synchronize again
              //Man_Receive_Init();  // Alternative, try to Initialize
Receiver again
                ErrorCount = 0;     // Reset error counter
                }
              }
            else {                    // No error occured
                if (temp != 0x0E)  // If "End" byte was received(see
Transmitter example)
               Lcd_Chr_CP(temp);//   do not write received byte on LCD
                }
            Delay_ms(25);
          }
       while (temp != 0x0E);// If "End" byte was received exit do loop
    }
}
```

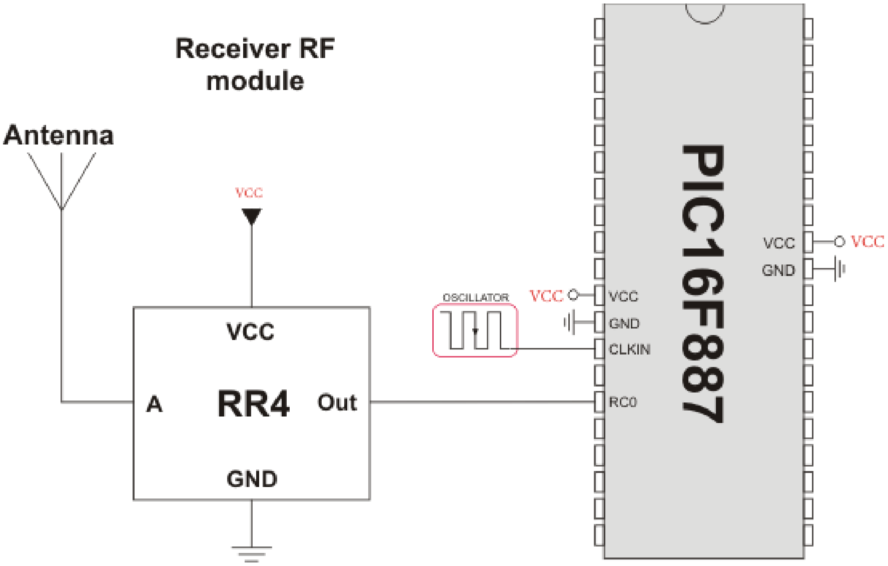
The following code is code for the Manchester transmitter, it shows how to use the Manchester Library for transmitting data:

```
// Manchester module connections
sbit MANRXPIN at RC0_bit;
sbit MANRXPIN_Direction at TRISC0_bit;
sbit MANTXPIN at RC1_bit;
sbit MANTXPIN_Direction at TRISC1_bit;
// End Manchester module connections

char index, character;
char s1[] = "mikroElektronika";

void main() {

  Man_Send_Init();                 // Initialize transmitter

  while (1) {                      // Endless loop
    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) {            // String ends with zero
      Man_Send(character);         // Send character
      Delay_ms(90);                // Wait for a while
      index++;                     // Increment index variable
      character = s1[index];       // Take next char from string
      }
    Man_Send(0x0E);                // Send "end" byte
    Delay_ms(1000);
  }
}
```

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

Secure Digital Card

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

Notes:

- Library works with PIC18 family only;
- The library uses the SPI module for communication. User must initialize SPI module before using the SPI Graphic Lcd Library.
- For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine.
- 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.

Note: The SPI module has to be initialized through `SPI1_Init_Advanced` routine with the following parameters:

- SPI Master
- 8bit mode
- primary prescaler 16
- Slave Select disabled
- data sampled in the middle of data output time
- clock idle low
- Serial output data changes on transition from idle clock state to active clock state

`SPI1_Init_Advanced(_SPI_MASTER_OSC_DIV16, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_LOW, _SPI_LOW_2_HIGH)`; must be called before initializing Mmc_Init.

Note: Once the MMC/SD card is initialized, the user can reinitialize SPI at higher speed. See the Mmc_Init and Mmc_Fat_Init routines.

External dependencies of MMC Library

The following variable must be defined in all projects using MMC library:	Description:	Example:
`extern sfr sbit Mmc_Chip_Select;`	Chip select pin.	`sbit Mmc_Chip_Select at RC2_bit`
`extern sfr sbit Mmc_Chip_Select_Direction;`	Direction of the chip select pin.	`sbit Mmc_Chip_Select_Direction at TRISC2_bit;`

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	`unsigned char Mmc_Init();`
Returns	`- 0` - if MMC/SD card was detected and successfuly initialized `- 1` - otherwise
Description	Initializes hardware SPI communication; The function returns 1 if MMC card is present and successfuly initialized, otherwise returns 0. Mmc_Init needs to be called before using other functions of this library.
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.

Example

```
// MMC module connections
sfr sbit Mmc_Chip_Select at RC2_bit;
sfr sbit Mmc_Chip_Select_Direction at TRISC2_bit;
// MMC module connections
...
SPI1_Init();
error = Mmc_Init();  // Init with CS line at RC2_bit
```

Mmc_Read_Sector

Prototype	`unsigned char Mmc_Read_Sector(unsigned long sector, char* dbuff);`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads one sector (512 bytes) from MMC card at sector address `sector`. Read data is stored in the array `data`. Function returns 0 if read was successful, or 1 if an error occurred.
Requires	Library needs to be initialized, see Mmc_Init.
Example	`error = Mmc_Read_Sector(sector, data);`

Mmc_Write_Sector

Prototype	`unsigned char Mmc_Write_Sector(unsigned long sector, char *dbuff);`
Returns	Returns 0 if write was successful; returns 1 if there was an error in sending write command; returns 2 if there was an error in writing.
Description	Function writes 512 bytes of `data` to MMC card at sector address `sector`. Function returns 0 if write was successful, or 1 if there was an error in sending write command, or 2 if there was an error in writing.
Requires	Library needs to be initialized, see Mmc_Init.
Example	`error := Mmc_Write_Sector(sector, data);`

Mmc_Read_Cid

Prototype	`unsigned char Mmc_Read_Cid(char * data_for_registers);`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CID register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see Mmc_Init.
Example	`error = Mmc_Read_Cid(data);`

Mmc_Read_Csd

Prototype	`unsigned char Mmc_Read_Csd(char * data_for_registers);`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CSD register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see Mmc_Init.
Example	`error = Mmc_Read_Csd(data);`

Mmc_Fat_Init

Prototype	`unsigned short Mmc_Fat_Init();`
Returns	- `0` - if MMC/SD card was detected and successfuly 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

```
// MMC module connections
sfr sbit Mmc_Chip_Select at RC2_bit;
sfr sbit Mmc_Chip_Select_Direction at TRISC2_bit;
// MMC module connections

// Initialize SPI1 module and set pointer(s) to SPI1 functions
SPI1_Init_Advanced(MASTER_OSC_DIV64, DATA_SAMPLE_MIDDLE,
CLK_IDLE_LOW, LOW_2_HIGH);

// use fat16 quick format instead of init routine if a formatting
is needed
if (!Mmc_Fat_Init()) {
// reinitialize SPI1 at higher speed
SPI1_Init_Advanced(MASTER_OSC_DIV4, DATA_SAMPLE_MIDDLE,
CLK_IDLE_LOW, LOW_2_HIGH);
...
}
```

Mmc_Fat_QuickFormat

Prototype	`unsigned char Mmc_Fat_QuickFormat(char * mmc_fat_label);`
Returns	- `0` - if MMC/SD card was detected and successfuly 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: - `mmc_fat_label`: volume label (11 characters in length). If less than 11 characters are provided, the label will be padded with spaces. If null string is passed volume will not be labeled **Note**: This routine can be used instead or in conjunction with 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	

```
// Initialize SPI1 module and set pointer(s) to SPI1 functions
SPI1_Init_Advanced(MASTER_OSC_DIV64, DATA_SAMPLE_MIDDLE,
CLK_IDLE_LOW, LOW_2_HIGH);

// Format and initialize MMC/SD card and MMC_FAT16 library glob-
als
if (!Mmc_Fat_QuickFormat(&mmc_fat_label)) {

// Reinitialize the SPI module at higher speed (change primary
prescaler).
SPI1_Init_Advanced(MASTER_OSC_DIV4, DATA_SAMPLE_MIDDLE,
CLK_IDLE_LOW, LOW_2_HIGH);
...
}
```

Mmc_Fat_Assign

<table>
<tr><td>Prototype</td><td><code>unsigned short Mmc_Fat_Assign(char * filename, char file_cre_attr);</code></td></tr>
<tr><td>Returns</td><td>- <code>1</code> - if file already exists or file does not exist but new file is created.
- <code>0</code> - if file does not exist and no new file is created.</td></tr>
<tr><td>Description</td><td>Assigns file for file operations (read, write, delete...). All subsequent file operations will be applied over the 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 no 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 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 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 file does not exist and this flag is set, new file with specified name will be created.</td></tr>
</table>

Note: Long File Names (LFN) are not supported.</td></tr>
<tr><td>Requires</td><td>MMC/SD card and MMC library must be initialized for file operations. See Mmc_Fat_Init.</td></tr>
<tr><td>Example</td><td><code>//Create file with archive attribut if it does not already exists</code>
<code>Mmc_Fat_Assign('MIKROELE.TXT',0xA0);</code></td></tr>
</table>

Mmc_Fat_Reset

Prototype	`void Mmc_Fat_Reset(unsigned long * size);`
Returns	Nothing.
Description	Procedure resets the file pointer (moves it to the start of the file) of the assigned file, so that the file can be read. Parameter `size` stores the size of the assigned file, in bytes.
Requires	The file must be assigned, see Mmc_Fat_Assign.
Example	`Mmc_Fat_Reset(size);`

Mmc_Fat_Rewrite

Prototype	`void Mmc_Fat_Rewrite();`
Returns	Nothing.
Description	Procedure resets the file pointer and clears the assigned file, so that new data can be written into the file.
Requires	The file must be assigned, see Mmc_Fat_Assign.
Example	`Mmc_Fat_Rewrite;`

Mmc_Fat_Append

Prototype	`void Mmc_Fat_Append();`
Returns	Nothing.
Description	The procedure moves the file pointer to the end of the assigned file, so that data can be appended to the file.
Requires	The file must be assigned, see Mmc_Fat_Assign.
Example	`Mmc_Fat_Append;`

Mmc_Fat_Read

Prototype	`void Mmc_Fat_Read(unsigned short *bdata);`
Returns	Nothing.
Description	Procedure reads the byte at which the file pointer points to and stores data into parameter data. The file pointer automatically increments with each call of `Mmc_Fat_Read`.
Requires	The file must be assigned, see Mmc_Fat_Assign. Also, file pointer must be initialized; see Mmc_Fat_Reset.
Example	`Mmc_Fat_Read(mydata);`

Mmc_Fat_Delete

Prototype	`void 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	`void Mmc_Fat_Write(char * fdata, unsigned data_len);`
Returns	Nothing.
Description	Procedure writes a chunk of bytes (`fdata`) to the currently assigned file, at the position of the file pointer.
Requires	The file must be assigned, see Mmc_Fat_Assign. Also, file pointer must be initialized; see Mmc_Fat_Append or Mmc_Fat_Rewrite.
Example	`Mmc_Fat_Write(txt,255);` `Mmc_Fat_Write('Hello world',255);`

Mmc_Fat_Set_File_Date

Prototype	`void Mmc_Fat_Set_File_Date(unsigned int year, unsigned short month, unsigned short day, unsigned short hours, unsigned short mins, unsigned short seconds);`
Returns	Nothing.
Description	Writes system timestamp to a file. Use this routine before each writing to file; otherwise, the file will be appended an unknown timestamp.
Requires	The file must be assigned, see Mmc_Fat_Assign. Also, file pointer must be initialized; see Mmc_Fat_Reset.
Example	`// April 1st 2005, 18:07:00` `Mmc_Fat_Set_File_Date(2005, 4, 1, 18, 7, 0);`

Mmc_Fat_Get_File_Date

Prototype	`void Mmc_Fat_Get_File_Date(unsigned int *year, unsigned short *month, unsigned short *day, unsigned short *hours, unsigned short *mins);`
Returns	Nothing.
Description	Retrieves date and time for the currently selected file. Seconds are not being retrieved since they are written in 2-sec increments.
Requires	The file must be assigned, see Mmc_Fat_Assign.

Example	`// get Date/time of file`
	`unsigned yr;`
	`char mnth, dat, hrs, mins;`
	`...`
	`file_Name = "MYFILEABTXT";`
	`Mmc_Fat_Assign(file_Name);`
	`Mmc_Fat_Get_File_Date(yr, mnth, dat, hrs, mins);`

Mmc_Fat_Get_File_Size

Prototype	`unsigned long Mmc_Fat_Get_File_Size();`
Returns	This function returns size of active file (in bytes).
Description	Retrieves size for currently selected file.
Requires	The file must be assigned, see Mmc_Fat_Assign.

Example	`// get Date/time of file`
	`unsigned yr;`
	`char mnth, dat, hrs, mins;`
	`...`
	`file_name = "MYFILEXXTXT";`
	`Mmc_Fat_Assign(file_name);`
	`mmc_size = Mmc_Fat_Get_File_Size;`

Mmc_Fat_Get_Swap_File

Prototype	`unsigned long Mmc_Fat_Get_Swap_File(unsigned long sectors_cnt, char* filename, char file_attr);`
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 recomended to erase such file if it exists before calling this function. If it is not erased and there is still enough space for new swap file, this function will delete it after allocating new memory space for 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. Swap file can be considered as a "window" on the media where user can freely write/read the data. It's main purpose in mikroC'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 no 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 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.

Description

- `file_attr`: file creation and attributs flags. Each bit corresponds to appropriate file attribut:

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

```
//-------------- Tries to create a swap file, whose size will be at
least 100 sectors.
//If it succeeds, it sends the No. of start sector over UART
void M_Create_Swap_File(){
  size = Mmc_Fat_Get_Swap_File(100);
  if (size <> 0) {
    UART_Write(0xAA);
    UART_Write(Lo(size));
    UART_Write(Hi(size));
    UART_Write(Higher(size));
    UART_Write(Highest(size));
    UART_Write(0xAA);
  }
}
```

Library Example

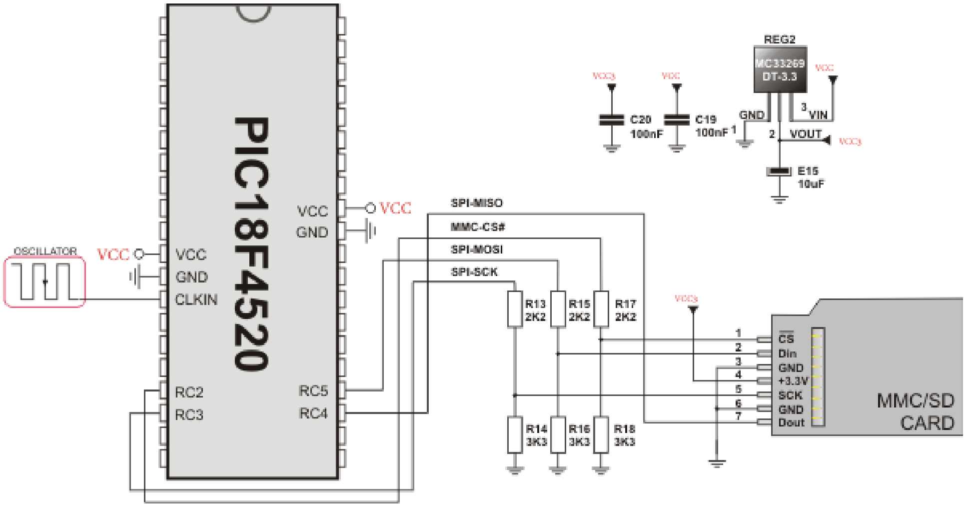
The following example demonstrates MMC library test. Upon flashing, insert a MMC/SD card into the module, when you should receive the "Init-OK" message. Then, you can experiment with MMC read and write functions, and observe the results through the Usart Terminal.

```
// MMC module connections
sbit Mmc_Chip_Select            at RC2_bit;
sbit Mmc_Chip_Select_Direction at TRISC2_bit;
// eof MMC module connections

// Variables for MMC routines
unsigned char SectorData[512]; // Buffer for MMC sector reading/writing
unsigned char data_for_registers[16];// buffer for CID and CSD registers

// UART1 write text and new line (carriage return + line feed)
void UART1_Write_Line(char *uart_text) {
  UART1_Write_Text(uart_text);
  UART1_Write(13);
  UART1_Write(10);
}

// Display byte in hex
void PrintHex(unsigned char i) {
  unsigned char hi,lo;

  hi = i & 0xF0;                 // High nibble
  hi = hi >> 4;
  hi = hi + '0';
  if (hi>'9') hi=hi+7;
  lo = (i & 0x0F) + '0';         // Low nibble
  if (lo>'9') lo=lo+7;

  UART1_Write(hi);
  UART1_Write(lo);
}

void main() {

  const char    FILL_CHAR = 'm';
  unsigned int i, SectorNo;
  char           mmc_error;
  bit            data_ok;

  ADCON1 |= 0x0F;                // Configure AN pins as digital
  CMCON  |= 7;                   // Turn off comparators

  // Initialize UART1 module
```

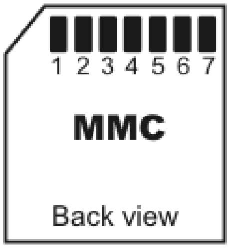
```
  UART1_Init(19200);
  Delay_ms(10);

  UART1_Write_Line("PIC-Started"); // PIC present report

  // Initialize SPI1 module
  SPI1_Init_Advanced(_SPI_MASTER_OSC_DIV64, _SPI_DATA_SAMPLE_MIDDLE,
_SPI_CLK_IDLE_LOW, _SPI_LOW_2_HIGH);

  // initialise a MMC card
  mmc_error = Mmc_Init();
  if(mmc_error == 0)
    UART1_Write_Line("MMC Init-OK"); // If MMC present report
  else
    UART1_Write_Line("MMC Init-error"); // If error report

  // Fill MMC buffer with same characters
  for(i=0; i<=511; i++)
    SectorData[ i]  = FILL_CHAR;

  // Write sector
  mmc_error = Mmc_Write_Sector(SectorNo, SectorData);
  if(mmc_error == 0)
    UART1_Write_Line("Write-OK");
  else  // if there are errors.....
    UART1_Write_Line("Write-Error");

  // Reading of CID register
  mmc_error = Mmc_Read_Cid(data_for_registers);
  if(mmc_error == 0) {
    UART1_Write_Text("CID : ");
    for(i=0; i<=15; i++)
      PrintHex(data_for_registers[ i] );
    UART1_Write_Line("");
  }
  else
    UART1_Write_Line("CID-error");

  // Reading of CSD register
  mmc_error = Mmc_Read_Csd(data_for_registers);
  if(mmc_error == 0) {
    UART1_Write_Text("CSD : ");
    for(i=0; i<=15; i++)
      PrintHex(data_for_registers[ i] );
    UART1_Write_Line("");
  }
  else
    UART1_Write_Line("CSD-error");

  // Read sector
```

```
    mmc_error = Mmc_Read_Sector(SectorNo, SectorData);
    if(mmc_error == 0) {
      UART1_Write_Line("Read-OK");
      // Chech data match
      data_ok = 1;
      for(i=0; i<=511; i++) {
        UART1_Write(SectorData[ i] );
        if (SectorData[ i]  != FILL_CHAR) {
          data_ok = 0;
          break;
        }
      }
      UART1_Write_Line("");
      if (data_ok)
        UART1_Write_Line("Content-OK");
      else
        UART1_Write_Line("Content-Error");
    }
    else  // if there are errors.....
      UART1_Write_Line("Read-Error");

    // Signal test end
    UART1_Write_Line("Test End.");
  }
```

HW Connection



MMC interface



MMC back view

ONEWIRE LIBRARY

The OneWire library provides routines for communication via the Dallas OneWire protocol, for example 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 also has a unique 64-bit registration number (8-bit device type, 48-bit serial number and 8-bit CRC), so multiple slaves can co-exist on the same bus.
Note that oscillator frequency Fosc needs to be at least 4MHz in order to use the routines with Dallas digital thermometers.

Note: This library implements time-based activities, so interrupts need to be disabled when using OneWire library.

Library Routines

- Ow_Reset
- Ow_Read
- Ow_Write

Ow_Reset

Prototype	`unsigned short Ow_Reset(unsigned short *port, unsigned short pin);`
Returns	0 if DS1820 is present, and 1 if not present.
Description	Issues OneWire reset signal for DS1820. Parameters PORT and pin specify the location of DS1820.
Requires	Works with Dallas DS1820 temperature sensor only.
Example	To reset the DS1820 that is connected to the RA5 pin: `Ow_Reset(&PORTA, 5);`

Ow_Read

Prototype	`unsigned short Ow_Read(unsigned short *port, unsigned short pin);`
Returns	Data read from an external device over the OneWire bus.
Description	Reads one byte of data via the OneWire bus.
Requires	Nothing.

Example	`unsigned short tmp;`
	`...`
	`tmp = Ow_Read(&PORTA, 5);`

Ow_Write

Prototype	`void Ow_Write(unsigned short *port, unsigned short pin, unsigned short par);`
Returns	Nothing.
Description	Writes one byte of data (argument `par`) via OneWire bus.
Requires	Nothing.
Example	`Ow_Write(&PORTA, 5, 0xCC);`

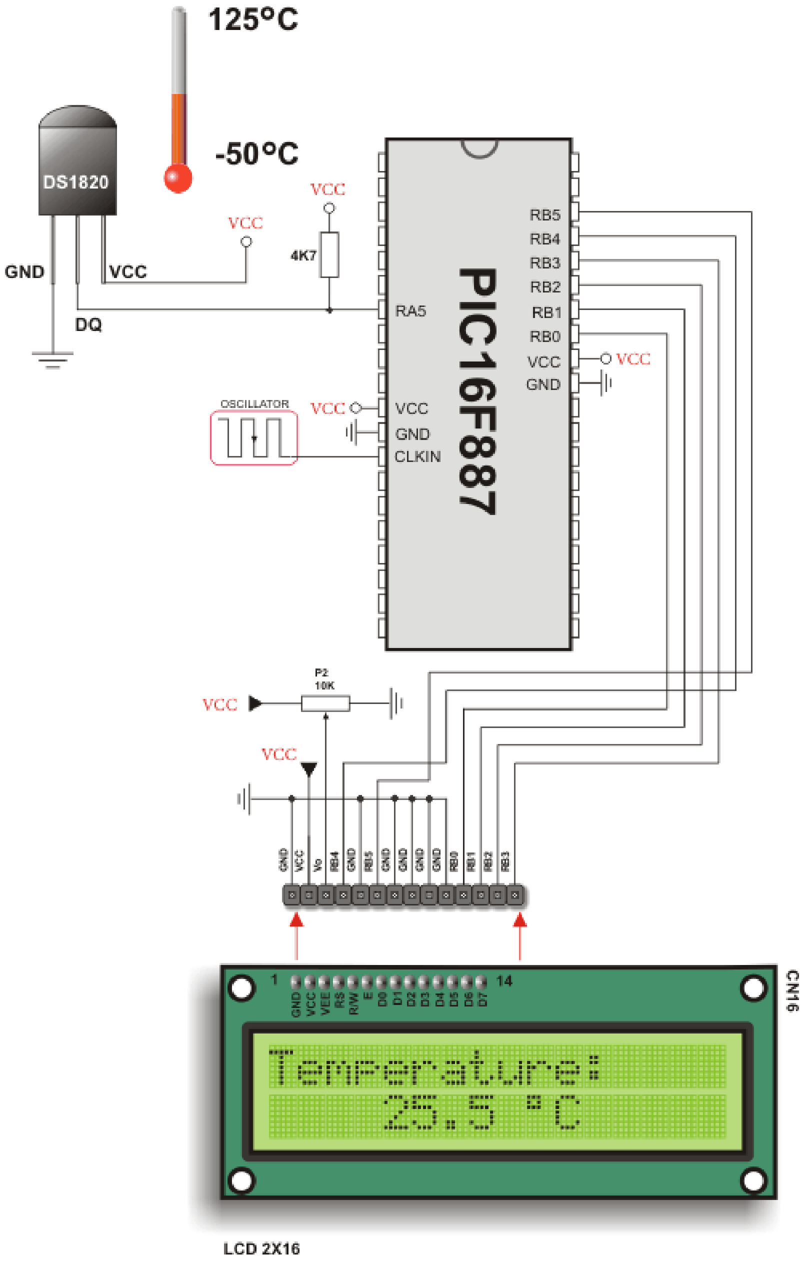
Library Example

This example reads the temperature using DS18x20 connected to pin PORTA.B5. After reset, MCU obtains temperature from the sensor and prints it on the Lcd. Make sure to pull-up PORTA.B5 line and to turn off the PORTA LEDs.

```
// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

//   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 unsigned short TEMP_RESOLUTION = 9;

char *text = "000.0000";
unsigned temp;
void Display_Temperature(unsigned int temp2write) {
  const unsigned short RES_SHIFT = TEMP_RESOLUTION - 8;
  char temp_whole;
  unsigned int temp_fraction;

  // check if temperature is negative
  if (temp2write & 0x8000) {
     text[0] = '-';
     temp2write = ~temp2write + 1;
     }
  // extract temp_whole
  temp_whole = temp2write >> RES_SHIFT;

  // convert temp_whole to characters
  if (temp_whole/100)
     text[0] = temp_whole/100  + 48;
  else
     text[0] = '0';
```

```
  text[ 1] = (temp_whole/10)%10 + 48;     // Extract tens digit
  text[ 2] =  temp_whole%10      + 48;    // Extract ones digit

  // extract temp_fraction and convert it to unsigned int
  temp_fraction  = temp2write << (4-RES_SHIFT);
  temp_fraction &= 0x000F;
  temp_fraction *= 625;
  // convert temp_fraction to characters
  text[ 4]  =  temp_fraction/1000    + 48;   // Extract thousands digit
  text[ 5] = (temp_fraction/100)%10 + 48;   // Extract hundreds digit
  text[ 6] = (temp_fraction/10)%10  + 48;   // Extract tens digit
  text[ 7] =  temp_fraction%10       + 48;   // Extract ones digit
  // print temperature on LCD
  Lcd_Out(2, 5, text);
}
void main() {
  ANSEL  = 0;                  // Configure AN pins as digital I/O
  ANSELH = 0;
  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
  do {
    //--- perform temperature reading
    Ow_Reset(&PORTA, 5);                 // Onewire reset signal
    Ow_Write(&PORTA, 5, 0xCC);           // Issue command SKIP_ROM
    Ow_Write(&PORTA, 5, 0x44);           // Issue command CONVERT_T
    Delay_us(120);

    Ow_Reset(&PORTA, 5);
    Ow_Write(&PORTA, 5, 0xCC);            // Issue command SKIP_ROM
    Ow_Write(&PORTA, 5, 0xBE);     // Issue command READ_SCRATCHPAD

    temp =  Ow_Read(&PORTA, 5);
    temp = (Ow_Read(&PORTE, 5) << 8) + temp;

    //--- Format and display result on Lcd
    Display_Temperature(temp);

    Delay_ms(500);
  } while (1);
}
```

HW Connection



Example of DS1820 connection

PORT EXPANDER LIBRARY

The *mikroC PRO for PIC* provides a library for communication with the Microchip's Port Expander MCP23S17 via SPI interface. Connections of the PIC compliant MCU and MCP23S17 is given on the schematic at the bottom of this page.

Note: Library does not use Port Expander interrupts.
Note: The appropriate hardware SPI module must be initialized before using any of the Port Expander library routines. Refer to SPI Library.

External dependencies of Port Expander Library

The following variables must be defined in all projects using Port Expander Library:	Description:	Example:
`extern sfr sbit SPExpanderRST;`	Reset line.	`SPExpanderCS : sbit at P1.B1;`
`extern sfr sbit SPExpanderCS;`	Chip Select line.	`SPExpanderRST : sbit at P1.B0;`
`extern sfr sbit SPExpanderRST_Direction;`	Direction of the Reset pin.	`sbit SPExpanderRST_Direction at TRISC0_bit;`
`extern sfr sbit SPExpanderCS_Direction;`	Direction of the Chip Select pin.	`sbit SPExpanderCS_Direction at TRISC1_bit`

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	`void Expander_Init(char ModuleAddress);`
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
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

...

ANSEL  = 0;                // Configure AN pins as digital I/O
ANSELH = 0;

// If Port Expander Library uses SPI module
SPI1_Init();               // Initialize SPI module used with 
PortExpander
Expander_Init(0);          // Initialize Port Expander
```

Expander_Read_Byte

Prototype	`char Expander_Read_Byte(char ModuleAddress, char RegAddress);`
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`
	`char read_data;`
	`...`
	`read_data = Expander_Read_Byte(0,1);`

Expander_Write_Byte

Prototype	`void Expander_Write_Byte(char ModuleAddress, char RegAddress, char Data);`
Returns	Nothing.
Description	Routine writes a byte to Port Expander. Parameters: - `ModuleAddress:` Port Expander hardware address, see schematic at the bottom 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,$FF);`

Expander_Read_PortA

Prototype	`char Expander_Read_PortA(char ModuleAddress);`
Returns	Byte read.
Description	The function reads byte from Port Expander's PortA. 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 should be configured as an input. See Expander_Set_DirectionPortA and Expander_Set_DirectionPortAB routines.
Example	

```
// Read a byte from Port Expander's PORTA
char read_data;
...
Expander_Set_DirectionPortA(0,0xFF);  // set expander's porta to
be input
...
read_data = Expander_Read_PortA(0);
```

Expander_Read_PortB

Prototype	`char Expander_Read_PortB(char ModuleAddress);`
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
char read_data;
...
Expander_Set_DirectionPortB(0,0xFF);        // set expander's
portb to be input
...
read_data = Expander_Read_PortB(0);
```

Expander_Read_PortAB

Prototype	`unsigned int Expander_Read_PortAB(char ModuleAddress);`
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
unsigned int read_data;
...
Expander_Set_DirectionPortAB(0,0xFFFF);  // set expander's porta and portb to be input
...
read_data = Expander_Read_PortAB(0);
```

Expander_Write_PortA

Prototype	`void Expander_Write_PortA(char ModuleAddress, char Data_);`
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	`void Expander_Write_PortB(char ModuleAddress, char Data_);`
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	`void Expander_Write_PortAB(char ModuleAddress, unsigned int Data_);`
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	`void Expander_Set_DirectionPortA(char ModuleAddress, char Data_);`
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 configures the corresponding pin as an input. Cleared bit configures the corresponding pin as an 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	`void Expander_Set_DirectionPortB(char ModuleAddress, char Data_);`
Returns	Nothing.
Description	The function sets Port Expander's PortB direction. Parameters: - `ModuleAddress:` Port Expander hardware address, see schematic at the bottom 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 configures the corresponding pin as an input. Cleared bit configures the corresponding pin as an 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	`void Expander_Set_DirectionPortAB(char ModuleAddress, unsigned int Direction);`
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. Each bit corresponds to the appropriate pin of the PortA/PortB register. Set bit configures the corresponding pin as an input. Cleared bit configures the corresponding pin as an 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	`void Expander_Set_PullUpsPortA(char ModuleAddress, char Data_);`
Returns	Nothing.
Description	The function sets Port Expander's PortA pull up/down resistors. Parameters: - `ModuleAddress:` Port Expander hardware address, see schematic at the bottom of this page - `Data:` data for choosing pull up/down resistors configuration. Each bit corresponds 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	`void Expander_Set_PullUpsPortB(char ModuleAddress, char Data_);`
Returns	Nothing.
Description	The function sets Port Expander's PortB pull up/down resistors. Parameters: - `ModuleAddress:` Port Expander hardware address, see schematic at the bottom of this page - `Data:` data for choosing pull up/down resistors configuration. Each bit corresponds 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

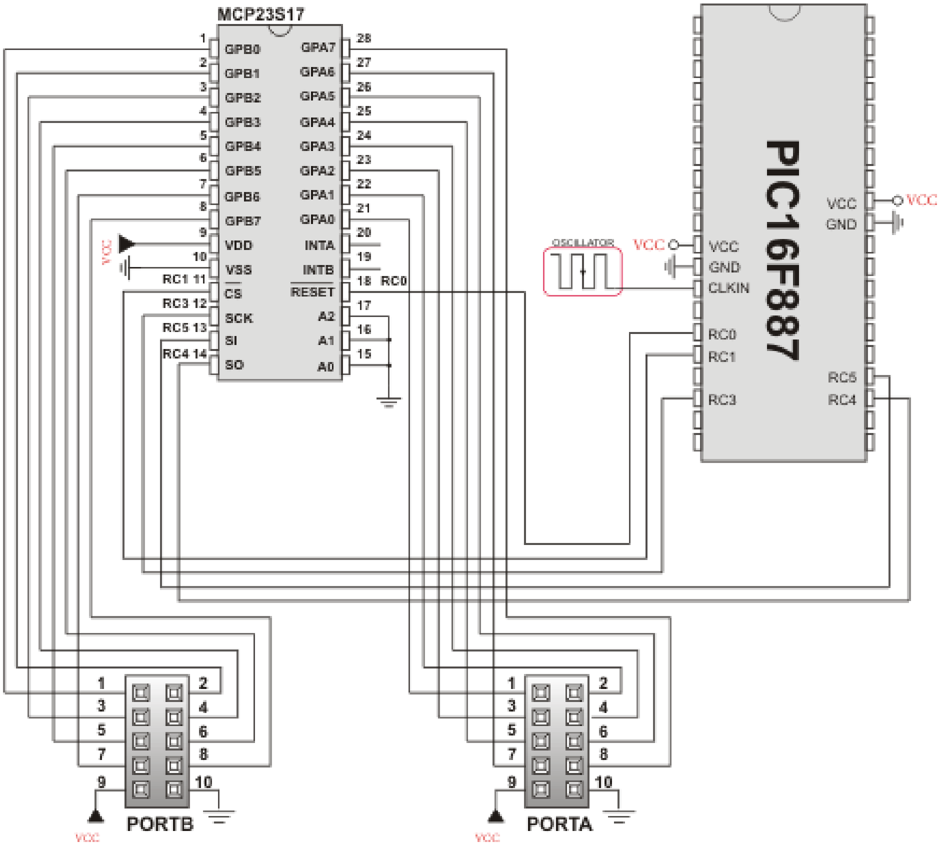
Prototype	`void Expander_Set_PullUpsPortAB(char ModuleAddress, unsigned int PullUps);`
Returns	Nothing.
Description	The function sets Port Expander's PortA and PortB pull up/down resistors. Parameters: - `ModuleAddress:` Port Expander hardware address, see schematic at the bottom of this page - `PullUps:` 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`.

```
// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

unsigned char i = 0;

void main() {
  ANSEL  = 0;                     // Configure AN pins as digital I/O
  ANSELH = 0;
  TRISB = 0;                      // Set PORTB as output
  PORTB  = 0xFF;

  // If Port Expander Library uses SPI1 module
  SPI1_Init();  // Initialize SPI module used with PortExpander

//  If Port Expander Library uses SPI2 module
//  SPI2_Init();  // Initialize SPI module used with PortExpander

  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(1) {                         // Endless loop
    Expander_Write_PortA(0, i++);  // Write i to expander's PORTA
     PORTB = Expander_Read_PortB(0);  // Read expander's PORTB and
write it to LEDs
    Delay_ms(100);
  }

}
```

HW Connection



Port Expander HW connection

PS/2 LIBRARY

The *mikroC PRO for PIC* 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:
`extern sfr sbit PS2_Data;`	PS/2 Data line.	`sbit PS2_Data at RC0_bit`
`extern sfr sbit PS2_Clock;`	PS/2 Clock line.	`sbit PS2_Clock at RC1_bit;`
`extern sfr sbit PS2_Data_Direction;`	Direction of the PS/2 Data pin.	`sbit PS2_Data_Direction at TRISC0_bit;`
`extern sfr sbit PS2_Clock_Direction;`	Direction of the PS/2 Clock pin.	`sbit PS2_Clock_Direction at TRISC1_bit;`

Library Routines

- Ps2_Config
- Ps2_Key_Read

Ps2_Config

Prototype	`void Ps2_Config();`
Returns	Nothing.
Description	Initializes the MCU for work with the PS/2 keyboard.
Requires	Global variables: - `PS2_Data`: Data signal line - `PS2_Clock`: Clock signal line in - `PS2_Data_Direction`: Direction of the Data pin - `PS2_Clock_Direction`: Direction of the Clock pin must be defined before using this function.

Example	`sbit PS2_Data at RC0_bit;`
	`sbit PS2_Clock at RC1_bit;`
	`sbit PS2_Data_Direction at TRISC0_bit;`
	`sbit PS2_Clock_Direction at TRISC1_bit;`
	`...`
	`Ps2_Config();          // Init PS/2 Keyboard`

Ps2_Key_Read

Prototype	`unsigned short Ps2_Key_Read(unsigned short *value, unsigned short *special, unsigned short *pressed);`
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

```
unsigned short keydata = 0, special = 0, down = 0;
...
// Press Enter to continue:
do {
  if (Ps2_Key_Read(&keydata, &special, &down)) {
    if (down && (keydata == 16)) break;
  }
} while (1);
```

Special Function Keys

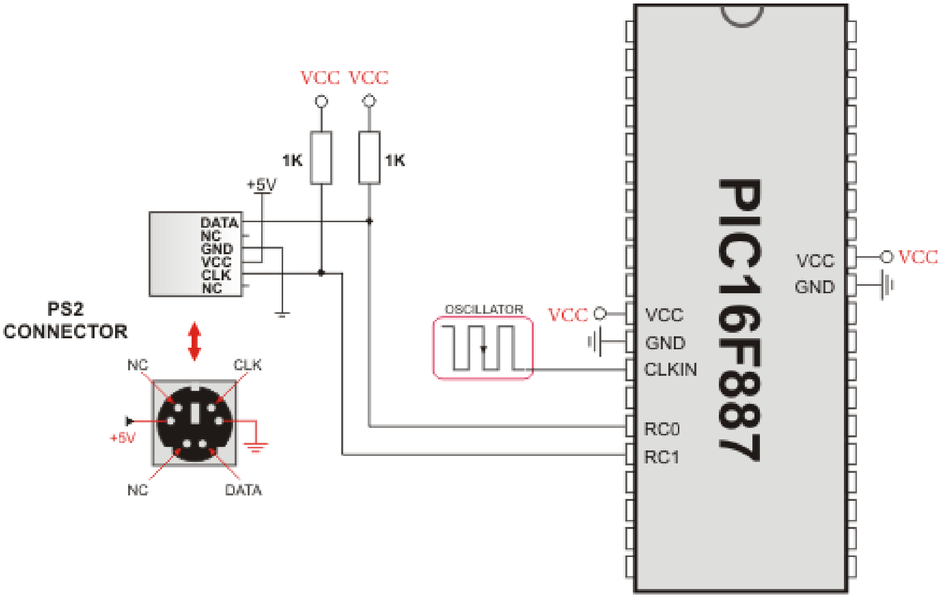
Key	Value returned
F1	1
F2	2
F3	3
F4	4
F5	5
F6	6
F7	7
F8	8
F9	9
F10	10
F11	11
F12	12
Enter	13
Page Up	14
Page Down	15
Backspace	16
Insert	17
Delete	18
Windows	19
Ctrl	20
Shift	21
Alt	22
Print Screen	23
Pause	24
Caps Lock	25
End	26
Home	27
Scroll Lock	28
Num Lock	29
Left Arrow	30
Right Arrow	31
Up Arrow	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.

```
unsigned short keydata = 0, special = 0, down = 0;

sbit PS2_Data            at RC0_bit;
sbit PS2_Clock           at RC1_bit;
sbit PS2_Data_Direction  at TRISC0_bit;
sbit PS2_Clock_Direction at TRISC1_bit;

void main() {

  ANSEL  = 0;              // Configure AN pins as digital I/O
  ANSELH = 0;

  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_Text("Ready");

  do {
    if (Ps2_Key_Read(&keydata, &special, &down)) {
      if (down && (keydata == 16)) {// Backspace
         UART1_Write(0x08);
      }
      else if (down && (keydata == 13)) {// Enter
        UART1_Write('r'); // send carriage return to usart terminal
        //Usart_Write('n');      //  uncomment  this  line  if  usart
terminal also expects line feed
                                // for new line transition
      }
      else if (down && !special && keydata) {
        UART1_Write(keydata);
      }
    }
    Delay_ms(1);                // debounce
  } while (1);
}
```

HW Connection



Example of PS2 keyboard connection

PWM LIBRARY

CCP module is available with a number of PIC MCUs. *mikroC PRO for PIC* provides library which simplifies using PWM HW Module.

Note: Some MCUs have multiple CCP modules. In order to use the desired CCP library routine, simply change the number 1 in the prototype with the appropriate module number, i.e. `PWM2_Start()`;

Library Routines

- PWM1_Init
- PWM1_Set_Duty
- PWM1_Start
- PWM1_Stop

PWM1_Init

Prototype	`void PWM1_Init(long freq);`
Returns	Nothing.
Description	Initializes the PWM module with duty ratio 0. Parameter `freq` is a desired PWM frequency in Hz (refer to device data sheet for correct values in respect with Fosc). This routine needs to be called before using other functions from PWM Library.
Requires	MCU must have CCP module. **Note**: Calculation of the PWM frequency value is carried out by the compiler, as it would produce a relatively large code if performed on the libary level. Therefore, compiler needs to know the value of the parameter in the compile time. That is why this parameter needs to be a constant, and not a variable.
Example	Initialize PWM module at 5KHz: `PWM1_Init(5000);`

PWM1_Set_Duty

Prototype	`void PWM1_Set_Duty(unsigned short duty_ratio);`
Returns	Nothing.
Description	Sets 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	MCU must have CCP module. PWM1_Init must be called before using this routine.
Example	Set duty ratio to 75%: `PWM1_Set_Duty(192);`

PWM1_Start

Prototype	`void PWM1_Start(void);`
Returns	Nothing.
Description	Starts PWM.
Requires	MCU must have CCP module. PWM1_Init must be called before using this routine.
Example	`PWM1_Start();`

PWM1_Stop

Prototype	`void PWM1_Stop(void);`
Returns	Nothing.
Description	Starts PWM.
Requires	MCU must have CCP module. PWM1_Init must be called before using this routine. PWM1_Start should be called before using this routine, otherwise it will have no effect as the PWM module is not running.
Example	`PWM1_Stop();`

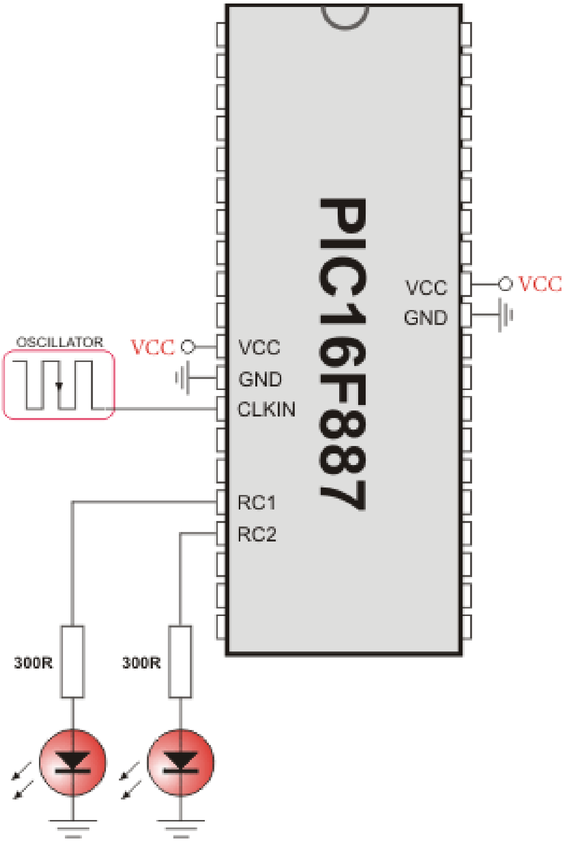
Library Example

The example changes PWM duty ratio on RC1 and RC2 pins continually. If LED is connected to these pins, you can observe the gradual change of emitted light.

```
unsigned short current_duty, old_duty, current_duty1, old_duty1;

void InitMain() {
  ANSEL  = 0;             // Configure AN pins as digital I/O
  ANSELH = 0;
  PORTA = 255;
  TRISA = 255;            // configure PORTA pins as input
  PORTB = 0;              // set PORTB to 0
  TRISB = 0;              // designate PORTB pins as output
  PORTC = 0;              // set PORTC to 0
  TRISC = 0;              // designate PORTC pins as output
  PWM1_Init(5000);        // Initialize PWM1 module at 5KHz
  PWM2_Init(5000);        // Initialize PWM2 module at 5KHz
}

void main() {
  InitMain();
  current_duty  = 16;     // initial value for current_duty
  current_duty1 = 16;     // initial value for current_duty1

  PWM1_Start();                  // start PWM1
  PWM2_Start();                  // start PWM2
  PWM1_Set_Duty(current_duty);   // Set current duty for PWM1
  PWM2_Set_Duty(current_duty1);  // Set current duty for PWM2

  while (1) {                    // endless loop
    if (RA0_bit) {               // button on RA0 pressed
      Delay_ms(40);
      current_duty++;            // increment current_duty
      PWM1_Set_Duty(current_duty);
     }

    if (RA1_bit) {               // button on RA1 pressed
      Delay_ms(40);
      current_duty--;            // decrement current_duty
      PWM1_Set_Duty(current_duty);
     }

    if (RA2_bit) {               // button on RA2 pressed
      Delay_ms(40);
      current_duty1++;           // increment current_duty1
      PWM2_Set_Duty(current_duty1);
     }
```

```
     if (RA3_bit) {                // button on RA3 pressed
       Delay_ms(40);
       current_duty1--;            // decrement current_duty1
       PWM2_Set_Duty(current_duty1);
     }

     Delay_ms(5);                  // slow down change pace a little
  }
}
```

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 *mikroC PRO for PIC* 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).

Note: The library uses the UART module for communication. The user must initialize the appropriate UART module before using the RS-485 Library. For MCUs with two UART modules it is possible to initialize both of them and then switch by using the `UART_Set_Active` function. See the UART Library functions.

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: Since some PIC18 MCUs have multiple UART modules, appropiate UART module must be initialized. Switching between UART modules in the UART library is done by the UART_Set_Active function (UART module has to be previously initialized).

External dependencies of RS-485 Library

The following variable must be defined in all projects using RS-485 Library:	Description:	Example:
`extern sfr sbit RS485_rxtx_pin;`	Control RS-485 Transmit/Receive operation mode	`sbit RS485_rxtx_pin at RC2_bit;`
`extern sfr sbit RS485_rxtx_pin_direction;`	Direction of the RS-485 Transmit/Receive pin	`sbit RS485_rxtx_pin_direction at TRISC2_bit;`

Library Routines

- RS485master_Init
- RS485master_Receive
- RS485master_Send
- RS485slave_Init
- RS485slave_Receive
- RS485slave_Send

RS485Master_Init

Prototype	`void 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 UART1_Init.

Example

```
// RS485 module pinout
sbit RS485_rxtx_pin_direction at RC2_bit;   // transmit/receive
control set to PORTC.B2

// Pin direction
sbit RS485_rxtx_pin_direction at TRISC2_bit;    // RxTx pin direc-
tion set as output

...
UART1_Init(9600);        // initialize UART module
RS485Master_Init();      // intialize MCU as a Master for RS-485
communication
```

RS485Master_Receive

Prototype	`void RS485Master_Receive(char *data_buffer);`
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	`char msg[ 8] ;`
	`...`
	`RS485Master_Receive(msg);`

RS485Master_Send

Prototype	`void RS485Master_Send(char *data_buffer, char datalen, char Slave_address);`
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	`char msg[ 8] ;`
	`...`
	`// send 3 bytes of data to Slave with address 0x12`
	`RS485Master_Send(msg, 3, 0x12);`

RS485slave_Init

Prototype	`void RS485Slave_Init(char Slave_address);`
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 UART1_Init.

Example	`// RS485 module pinout`
	`sbit RS485_rxtx_pin at RC2_bit;      // transmit/receive control`
	`set to PORTC.B2`
	
	`// Pin direction`
	`sbit RS485_rxtx_pin_direction at TRISC2_bit;    // RxTx pin direc-`
	`tion set as output`
	
	`...`
	`UART1_Init(9600);                       // initialize UART module`
	`RS485Slave_Init(160);                    // intialize MCU as a Slave`
	`for RS-485 communication with address 160`

RS485slave_Receive

Prototype	`void RS485Slave_Receive(char *data_buffer);`
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	`char msg[ 8] ;`
	`...`
	`RS485Slave_Read(msg);`

RS485slave_Send

Prototype	`void RS485Slave_Send(char *data_buffer, char datalen);`
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	`char msg[ 8] ;`
	`...`
	`// 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 PORTD. Slave displays received data on PORTB, while error on receive (0xAA) is displayed on PORTD. Hardware configurations in this example are made for the EasyPIC5 board and 16F887.

RS485 Master code:

```
char dat[10];              // buffer for receving/sending messages
char i,j;

sbit  rs485_rxtx_pin  at RC2_bit;          // set transcieve pin
sbit  rs485_rxtx_pin_direction at TRISC2_bit;   // set transcieve pin
direction

// Interrupt routine
void interrupt() {
  RS485Master_Receive(dat);
}

void main(){
  long cnt = 0;

  ANSEL  = 0;              // Configure AN pins as digital I/O
  ANSELH = 0;

  PORTB  = 0;
  PORTD  = 0;
  TRISB  = 0;
  TRISD  = 0;


  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);


  PIE1.RCIE = 1;           // enable interrupt on UART1 receive
  PIE2.TXIE = 0;           // disable interrupt on UART1 transmit
  INTCON.PEIE = 1;         // enable peripheral interrupts
  INTCON.GIE = 1;          // enable all interrupts

  while (1){
                           // upon completed valid message receiving
                           //   data[4] is set to 255
    cnt++;
```

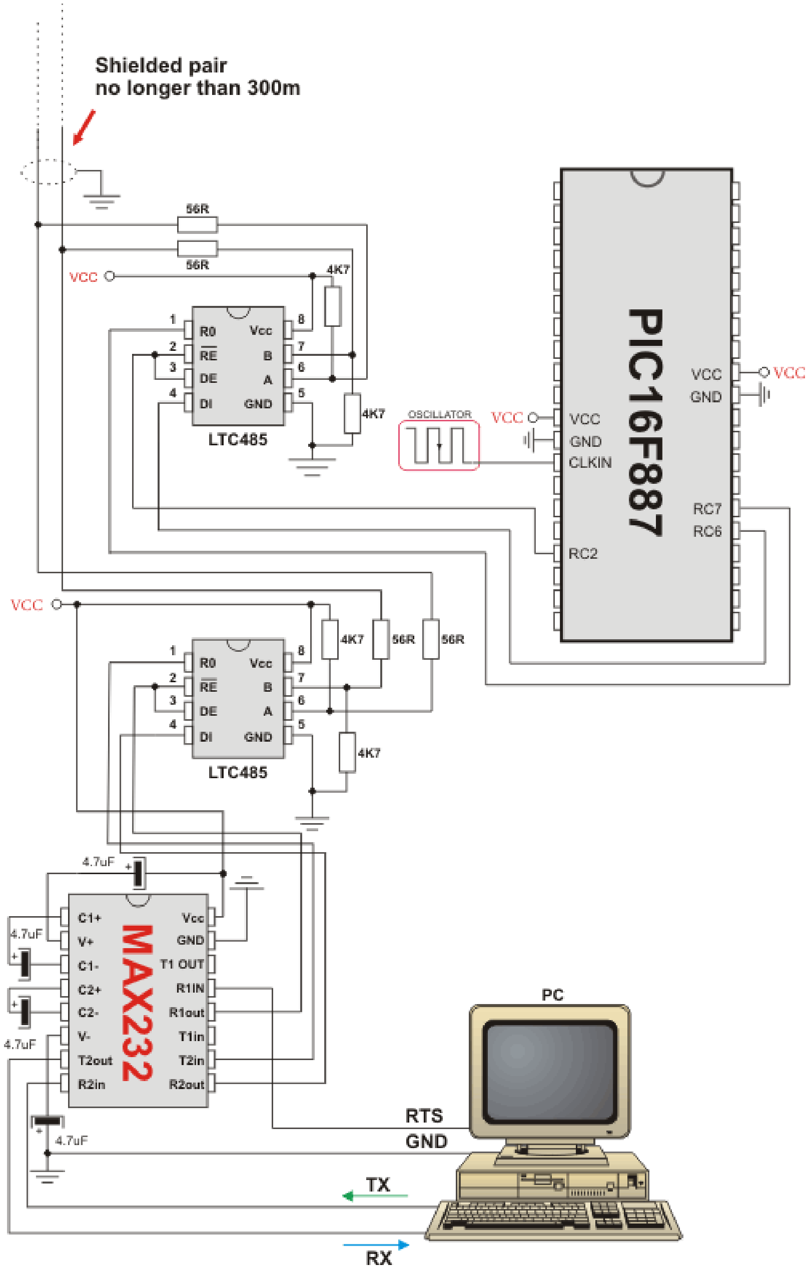
```
      if (dat[5])  {          // if an error detected, signal it
        PORTD = 0xAA;         //   by setting portd to 0xAA
      }
      if (dat[4]) {           // if message received successfully
        cnt = 0;
        dat[4] = 0;           // clear message received flag
        j = dat[3];
        for (i = 1; i <= dat[3]; i++) {  // show data on PORTB
          PORTB = dat[i-1];
        }                                // increment received dat[0]
        dat[0] = dat[0]+1;              // send back to master
        Delay_ms(1);
        RS485Master_Send(dat,1,160);

      }
     if (cnt > 100000) {
        PORTD ++;
        cnt = 0;
        RS485Master_Send(dat,1,160);
        if (PORTD > 10)                 // if sending failed 10 times
           RS485Master_Send(dat,1,50);   // send message on broadcast 
address
       }
    }
                          //   function to be properly linked.
}
```

RS485 Slave code:

```
char dat[9];              // buffer for receving/sending messages
char i,j;

sbit  rs485_rxtx_pin at RC2_bit;             // set transcieve pin
sbit  rs485_rxtx_pin_direction at TRISC2_bit; // set transcieve pin 
direction

// Interrupt routine
void interrupt() {
 RS485Slave_Receive(dat);
}

void main() {
  ANSEL  = 0;             // Configure AN pins as digital I/O
  ANSELH = 0;

  PORTB = 0;
  PORTD = 0;
  TRISB = 0;
  TRISD = 0;
```

```
  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

  PIE1.RCIE = 1;             // enable interrupt on UART1 receive
  PIE2.TXIE = 0;            // disable interrupt on UART1 transmit
  INTCON.PEIE = 1;          // enable peripheral interrupts
  INTCON.GIE = 1;           // enable all interrupts

  while (1) {
    if (dat[ 5])  {         // if an error detected, signal it by
      PORTD = 0xAA;         //   setting portd to 0xAA
      dat[ 5]  = 0;
    }
    if (dat[ 4]) {         // upon completed valid message receive
      dat[ 4]  = 0;        //   data[ 4]  is set to 0xFF
      j = dat[ 3];
      for (i = 1; i <= dat[ 3];i++){
        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
    }
  }
}
```

HW Connection



Example of interfacing PC to 8051 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 ^ address;
crc_send ^= data[ 0] ;    // if exists
crc_send ^= data[ 1] ;    // if exists
crc_send ^= data[ 2] ;    // if exists
crc_send = ~crc_send;
if ((crc_send == START_BYTE) || (crc_send == STOP_BYTE))
   crc_send++;

NOTE: DATALEN<4..0> can not take the START_BYTE<4..0> or
STOP_BYTE<4..0> values.
```

SOFTWARE I²C LIBRARY

The *mikroC PRO for PIC* 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 the 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 dependecies of Soft_I2C Library

The following variables must be defined in all projects using Software I^{2}C Library:	Description:	Example:
`extern sbit Soft_I2C_Scl;`	Soft I^{2}C Clock line.	`sbit Soft_I2C_Scl at RC3_bit;`
`extern sbit Soft_I2C_Sda;`	Soft I^{2}C Data line.	`sbit Soft_I2C_Sda at RC4_bit;`
`extern sbit Soft_I2C_Scl_Direction;`	Direction of the Soft I^{2}C Clock pin.	`sbit Soft_I2C_Scl_Direction at TRISC3_bit;`
`extern sbit Soft_I2C_Sda_Direction;`	Direction of the Soft I^{2}C Data pin.	`sbit Soft_I2C_Sda_Direction at TRISC4_bit;`

Library Routines

- Soft_I2C_Init
- Soft_I2C_Start
- Soft_I2C_Read
- Soft_I2C_Write
- Soft_I2C_Stop
- Soft_I2C_Break

Soft_I2C_Init

Prototype	`void Soft_I2C_Init();`
Returns	Nothing.
Description	Configures the software I¸C module.
Requires	Global variables: - `Soft_I2C_Scl:` Soft I¸C clock line - `Soft_I2C_Sda:` Soft I¸C data line - `Soft_I2C_Scl_Pin_Direction:` Direction of the Soft I¸C clock pin - `Soft_I2C_Sda_Pin_Direction:` Direction of the Soft I¸C data pin must be defined before using this function.

Example

```
// Software I2C connections
sbit Soft_I2C_Scl           at RC3_bit;
sbit Soft_I2C_Sda           at RC4_bit;
sbit Soft_I2C_Scl_Direction at TRISC3_bit;
sbit Soft_I2C_Sda_Direction at TRISC4_bit;
// End Software I2C connections

...
Soft_I2C_Init();
```

Soft_I2C_Start

Prototype	`void Soft_I2C_Start(void);`
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	`unsigned short Soft_I2C_Read(unsigned int ack);`
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^2C 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	`unsigned short take;`
	`...`
	`// Read data and send the not_acknowledge signal`
	`take = Soft_I2C_Read(0);`

Soft_I2C_Write

Prototype	`unsigned short Soft_I2C_Write(unsigned short Data_);`
Returns	- `0` if there were no errors. - `1` if write collision was detected on the I^2C bus.
Description	Sends data byte via the I^2C bus. Parameters: - `Data`: data to be sent
Requires	Soft I^2C 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	`unsigned short data, error;`
	`...`
	`error = Soft_I2C_Write(data);`
	`error = Soft_I2C_Write(0xA3);`

Soft_I2C_Stop

Prototype	`void Soft_I2C_Stop(void);`
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

Prototype	`void Soft_I2C_Break(void);`
Returns	Nothing.
Description	All Software I^{2}C Library functions can block the program flow (see note at the top of this page). Calling this routine from interrupt will unblock the program execution. This mechanism is similar to WDT. **Note**: Interrupts should be disabled before using Software I^{2}C routines again (see note at the top of this page).
Requires	Nothing.

Example

```
// Software I2C connections
sbit Soft_I2C_Scl           at RC0_bit;
sbit Soft_I2C_Sda           at RC1_bit;
sbit Soft_I2C_Scl_Direction at TRISC0_bit;
sbit Soft_I2C_Sda_Direction at TRISC1_bit;
// End Software I2C connections

char counter = 0;

void interrupt {

  if (INTCON.T0IF) {
    if (counter >= 20) {
      Soft_I2C_Break();
      counter = 0;              // reset counter
    }
    else
      counter++;                // increment counter

    INTCON.T0IF = 0;  // Clear Timer0 overflow interrupt flag
```

Example

```
    }
}

void main() {

  OPTION_REG = 0x04;   // TMR0 prescaler set to 1:32

  ...

  // try Soft_I2C_Init with blocking prevention mechanism
  INTCON.GIE = 1;      // Global interrupt enable
  INTCON.T0IE = 1;    // Enable Timer0 overflow interrupt
  Soft_I2C_Init();
  INTCON.GIE = 0;    // Global interrupt disable

  ...

}
```

Library Example

The example demonstrates Software I˛C Library routines usage. The PIC 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.

```
char seconds, minutes, hours, day, month, year;    // Global date/time
variables

// Software I2C connections
sbit Soft_I2C_Scl            at RC3_bit;
sbit Soft_I2C_Sda            at RC4_bit;
sbit Soft_I2C_Scl_Direction at TRISC3_bit;
sbit Soft_I2C_Sda_Direction at TRISC4_bit;
// End Software I2C connections

// LCD module connections
sbit LCD_RS at RB4_bit;
sbit LCD_EN at RB5_bit;
sbit LCD_D4 at RB0_bit;
sbit LCD_D5 at RB1_bit;
sbit LCD_D6 at RB2_bit;
sbit LCD_D7 at RB3_bit;

sbit LCD_RS_Direction at TRISB4_bit;
sbit LCD_EN_Direction at TRISB5_bit;
sbit LCD_D4_Direction at TRISB0_bit;
```

```
sbit LCD_D5_Direction at TRISB1_bit;
sbit LCD_D6_Direction at TRISB2_bit;
sbit LCD_D7_Direction at TRISB3_bit;
// End LCD module connections

//--------------------- Reads time and date information from RTC
(PCF8583)
void Read_Time() {

  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

}

//-------------------- Formats date and time
void Transform_Time() {
  seconds  =  ((seconds & 0xF0) >> 4)*10 + (seconds & 0x0F);  //
Transform seconds
  minutes  =  ((minutes & 0xF0) >> 4)*10 + (minutes & 0x0F);  //
Transform months
  hours    =  ((hours & 0xF0)  >> 4)*10  + (hours & 0x0F);    //
Transform hours
  year     =   (day & 0xC0) >> 6;                             //
Transform year
  day      =  ((day & 0x30) >> 4)*10    + (day & 0x0F);       //
Transform day
  month    =  ((month & 0x10)  >> 4)*10 + (month & 0x0F);     //
Transform month
}

//-------------------- Output values to LCD
void Display_Time() {

    Lcd_Chr(1, 6, (day / 10)    + 48);   // Print tens digit of day
variable
    Lcd_Chr(1, 7, (day % 10)    + 48); // Print oness digit of day
variable
   Lcd_Chr(1, 9, (month / 10) + 48);
   Lcd_Chr(1,10, (month % 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 % 10)   + 48);
   Lcd_Chr(2, 9, (minutes / 10) + 48);
   Lcd_Chr(2,10, (minutes % 10) + 48);
   Lcd_Chr(2,12, (seconds / 10) + 48);
   Lcd_Chr(2,13, (seconds % 10) + 48);
}


//------------------ Performs project-wide init
void Init_Main() {

  TRISB = 0;
  PORTB = 0xFF;
  TRISB = 0xff;
  ANSEL  = 0;              // Configure AN pins as digital I/O
  ANSELH = 0;
  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");
}

//----------------- Main procedure
void main() {

  Delay_ms(2000);

  Init_Main();                   // Perform initialization

  while (1) {                    // Endless loop
    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
  }
}
```

SOFTWARE SPI LIBRARY

The *mikroC PRO for PIC* 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:
`extern sfr sbit SoftSpi_SDI;`	Data In line.	`sbit SoftSpi_SDI at RC4_bit;`
`extern sfr sbit SoftSpi_SDO;`	Data Out line.	`sbit SoftSpi_SDO at RC5_bit;`
`extern sfr sbit SoftSpi_CLK;`	Clock line.	`sbit SoftSpi_CLK at RC3_bit;`
`extern sfr sbit SoftSpi_SDI_Direction;`	Direction of the Data In pin.	`sbit SoftSpi_SDI_Direction at TRISC4_bit;`
`extern sfr sbit SoftSpi_SDO_Direction;`	Direction of the Data Out pin	`sbit SoftSpi_SDO_Direction at TRISC5_bit;`
`extern sfr sbit SoftSpi_CLK_Direction;`	Direction of the Clock pin.	`sbit SoftSpi_CLK_Direction at TRISC3_bit;`

Library Routines

- Soft_Spi_Init
- Soft_Spi_Read
- Soft_Spi_Write

Soft_Spi_Init

Prototype	`void 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	

```
// Software SPI module connections
sbit Chip_Select at RC0_bit;
sbit SoftSpi_SDI at RC4_bit;
sbit SoftSpi_SDO at RC5_bit;
sbit SoftSpi_CLK at RC3_bit;

sbit Chip_Select_Direction at TRISC0_bit;
sbit SoftSpi_SDI_Direction at TRISC4_bit;
sbit SoftSpi_SDO_Direction at TRISC5_bit;
sbit SoftSpi_CLK_Direction at TRISC3_bit;
// End Software SPI module connections
...
Soft_SPI_Init(); // Init Soft_SPI
```

Soft_Spi_Read

Prototype	`unsigned short Soft_SPI_Read(char sdata);`
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	`unsigned short data_read;`
	`char data_send;`
	`...`
	`// Read a byte and assign it to data_read variable`
	`// (data_send byte will be sent via SPI during the Read opera-`
	`tion)`
	`data_read = Soft_SPI_Read(data_send);`

Soft_SPI_Write

Prototype	`void Soft_SPI_Write(char sdata);`
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.

```
// DAC module connections
sbit Chip_Select at RC0_bit;
sbit SoftSpi_CLK at RC3_bit;
sbit SoftSpi_SDI at RC4_bit;
sbit SoftSpi_SDO at RC5_bit;

sbit Chip_Select_Direction at TRISC0_bit;
sbit SoftSpi_CLK_Direction at TRISC3_bit;
sbit SoftSpi_SDI_Direction at TRISC4_bit;
sbit SoftSpi_SDO_Direction at TRISC5_bit;
// End DAC module connections

unsigned int value;

void InitMain() {
  TRISB0_bit = 1;                 // Set RA0 pin as input
  TRISB1_bit = 1;                 // Set RA1 pin as input
  Chip_Select = 1;                // Deselect DAC
  Chip_Select_Direction = 0;      // Set CS# pin as Output
  Soft_SPI_Init();                // Initialize Soft_SPI
}

// DAC increments (0..4095) --> output voltage (0..Vref)
void DAC_Output(unsigned int valueDAC) {
  char temp;

  Chip_Select = 0;                // Select DAC chip

  // Send High Byte
 temp = (valueDAC >> 8) & 0x0F;// Store valueDAC[11..8] to temp[3..0]
  temp |= 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
}

void main() {

  ANSEL  = 0;               // turn off analog inputs
```

```
    ANSELH = 0;

    InitMain();               // Perform main initialization

    value = 2048;             // When program starts, DAC gives
                              //   the output in the mid-range

    while (1) {                            // Endless loop

      if ((RA0_bit) && (value < 4095)) {  // If RA0 button is pressed
        value++;                           //   increment value
        }
      else {
        if ((RA1_bit) && (value > 0)) {   // If RA1 button is pressed
          value--;                         //   decrement value
          }
        }

      DAC_Output(value);      // Send value to DAC chip
      Delay_ms(1);            // Slow down key repeat pace
    }
  }
```

SOFTWARE UART LIBRARY

The *mikroC PRO for PIC* 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.

Library Routines

- Soft_Uart_Init
- Soft_Uart_Read
- Soft_Uart_Write
- Soft_Uart_Break

Soft_UART_Init

Prototype	`char Soft_UART_Init(char *port, char rx_pin, char tx_pin, unsigned long baud_rate, char inverted);`
Returns	`- 2` - error, requested baud rate is too low `- 1` - error, requested baud rate is too high `- 0` - successful initialization
Description	Configures and initializes the software UART module. Parameters: - `port`: port to be used. - `rx_pin`: sets rx_pin to be used. - `tx_pin`: sets tx_pin to be used. - `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	Nothing.

Example	This will initialize software UART and establish the communication at 9600 bps:
	
	`char error;`
	`...`
	`error = Soft_UART_Init(&PORTC, 7, 6, 14400, 0);    // Initialize Soft UART at 9600 bps`

Soft_UART_Read

Prototype	`char Soft_UART_Read(char * error);`
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	

```
char data, error;
...
// wait until data is received
do
  data = Soft_UART_Read(&error);
while (error);

// Now we can work with data:
if (data) {...}
```

Soft_UART_Write

Prototype	`void Soft_UART_Write(char udata);`
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

```
char some_byte = 0x0A;
...
// Write a byte via Soft Uart
Soft_UART_Write(some_byte);
```

Soft_Uart_Break

Prototype	`void Soft_UART_Break();`
Returns	Nothing.
Description	Soft_UART_Read is blocking routine and it can block the program flow. Calling this routine from the interrupt will 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

```
char data1, error, counter = 0;

void interrupt() {

  if (INTCON.T0IF) {
    if (counter >= 20) {
      Soft_UART_Break();
      counter = 0;               // reset counter
    }
    else
      counter++;                // increment counter

    INTCON.T0IF = 0;           // Clear Timer0 overflow interrupt flag

    }
}
```

Example

```
void main() {

  OPTION_REG = 0x04;             // TMR0 prescaler set to 1:32

  ...

  if (Soft_UART_Init(&PORTC, 7, 6, 9600, 0) = 0)
    Soft_UART_Write(0x55);

  ...

  // try Soft_UART_Read with blocking prevention mechanism
  INTCON.GIE = 1;            // Global interrupt enable
  INTCON.T0IE = 1;           // Enable Timer0 overflow interrupt
  data1 = Soft_UART_Read(&error);
  INTCON.GIE = 0;            // Global interrupt disable

}
```

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 *mikroC PRO for PIC* USART Terminal Tool.

```
char i, error, byte_read;         // Auxiliary variables

void main(){

  ANSEL  = 0;                      // Configure AN pins as digital I/O
  ANSELH = 0;

  TRISB = 0x00;         // Set PORTB as output (error signalization)
  PORTB = 0;                       // No error

  error = Soft_UART_Init(&PORTC, 7, 6, 14400, 0); // Initialize Soft
UART at 9600 bps
  if (error > 0) {
    PORTB = error;                 // Signalize Init error
    while(1);                      // Stop program
  }
  Delay_ms(100);

  for (i = 'z'; i >= 'A'; i--) { // Send bytes from 'z' downto 'A'
    Soft_UART_Write(i);
    Delay_ms(100);
  }

  while(1) {                                    // Endless loop
    byte_read = Soft_UART_Read(&error);         //  Read  byte,  then
test error flag
    if (error)                            // 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
    }
}
```

SOUND LIBRARY

The *mikroC PRO for PIC* 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).

Library Routines

- Sound_Init
- Sound_Play

Sound_Init

Prototype	`void Sound_Init(char *snd_port, char snd_pin);`
Returns	Nothing.
Description	Configures the appropriate MCU pin for sound generation. Parameters: - `snd_port`: sound output port address - `snd_pin`: sound output pin
Requires	Nothing.
Example	`// Initialize the pin RD3 for playing sound` `Sound_Init(&PORTD, 3);`

Sound_Play

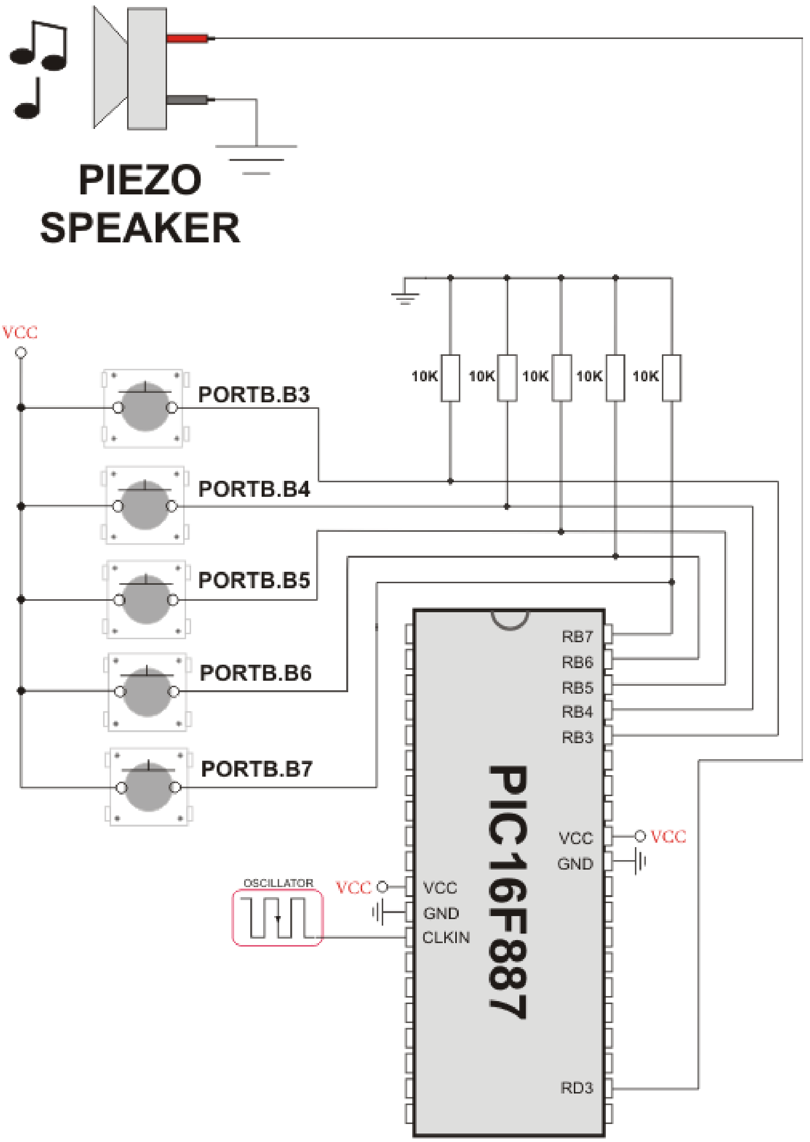
Prototype	`void Sound_Play(unsigned freq_in_hz, unsigned duration_ms);`
Returns	Nothing.
Description	Generates the square wave signal on the appropriate pin. Parameters: - `freq_in_Hz:` signal frequency in Hertz (Hz) - `duration_ms:` signal duration in miliseconds (ms) **Note**: frequency range is limited by Delay_Cyc parameter. Maximum frequency that can be produced by this function is `Freq_max = Fosc/(80*3)`. Minimum frequency is `Freq_min = Fosc/(80*255)`. Generated frequency may differ from the `freq_in_hz` parameter due to integer arithmetics.
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 play-
ing tones on a piezo speaker.


void Tone1() {
  Sound_Play(659, 250);    // Frequency = 659Hz, duration = 250ms
}

void Tone2() {
  Sound_Play(698, 250);    // Frequency = 698Hz, duration = 250ms
}

void Tone3() {
  Sound_Play(784, 250);    // Frequency = 784Hz, duration = 250ms
}

void Melody() {               // Plays the melody "Yellow house"
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3();
  Tone1(); Tone2(); Tone3(); Tone3();
  Tone1(); Tone2(); Tone3();
  Tone3(); Tone3(); Tone2(); Tone2(); Tone1();
```

```
}

void ToneA() {
  Sound_Play( 880, 50);
}
void ToneC() {
  Sound_Play(1046, 50);
}
void ToneE() {
  Sound_Play(1318, 50);
}

void Melody2() {
  unsigned short i;
  for (i = 9; i > 0; i--) {
    ToneA(); ToneC(); ToneE();
  }
}

void main() {
  ANSEL  = 0;                        // Configure AN pins as digital I/O
  ANSELH = 0;
  TRISB  = 0xF8;                     // Configure RB7..RB3 as input
  TRISD  = 0xF7;                     // Configure RD3 as output

  Sound_Init(&PORTD, 3);
  Sound_Play(1000, 1000);

  while (1) {
    if (Button(&PORTB,7,1,1))    // RB7 plays Tone1
      Tone1();
    while (PORTB & 0x80);        // Wait for button to be released

    if (Button(&PORTB,6,1,1))    // RB6 plays Tone2
      Tone2();
    while (PORTB & 0x40);        // Wait for button to be released

    if (Button(&PORTB,5,1,1))    // RB5 plays Tone3
      Tone3();
    while (PORTB & 0x20);        // Wait for button to be released

    if (Button(&PORTB,4,1,1))    // RB4 plays Melody2
      Melody2();
    while (PORTB & 0x10);        // Wait for button to be released

    if (Button(&PORTB,3,1,1))    // RB3 plays Melody
      Melody();
    while (PORTB & 0x08);        // Wait for button to be released
  }
}
```

HW Connection



Example of Sound Library sonnection

SPI LIBRARY

SPI module is available with a number of PIC MCU models. *mikroC PRO for PIC* provides a library for initializing Slave mode and comfortable work with Master mode. PIC can easily communicate with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc. You need PIC MCU with hardware integrated SPI (for example, PIC16F877).

Note: Some PIC18 MCUs have multiple SPI modules. Switching between the SPI modules in the SPI library is done by the SPI_Set_Active function (SPI module has to be previously initialized).

Note: In order to use the desired SPI library routine, simply change the number **1** in the prototype with the appropriate module number, i.e. SPI2_Init();

Library Routines

- Spi1_Init
- Spi1_Init_Advanced
- Spi1_Read
- Spi1_Write
- Spi_Set_Active

Spi_Init

Prototype	`void SPI1_Init(void);`
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	You need PIC MCU with hardware integrated SPI.
Example	`SPI1_Init();  // Initialize the SPI module with default settings`

Spi1_Init_Advanced

<table>
<tr><td>Prototype</td><td><code>void SPI1_Init_Advanced(unsigned short master_slav, unsigned short data_sample, unsigned short clock_idle, unsigned short transmit_edge);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</td><td>Configures and initializes SPI. SPI1_Init or SPI1_Init_Advanced needs to be called before using other functions of SPI Library.

Parameters <code>mode,</code> <code>data_sample</code> and <code>clock_idle</code> configure the SPI module, and can have the following values:
<table>
<tr><th>Description</th><th>Predefined library const</th></tr>
<tr><td colspan="2">SPI work mode:</td></tr>
<tr><td><code>Master clock = Fosc/4</code></td><td><code>_SPI_MASTER_OSC_DIV4</code></td></tr>
<tr><td><code>Master clock = Fosc/16</code></td><td><code>_SPI_MASTER_OSC_DIV16</code></td></tr>
<tr><td><code>Master clock = Fosc/64</code></td><td><code>_SPI_MASTER_OSC_DIV64</code></td></tr>
<tr><td><code>Master clock source TMR2</code></td><td><code>_SPI_MASTER_TMR2</code></td></tr>
<tr><td><code>Slave select enabled</code></td><td><code>_SPI_SLAVE_SS_ENABLE</code></td></tr>
<tr><td><code>Slave select disabled</code></td><td><code>_SPI_SLAVE_SS_DIS</code></td></tr>
<tr><td colspan="2">Data sampling interval:</td></tr>
<tr><td><code>Input data sampled in middle of interval</code></td><td><code>_SPI_DATA_SAMPLE_MIDDLE</code></td></tr>
<tr><td><code>Input data sampled at the end of interval</code></td><td><code>_SPI_DATA_SAMPLE_END</code></td></tr>
<tr><td colspan="2">SPI clock idle state:</td></tr>
<tr><td><code>Clock idle HIGH</code></td><td><code>_SPI_CLK_IDLE_HIGH</code></td></tr>
<tr><td><code>Clock idle LOW</code></td><td><code>_SPI_CLK_IDLE_LOW</code></td></tr>
<tr><td colspan="2">Transmit edge:</td></tr>
<tr><td><code>Data transmit on low to high edge</code></td><td><code>_SPI_LOW_2_HIGH</code></td></tr>
<tr><td><code>Data transmit on high to low edge</code></td><td><code>_SPI_HIGH_2_LOW</code></td></tr>
</table></td></tr>
<tr><td>Requires</td><td>You need PIC MCU with hardware integrated SPI.</td></tr>
<tr><td>Example</td><td><code>// Set SPI1 module to master mode, clock = Fosc/4, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge:
SPI1_Init_Advanced(_SPI_MASTER_OSC_DIV4, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_LOW, _SPI_LOW_2_HIGH);</code></td></tr>
</table>

Spi1_Read

Prototype	`unsigned short SPI1_Read(unsigned short buffer);`
Returns	Returns the received data.
Description	Reads one byte from the SPI bus. Parameters: - `buffer`: dummy data for clock generation (see device Datasheet for SPI modules implementation details)
Requires	You need PIC MCU with hardware integrated SPI. SPI must be initialized and communication established before using this function. See SPI1_Init_Advanced or SPI1_Init.

Example	`short take, buffer;`
	`...`
	`take = SPI1_Read(buffer);`

Spi1_Write

Prototype	`void SPI1_Write(unsigned short data_);`
Returns	Nothing.
Description	Writes byte via the SPI bus. Parameters: - `wrdata:` data to be sent
Requires	You need PIC MCU with hardware integrated SPI. SPI must be initialized and communication established before using this function. See SPI1_Init_Advanced or SPI1_Init.
Example	`SPI1_Write(1);`

SPI_Set_Active

Prototype	`void SPI_Set_Active(char (*read_ptr)(char))`
Returns	Nothing.
Description	Sets the active SPI module which will be used by the SPI routines. Parameters: - `read_ptr`: SPI1_Read handler
Requires	Routine is available only for MCUs with two SPI modules. Used SPI module must be initialized before using this function. See the SPI1_Init, SPI1_Init_Advanced
Example	`SPI_Set_Active(&SPI2_Read); // Sets the SPI2 module active`

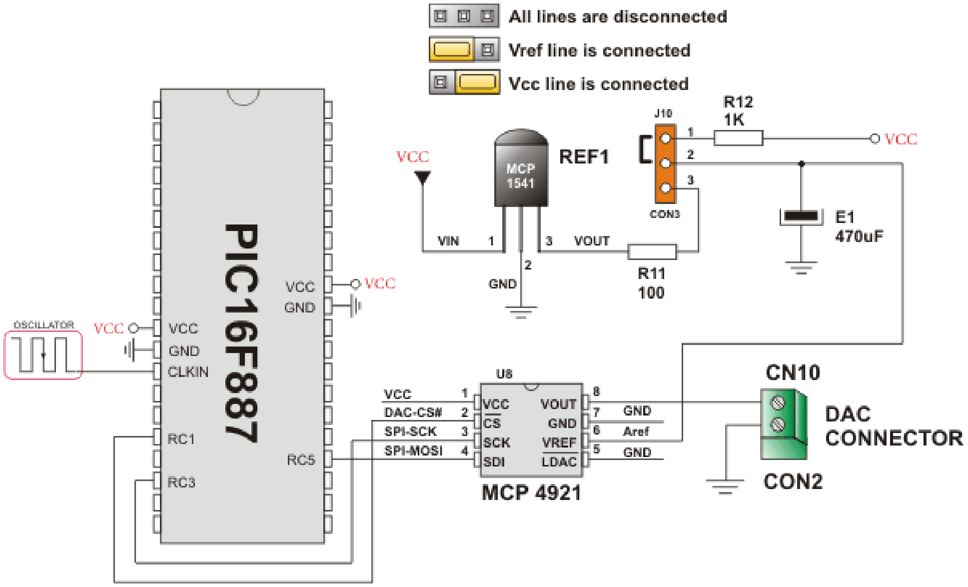
Library Example

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

```
// DAC module connections
sbit Chip_Select at RC0_bit;
sbit Chip_Select_Direction at TRISC0_bit;
// End DAC module connections

unsigned int value;

void InitMain() {
  TRISB0_bit = 1;                           // Set RA0 pin as input
  TRISB1_bit = 1;                           // Set RA1 pin as input
  Chip_Select = 1;                          // Deselect DAC
  Chip_Select_Direction = 0;               // Set CS# pin as Output
  SPI1_Init();                              // Initialize SPI module
}

// DAC increments (0..4095) --> output voltage (0..Vref)
void DAC_Output(unsigned int valueDAC) {
  char temp;

  Chip_Select = 0;                          // Select DAC chip

  // Send High Byte
  temp = (valueDAC >> 8) & 0x0F;         // Store valueDAC[11..8] to temp[3..0]
  temp |= 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
}

void main() {
  ANSEL = 0;
  ANSELH = 0;
  InitMain();              // Perform main initialization

  value = 2048;            // When program starts, DAC gives
                           //   the output in the mid-range

 while (1) {               // Endless loop

    if ((RA0_bit) && (value < 4095)) { // If RA0 button is pressed
      value++;                         //   increment value
      }
    else {
      if ((RA1_bit) && (value > 0)) {  // If RA1 button is pressed
        value--;                       //   decrement value
        }
      }
    DAC_Output(value);                 // Send value to DAC chip
    Delay_ms(1);                       // Slow down key repeat pace
  }
}
```

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 PIC with integrated SPI and more than 4 Kb ROM memory. 38 to 40 MHz clock is recommended to get from 8 to 10 Mhz SPI clock, otherwise PIC should be clocked by `ENC28J60` clock output due to its silicon bug in SPI hardware. If you try lower PIC clock speed, there might be board hang or miss some requests.

SPI Ethernet library supports:

- IPv4 protocol.
- ARP requests.
- ICMP echo requests.
- UDP requests.
- TCP requests (no stack, no packet reconstruction).
- ARP client with cache.
- DNS client.
- UDP client.
- DHCP client.
- packet fragmentation is NOT supported.

Note: Due to PIC16 RAM/Flash limitations PIC16 library does NOT have ARP, DNS, UDP and DHCP client support implemented.

Note: Global library variable `SPI_Ethernet_userTimerSec` is used to keep track of time for all client implementations (ARP, DNS, UDP and DHCP). It is user responsibility to increment this variable each second in it's code if any of the clients is used.

Note: For advanced users there are header files (`"eth_enc28j60LibDef.h"` and `"eth_enc28j60LibPrivate.h"`) in Uses\P16 and Uses\P18 folders of the compiler with description of all routines and global variables, relevant to the user, implemented in the SPI Ethernet Library.

Note: The appropriate hardware SPI module must be initialized before using any of the SPI Ethernet library routines. Refer to SPI Library.

For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine.

External dependencies of SPI Ethernet Library

The following variables must be defined in all projects using SPI Ethernet Library:	Description:	Example:
`extern sfr sbit SPI_Ethernet_CS`	ENC28J60 chip select pin.	`sbit SPI_Ethernet_CS at RC1_bit;`
`extern sfr sbit SPI_Ethernet_RST;`	ENC28J60 reset pin.	`sbit SPI_Ethernet_Rst at RC0_bit;`
`extern sfr sbit SPI_Ethernet_CS_Direction;`	Direction of the ENC28J60 chip select pin.	`sbit SPI_Ethernet_CS_Direction at TRISC1_bit;`
`extern sfr sbit SPI_Ethernet_RST_Direction;`	Direction of the ENC28J60 reset pin.	`sbit SPI_Ethernet_Rst_Direction at TRISC0_bit;`

The following routines must be defined in all project using SPI Ethernet Library:	Description:	Example:
`unsigned int SPI_Ethernet_UserTCP(unsigned char *remoteHost, unsigned int remotePort, unsigned int localPort, unsigned int reqLength);`	TCP request handler.	Refer to the library example at the bottom of this page for code implementation.
`unsigned int SPI_Ethernet_UserUDP(unsigned char *remoteHost, unsigned int remotePort, unsigned int destPort, unsigned int reqLength);`	UDP request handler.	Refer to the library example at the bottom of this page for code implementation.

Library Routines

PIC16 and PIC18:

- 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

PIC18 Only:

- SPI_Ethernet_getIpAddress
- SPI_Ethernet_getGwIpAddress
- SPI_Ethernet_getDnsIpAddress
- SPI_Ethernet_getIpMask
- SPI_Ethernet_confNetwork
- SPI_Ethernet_arpResolve
- SPI_Ethernet_sendUDP
- SPI_Ethernet_dnsResolve
- SPI_Ethernet_initDHCP
- SPI_Ethernet_doDHCPLeaseTime
- SPI_Ethernet_renewDHCP

Spi_Ethernet_Init

Prototype	`void SPI_Ethernet_Init(unsigned char *mac, unsigned char *ip, unsigned char fullDuplex);`
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). 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	Global variables: - `SPI_Ethernet_CS`: Chip Select line - `SPI_Ethernet_CS_Direction`: Direction of the Chip Select pin - `SPI_Ethernet_RST`: Reset line - `SPI_Ethernet_RST_Direction`: Direction of the Reset pin must be defined before using this function. The SPI module needs to be initialized. See the SPI1_Init and SPI1_Init_Advanced routines.
Example	

```
#define SPI_Ethernet_HALFDUPLEX      0
#define SPI_Ethernet_FULLDUPLEX      1

// mE ehternet NIC pinout
sfr sbit SPI_Ethernet_Rst at RC0_bit;
sfr sbit SPI_Ethernet_CS  at RC1_bit;
sfr sbit SPI_Ethernet_Rst_Direction at TRISC0_bit;
sfr sbit SPI_Ethernet_CS_Direction  at TRISC1_bit;
// end ethernet NIC definitions

unsigned char myMacAddr[6]  = {0x00, 0x14, 0xA5, 0x76, 0x19,
0x3f} ; // my MAC address
unsigned char myIpAddr      = {192, 168,   1, 60 } ;  // my IP
addr

SPI1_Init();
SPI_Ethernet_Init(myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX);
```

Spi_Ethernet_Enable

<table>
<tr><td>**Prototype**</td><td>`void SPI_Ethernet_Enable(unsigned char enFlt);`</td></tr>
<tr><td>**Returns**</td><td>Nothing.</td></tr>
<tr><td>**Description**</td><td>
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.

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>
Note: 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.

Note: 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>
</table>

Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init.
Example	`SPI_Ethernet_Enable(_SPI_Ethernet_CRC | _SPI_Ethernet_UNICAST);` `// enable CRC checking and Unicast traffic`

Spi_Ethernet_Disable

<table>
<tr><td>Prototype</td><td><code>void SPI_Ethernet_Disable(unsigned char disFlt);</code></td></tr>
<tr><td>Returns</td><td>Nothing.</td></tr>
<tr><td>Description</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:
- <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>0</td><td>0x01</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>1</td><td>0x02</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>2</td><td>0x04</td><td>not used</td><td><code>none</code></td></tr>
<tr><td>3</td><td>0x08</td><td>not used</td><td><code>none</code></td></tr>
<tr><td>4</td><td>0x10</td><td>not used</td><td><code>none</code></td></tr>
<tr><td>5</td><td>0x20</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>6</td><td>0x40</td><td>not used</td><td><code>none</code></td></tr>
<tr><td>7</td><td>0x80</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>Note: Advance filtering available in the <code>ENC28J60</code> module such as <code>Pattern Match</code>, <code>Magic Packet</code> and <code>Hash Table</code> can not be disabled by this routine.</p>
<p>Note: 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 <code>ENC28J60</code> 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 | _SPI_Ethernet_UNICAST);` `// disable CRC checking and Unicast traffic`

Spi_Ethernet_doPacket

Prototype	`unsigned char SPI_Ethernet_doPacket();`
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	

```
if (SPI_Ethernet_doPacket() == 0)(1) {   // process received pack-
ets
   ...
}
```

Spi_Ethernet_putByte

Prototype	`void SPI_Ethernet_putByte(unsigned char v);`
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	`char data;`
	`...`
	`SPI_Ethernet_putByte(data); // put an byte into ENC28J60 buffer`

Spi_Ethernet_putBytes

Prototype	`void SPI_Ethernet_putBytes(unsigned char *ptr, unsigned char n);`
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	`char *buffer =  "mikroElektronika";`
	`...`
	`SPI_Ethernet_putBytes(buffer, 16); // put an RAM array into ENC28J60 buffer`

Spi_Ethernet_putConstBytes

Prototype	`void SPI_Ethernet_putConstBytes(const unsigned char *ptr, unsigned char n);`
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 char *buffer =  "mikroElektronika";`
	`...`
	`SPI_Ethernet_putConstBytes(buffer, 16); // put a const array into ENC28J60 buffer`

Spi_Ethernet_putString

Prototype	`unsigned int SPI_Ethernet_putString(unsigned char *ptr);`
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	`char *buffer =  "mikroElektronika";`
	`...`
	`SPI_Ethernet_putString(buffer); // put a RAM string into ENC28J60 buffer`

Spi_Ethernet_putConstString

Prototype	`unsigned int SPI_Ethernet_putConstString(const unsigned char *ptr);`
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 char *buffer =  "mikroElektronika";`
	`...`
	`SPI_Ethernet_putConstString(buffer); // put a const string into ENC28J60 buffer`

Spi_Ethernet_getByte

Prototype	`unsigned char SPI_Ethernet_getByte();`
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	`char buffer;`
	`...`
	`buffer = SPI_Ethernet_getByte(); // read a byte from ENC28J60 buffer`

Spi_Ethernet_getBytes

Prototype	`void SPI_Ethernet_getBytes(unsigned char *ptr, unsigned int addr, unsigned char n);`
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	`char buffer[16];`
	`...`
	`SPI_Ethernet_getBytes(buffer, 0x100, 16); // read 16 bytes, starting from address 0x100`

Spi_Ethernet_UserTCP

Prototype	`unsigned int SPI_Ethernet_UserTCP(unsigned char *remoteHost, unsigned int remotePort, unsigned int localPort, unsigned int reqLength);`
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	`unsigned int SPI_Ethernet_UserUDP(unsigned char *remoteHost, unsigned int remotePort, unsigned int destPort, unsigned int reqLength);`
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.

SPI_Ethernet_getIpAddress

Prototype	`unsigned char * SPI_Ethernet_getIpAddress();`
Returns	Ponter to the global variable holding IP address.
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP address. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own IP address buffer. These locations should not be altered by the user in any case.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example	`unsigned char ipAddr[4]; // user IP address buffer`
	`...`
	`memcpy(ipAddr, SPI_Ethernet_getIpAddress(), 4); // fetch IP address`

SPI_Ethernet_getGwIpAddress

Prototype	`unsigned char * SPI_Ethernet_getGwIpAddress();`
Returns	Ponter to the global variable holding gateway IP address.
Description	This routine should be used when DHCP server is present on the network to fetch assigned gateway IP address. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own gateway IP address buffer. These locations should not be altered by the user in any case!
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example	`unsigned char gwIpAddr[4]; // user gateway IP address buffer`
	`...`
	`memcpy(gwIpAddr, SPI_Ethernet_getGwIpAddress(), 4); // fetch gateway IP address`

SPI_Ethernet_getDnsIpAddress

Prototype	`unsigned char * SPI_Ethernet_getDnsIpAddress()`
Returns	Ponter to the global variable holding DNS IP address.
Description	This routine should be used when DHCP server is present on the network to fetch assigned DNS IP address. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own DNS IP address buffer. These locations should not be altered by the user in any case.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example	`unsigned char dnsIpAddr[4]; // user DNS IP address buffer`
	`...`
	`memcpy(dnsIpAddr, SPI_Ethernet_getDnsIpAddress(), 4); // fetch DNS server address`

SPI_Ethernet_getIpMask

Prototype	`unsigned char * SPI_Ethernet_getIpMask()`
Returns	Ponter to the global variable holding IP subnet mask.
Description	This routine should be used when DHCP server is present on the network to fetch assigned IP subnet mask. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own IP subnet mask buffer. These locations should not be altered by the user in any case.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.
Example	

```
unsigned char IpMask[4] ;  // user IP subnet mask buffer
...
memcpy(IpMask, SPI_Ethernet_getIpMask(), 4); // fetch IP subnet mask
```

SPI_Ethernet_confNetwork

Prototype	`void SPI_Ethernet_confNetwork(char *ipMask, char *gwIpAddr, char *dnsIpAddr);`
Returns	Nothing.
Description	Configures network parameters (IP subnet mask, gateway IP address, DNS IP address) when DHCP is not used. Parameters: - `ipMask`: IP subnet mask. - `gwIpAddr` gateway IP address. - `dnsIpAddr`: DNS IP address. **Note**: The above mentioned network parameters should be set by this routine only if DHCP module is not used. Otherwise DHCP will override these settings
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.
Example	

```
char ipMask[4]    = {255, 255, 255,  0 };  // network mask (for example : 255.255.255.0)
char gwIpAddr[4]  = {192, 168,   1,  1 };  // gateway (router) IP address
char dnsIpAddr[4] = {192, 168,   1,  1 };  // DNS server IP address
...
SPI_Ethernet_confNetwork(ipMask, gwIpAddr, dnsIpAddr); // set network configuration parameters
```

SPI_Ethernet_arpResolve

Prototype	`unsigned char *SPI_Ethernet_arpResolve(unsigned char *ip, unsigned char tmax);`
Returns	- MAC address behind the IP address - the requested IP address was resolved. - `0` - otherwise.
Description	This is ARP module routine. It sends an ARP request for given IP address and waits for ARP reply. If the requested IP address was resolved, an ARP cash entry is used for storing the configuration. ARP cash can store up to 3 entries. For ARP cash structure refer to `"eth_enc28j60LibDef.h"` header file in the compiler's Uses/P18 folder. Parameters: - `ip`: IP address to be resolved. - `tmax:` time in seconds to wait for an reply. **Note**: The Ethernet services are not stopped while this routine waits for ARP reply. The incoming packets will be processed normaly during this time.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example	`unsigned char IpAddr[4]  = {192, 168,   1,   1 };   // IP address`
	`...`
	`SPI_Ethernet_arpResolve(IpAddr, 5); // get MAC address behind the above IP address, wait 5 secs for the response`

SPI_Ethernet_sendUDP

Prototype	`unsigned char SPI_Ethernet_sendUDP(unsigned char *destIP, unsigned int sourcePort, unsigned int destPort, unsigned char *pkt, unsigned int pktLen);`
Returns	- `1` - UDP packet was sent successfuly. - `0` - otherwise.
Description	This is UDP module routine. It sends an UDP packet on the network. Parameters: - `destIP`: remote host IP address. - `sourcePort`: local UDP source port number. - `destPort`: destination UDP port number. - `pkt`: packet to transmit. - `pktLen`: length in bytes of packet to transmit.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.
Example	see below

```
unsigned char IpAddr[4]  = {192, 168,   1,   1 };  // remote IP address
...
SPI_Ethernet_sendUDP(IpAddr, 10001, 10001, "Hello", 5); // send Hello message to the above IP address, from UDP port 10001 to UDP port 10001
```

SPI_Ethernet_dnsResolve

Prototype	`unsigned char * SPI_Ethernet_dnsResolve(unsigned char *host, unsigned char tmax);`
Returns	- pointer to the location holding the IP address - the requested host name was resolved. - `0` - otherwise.
Description	This is DNS module routine. It sends an DNS request for given host name and waits for DNS reply. If the requested host name was resolved, it's IP address is stored in library global variable and a pointer containing this address is returned by the routine. UDP port 53 is used as DNS port. Parameters: -`host`: host name to be resolved. -`tmax`: time in seconds to wait for an reply. **Note:** The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. **Note**: User should always copy the IP address from the RAM location returned by this routine into it's own resolved host IP address buffer. These locations should not be altered by the user in any case.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example	`unsigned char * remoteHostIpAddr[4]; // user host IP address buffer`
	`...`
	`// SNTP server:`
	`// Zurich, Switzerland: Integrated Systems Lab, Swiss Fed. Inst. of Technology`
	`// 129.132.2.21: swisstime.ethz.ch`
	`// Service Area: Switzerland and Europe`
	`memcpy(remoteHostIpAddr, SPI_Ethernet_dnsResolve("swisstime.ethz.ch", 5), 4);`

SPI_Ethernet_initDHCP

Prototype	`unsigned char SPI_Ethernet_initDHCP(unsigned char tmax);`
Returns	- `1` - network parameters were obtained successfully. - `0` - otherwise.
Description	This is DHCP module routine. It sends an DHCP request for network parameters (IP, gateway, DNS addresses and IP subnet mask) and waits for DHCP reply. If the requested parameters were obtained successfuly, their values are stored into the library global variables. These parameters can be fetched by using appropriate library IP get routines: - SPI_Ethernet_getIpAddress - fetch IP address. - SPI_Ethernet_getGwIpAddress - fetch gateway IP address. - SPI_Ethernet_getDnsIpAddress - fetch DNS IP address. - SPI_Ethernet_getIpMask - fetch IP subnet mask. UDP port `68` is used as DHCP client port and UDP port `67` is used as DHCP server port. Parameters: `- tmax`: time in seconds to wait for an reply. **Note**: The Ethernet services are not stopped while this routine waits for DNS reply. The incoming packets will be processed normaly during this time. **Note**: When DHCP module is used, global library variable `SPI_Ethernet_userTimerSec` is used to keep track of time. It is user responsibility to increment this variable each second in it's code.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example	`...`
	`SPI_Ethernet_initDHCP(5); // get network configuration from DHCP server, wait 5 sec for the response`
	`...`

SPI_Ethernet_doDHCPLeaseTime

Prototype	`unsigned char SPI_Ethernet_doDHCPLeaseTime();`
Returns	- `0` - lease time has not expired yet. - `1` - lease time has expired, it's time to renew it.
Description	This is DHCP module routine. It takes care of IP address lease time by decrementing the global lease time library counter. When this time expires, it's time to contact DHCP server and renew the lease.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example

```
while(1) {
   ...
   if(SPI_Ethernet_doDHCPLeaseTime())
      ... // it's time to renew the IP address lease
}
```

SPI_Ethernet_renewDHCP

Prototype	`unsigned char SPI_Ethernet_renewDHCP(unsigned char tmax);`
Returns	- `1` - upon success (lease time was renewed). - `0` - otherwise (renewal request timed out).
Description	This is DHCP module routine. It sends IP address lease time renewal request to DHCP server. Parameters: - `tmax`: time in seconds to wait for an reply.
Requires	Ethernet module has to be initialized. See SPI_Ethernet_Init. Available for PIC18 family MCUs only.

Example

```
while(1) {
   ...
   if(SPI_Ethernet_doDHCPLeaseTime())
      SPI_Ethernet_renewDHCP(5); // it's time to renew the IP
address lease, with 5 secs for a reply
   ...
}
```

Library Example

This code shows how to use the Ethernet mini 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 RD0 to RD7 bit and return HTML main page
  all other requests return also HTML main page.

```
// duplex config flags
#define Spi_Ethernet_HALFDUPLEX     0x00  // half duplex
#define Spi_Ethernet_FULLDUPLEX     0x01  // full duplex

// mE ehternet NIC pinout
sfr sbit SPI_Ethernet_Rst at RC0_bit;
sfr sbit SPI_Ethernet_CS  at RC1_bit;
sfr sbit SPI_Ethernet_Rst_Direction at TRISC0_bit;
sfr sbit SPI_Ethernet_CS_Direction  at TRISC1_bit;
// end ethernet NIC definitions

/************************************************************
 * ROM constant strings
 */
const unsigned char httpHeader[] = "HTTP/1.1 200 OKnContent-type: "
;  // HTTP header
const  unsigned  char  httpMimeTypeHTML[]  =  "text/htmlnn"  ;
// HTML MIME type
const  unsigned  char  httpMimeTypeScript[]  =  "text/plainnn"  ;
// TEXT MIME type
unsigned char httpMethod[] = "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 char   *indexPage =  // Change the IP address of the page to
be refreshed
"<meta http-equiv="refresh" content="3;url=http://192.168.20.60">
```

```
<HTML><HEAD></HEAD><BODY>
<h1>PIC + 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>ADC</th></tr>
<tr><td>AN2</td><td><script>document.write(AN2)</script></td></tr>
<tr><td>AN3</td><td><script>document.write(AN3)</script></td></tr>
</table></td><td><table border=1 style="font-size:20px ;font-family:
terminal ;">
<tr><th colspan=2>PORTB</th></tr>
<script>
var str,i;
str="";
for(i=0;i<8;i++)
{ str+="<tr><td bgcolor=pink>BUTTON #"+i+"</td>";
if(PORTB&(1<<i)){ str+="<td bgcolor=red>ON";}
else { str+="<td bgcolor=#cccccc>OFF";}
str+="</td></tr>";}
document.write(str) ;
</script>
" ;

const char    *indexPage2 =  "</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>
" ;

/***********************************
 * RAM variables
 */
unsigned char   myMacAddr[6]  = {0x00, 0x14, 0xA5, 0x76, 0x19, 0x3f} ;
// my MAC address
unsigned  char       myIpAddr[4]     =  {192,   168,   20,   60} ;
// my IP address
unsigned char   getRequest[15] ;      // HTTP request buffer
```

```
unsigned char    dyna[30];          // buffer for dynamic response
unsigned long    httpCounter = 0;    // counter of HTTP requests

/*******************************************
 *  functions
 */

/*
 * put the constant string pointed to by s to the ENC transmit buffer.
 */
/*unsigned int    putConstString(const char *s)
        {
        unsigned int ctr = 0;

        while(*s)
                {
                Spi_Ethernet_putByte(*s++);
                ctr++;
                }
        return(ctr);
        }*/
/*
 * it will be much faster to use library Spi_Ethernet_putConstString
routine
 * instead of putConstString routine above. However, the code will
be a little
 * bit bigger. User should choose between size and speed and pick the
implementation that
 * suites him best. If you choose to go with the putConstString def-
inition above
 * the #define line below should be commented out.
 *
 */
#define putConstString  SPI_Ethernet_putConstString

/*
 * put the string pointed to by s to the ENC transmit buffer
 */
/*unsigned int    putString(char *s)
        {
        unsigned int ctr = 0;

        while(*s)
                {
                Spi_Ethernet_putByte(*s++);

                ctr++;
                }
        return(ctr);
        }*/
```

```
/*
 * it will be much faster to use library Spi_Ethernet_putString rou-
tine
 * instead of putString routine above. However, the code will be a
little
 * bit bigger. User should choose between size and speed and pick the
implementation that
 * suites him best. If you choose to go with the putString defini-
tion above
 * the #define line below should be commented out.
 *
 */
#define putString  SPI_Ethernet_putString

/*
 * 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
 *
 */
unsigned int    SPI_Ethernet_UserTCP(unsigned char *remoteHost,
unsigned int remotePort, unsigned int localPort, unsigned int
reqLength)
        {
        unsigned int    len = 0;       // my reply length
        unsigned int    i;             // general purpose integer

        if(localPort != 80) // I listen only to web request on port 80
                {
                return(0);
                }

        // get 10 first bytes only of the request, the rest does not
matter here
        for(i = 0; i < 10; i++)
                {
                getRequest[ i]  = SPI_Ethernet_getByte();
                }
        getRequest[ i]  = 0;

          if(memcmp(getRequest, httpMethod, 5))         // only GET
method is supported here
                {
```

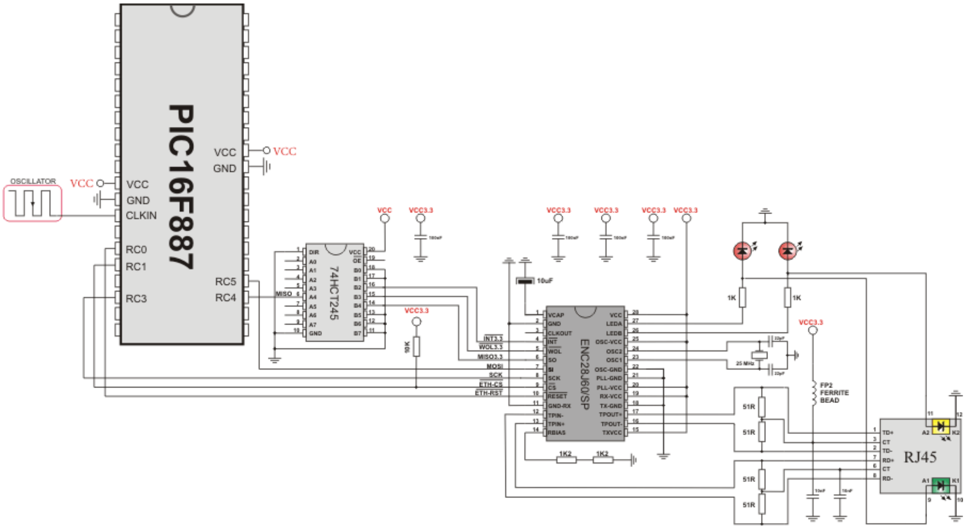
```
                    return(0);
                    }

               httpCounter++;                // one more request done

               if(getRequest[ 5] == 's')     // if request path name starts
with s, store dynamic data in transmit buffer
                    {
                     // the text string replied by this request can be
interpreted as javascript statements
                    // by browsers

                    len = putConstString(httpHeader);    // HTTP header
                     len += putConstString(httpMimeTypeScript); // with
text MIME type

                    // add AN2 value to reply
                    IntToStr(ADC_Read(2), dyna);
                    len += putConstString("var AN2=");
                    len += putString(dyna);
                    len += putConstString(";");

                    // add AN3 value to reply
                    IntToStr(ADC_Read(3), dyna);
                    len += putConstString("var AN3=");
                    len += putString(dyna);
                    len += putConstString(";");

                    // add PORTB value (buttons) to reply
                    len += putConstString("var PORTB=");
                    IntToStr(PORTB, dyna);
                    len += putString(dyna);
                    len += putConstString(";");

                    // add PORTD value (LEDs) to reply
                    len += putConstString("var PORTD=");
                    IntToStr(PORTD, dyna);
                    len += putString(dyna);
                    len += putConstString(";");

                    // add HTTP requests counter to reply
                    IntToStr(httpCounter, dyna);
                    len += putConstString("var REQ=");
                    len += putString(dyna);
                    len += putConstString(";");
                    }
              else if(getRequest[ 5] == 't')  // if request path name starts
with t, toggle PORTD (LED) bit number that comes after
                    {
                    unsigned char   bitMask = 0;     // for bit mask
```

```
                  if(isdigit(getRequest[ 6] ))   // if 0 <= bit number
<= 9, bits 8 & 9 does not exist but does not matter
                          {
                          bitMask = getRequest[ 6]  - '0'; // convert
ASCII to integer
                         bitMask = 1 << bitMask; // create bit mask
                        PORTD ^= bitMask;  // toggle PORTD with xor
operator
                          }
                 }

         if(len == 0)                // what do to by default
                 {
                 len =  putConstString(httpHeader); // HTTP header
                   len += putConstString(httpMimeTypeHTML); // with
HTML MIME type
                len += putConstString(indexPage); // HTML page first
part
                len += putConstString(indexPage2); // HTML page sec-
ond part
                 }

          return(len); // return to the library with the number of
bytes to transmit
         }

/*
 * 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
 *
 */
unsigned  int       SPI_Ethernet_UserUDP(unsigned  char  *remoteHost,
unsigned  int  remotePort,  unsigned  int  destPort,  unsigned  int
reqLength)
         {
         unsigned int    len;    // my reply length
         unsigned char   *ptr;   // pointer to the dynamic buffer

        // reply is made of the remote host IP address in human read-
able format
         ByteToStr(remoteHost[ 0] , dyna); // first IP address byte
         dyna[ 3]  = '.';
```

```
        ByteToStr(remoteHost[1], dyna + 4);   // second
        dyna[7] = '.';
        ByteToStr(remoteHost[2], dyna + 8);   // third
        dyna[11] = '.';
        ByteToStr(remoteHost[3], dyna + 12);  // fourth

        dyna[15] = ':';                       // add separator

        // then remote host port number
        WordToStr(remotePort, dyna + 16);
        dyna[21] = '[';
        WordToStr(destPort, dyna + 22);
        dyna[27] = ']';
        dyna[28] = 0;

         // the total length of the request is the length of the
dynamic string plus the text of the request
        len = 28 + reqLength;

        // puts the dynamic string into the transmit buffer
        SPI_Ethernet_putBytes(dyna, 28);

         // then puts the request string converted into upper char
into the transmit buffer
        while(reqLength--)
                {
             SPI_Ethernet_putByte(toupper(SPI_Ethernet_getByte()));
                }

        return(len);  // back to the library with the length of the
UDP reply
        }

/*
 * main entry
 */
void    main()
        {
        ANSEL = 0x0C;     // AN2 and AN3 convertors will be used
        PORTA = 0;
        TRISA = 0xff;     // set PORTA as input for ADC

        ANSELH = 0;       // Configure other AN pins as digital I/O
        PORTB = 0;
        TRISB = 0xff;     // set PORTB as input for buttons

        PORTD = 0;
        TRISD = 0;        // set PORTD as output

        /*
```

```
         * starts ENC28J60 with :
         * reset bit on RC0
         * CS bit on RC1
         * my MAC & IP address
         * full duplex
         */
        SPI1_Init();
        SPI_Ethernet_Init(myMacAddr, myIpAddr, Spi_Ethernet_FULLDU-
PLEX);

        while(1)                              // do forever
                {
           /*
            * 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
         */
                }
        }
```

HW Connection



SPI GRAPHIC LCD LIBRARY

The *mikroC PRO for PIC* 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.
For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine.

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.

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	`void SPI_Glcd_Init(char DeviceAddress);`
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. The SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.
Example	

```
// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

...

// If Port Expander Library uses SPI module :
SPI1_Init();  // Initialize SPI module used with PortExpander
SPI_Glcd_Init(0);
```

SPI_Glcd_Set_Side

Prototype	`void SPI_Glcd_Set_Side(char x_pos;`
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	`void SPI_Glcd_Set_Page(char page);`
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	`void SPI_Glcd_Set_X(char x_pos);`
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	`char SPI_Glcd_Read_Data();`
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	

```
char data;
...
data = SPI_Glcd_Read_Data();
```

SPI_Glcd_Write_Data

Prototype	`void SPI_Glcd_Write_Data(char Ddata);`
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	`char data;`
	`...`
	`SPI_Glcd_Write_Data(data);`

SPI_Glcd_Fill

Prototype	`void SPI_Glcd_Fill(char pattern);`
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	`void SPI_Glcd_Dot(char x_pos, char y_pos, char color);`
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	`void SPI_Glcd_Line(int x_start, int y_start, int x_end, int y_end, char color);`
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	`void SPI_Glcd_V_Line(char y_start, char y_end, char x_pos, char color);`
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	`void SPI_Glcd_H_Line(char x_start, char x_end, char y_pos, char color);`
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	`void SPI_Glcd_Rectangle(char x_upper_left, char y_upper_left, char x_bottom_right, char y_bottom_right, char color);`
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 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_Box

Prototype	`void SPI_Glcd_Box(char x_upper_left, char y_upper_left, char x_bottom_right, char y_bottom_right, char color);`
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	`void SPI_Glcd_Circle(int x_center, int y_center, int radius, char color);`
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	`void SPI_Glcd_Set_Font(const code char *activeFont, char aFontWidth, char aFontHeight, unsigned int aFontOffs);`
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 mikroC PRO character set and regular ASCII set (eg. if 'A' is 65 in ASCII character, and 'A' is 45 in the mikroC PRO 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” 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	`void SPI_Glcd_Write_Char(char chr1, char x_pos, char page_num, char color);`
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	`void SPI_Glcd_Write_Text(char text[], char x_pos, char page_num, char color);`
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	`void SPI_Glcd_Image(const code char *image);`
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 mikroC PRO for PIC pointer to const and pointer to RAM equivalency). Use the mikroC PRO's 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

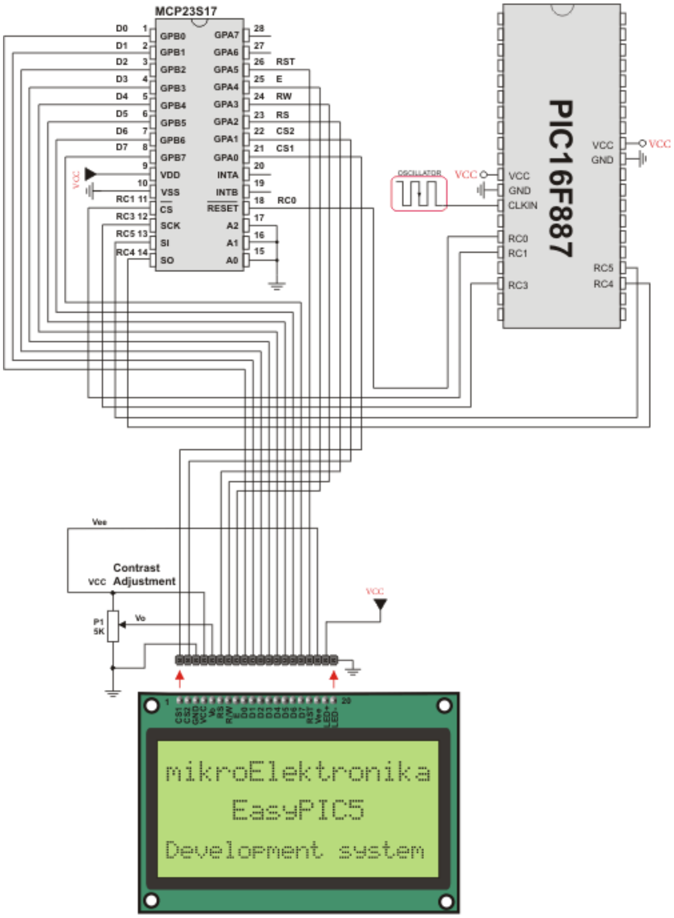
The example demonstrates how to communicate to KS0108 Glcd via the SPI module, using serial to parallel convertor MCP23S17.

```
const code char truck_bmp[1024];

// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

void Delay2s(){                           // 2 seconds delay function
  Delay_ms(2000);
}

void main() {
  char *someText;
  char counter;

  // If Port Expander Library uses SPI1 module
  SPI1_Init();         // Initialize SPI module used with PortExpander

//  // If Port Expander Library uses SPI2 module
```

```
//  SPI2_Init();    // Initialize SPI module used with PortExpander

  SPI_Glcd_Init(0);            // Initialize Glcd via SPI
  SPI_Glcd_Fill(0x00);         // Clear Glcd

  while(1) {

    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();

    for(counter = 5; counter < 60; counter+=5 ) {   // Draw horizon-
tal and vertical line
      Delay_ms(250);
      SPI_Glcd_V_Line(2, 54, counter, 1);
      SPI_Glcd_H_Line(2, 120, counter, 1);
      }
    Delay2s();

    SPI_Glcd_Fill(0x00);                          // Clear Glcd
    SPI_Glcd_Set_Font(Character8x7, 8, 8, 32);   // Choose font, see
__Lib_GLCDFonts.c in Uses folder
    SPI_Glcd_Write_Text("mikroE", 5, 7, 2);       // Write string

    for (counter = 1; counter <= 10; counter++)   // 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(System3x5, 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();

    }
}
```

HW Connection



SPI Glcd HW connection

SPI LCD LIBRARY

The *mikroC PRO for PIC* 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.
For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine.

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.

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	`void SPI_Lcd_Config(char DeviceAddress);`
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. The SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.

Example

```
// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

void main() {

// If Port Expander Library uses SPI module
SPI1_Init();      // Initialize SPI module used with PortExpander
SPI_Lcd_Config(0); // initialize Lcd over SPI interface
```

Spi_Lcd_Out

Prototype	`void SPI_Lcd_Out(char row, char column, char *text);`
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	`void SPI_Lcd_Out_CP(char *text);`
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	`void SPI_Lcd_Chr(char Row, char Column, char Out_Char);`
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	`void SPI_Lcd_Chr_CP(char Out_Char);`
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	`void SPI_Lcd_Cmd(char out_char);`
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 LCD Commands

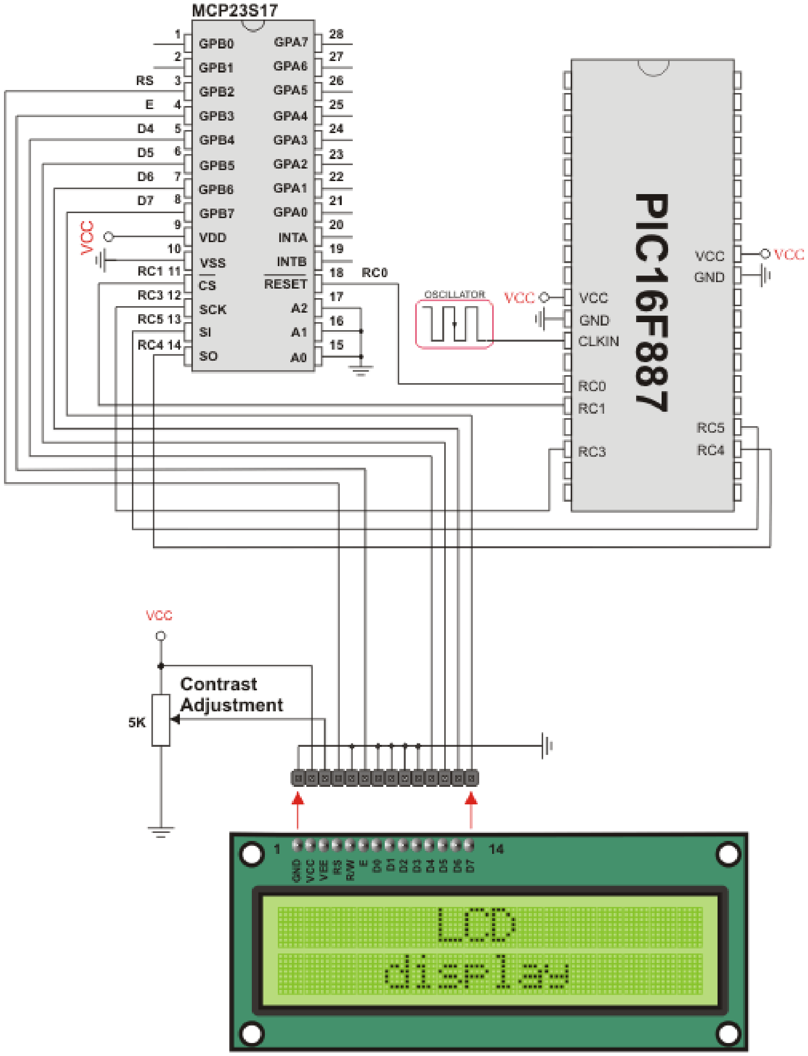
Lcd Command	Purpose
`LCD_FIRST_ROW`	Move cursor to the 1st row
`LCD_SECOND_ROW`	Move cursor to the 2nd row
`LCD_THIRD_ROW`	Move cursor to the 3rd row
`LCD_FOURTH_ROW`	Move cursor to the 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to its 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.

```
char *text = "mikroElektronika";

// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

void main() {

  // If Port Expander Library uses SPI1 module
  SPI1_Init();      // Initialize SPI module used with PortExpander

  // If Port Expander Library uses SPI2 module
  //  SPI2_Init(); // Initialize SPI module used with PortExpander

  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 col-
umn
  SPI_Lcd_Chr_CP('!');                     // Append '!'
  SPI_Lcd_Out(2,1, text); // Print text to Lcd, 2nd row, 1st column

  // SPI_Lcd_Out(3,1,"mikroE");   // For Lcd with more than two rows
  // SPI_Lcd_Out(4,15,"mikroE");  // For Lcd with more than two rows
}
```

HW Connection



SPI LCD HW connection

SPI LCD8 (8-BIT INTERFACE) LIBRARY

The *mikroC PRO for PIC* 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.
For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine.

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.

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	`void SPI_Lcd8_Config(char DeviceAddress);`
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. The SPI module needs to be initialized. See SPI1_Init and SPI1_Init_Advanced routines.
Example	

```
// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections
...
// If Port Expander Library uses SPI module
SPI1_Init();    // Initialize SPI module used with PortExpander
SPI_Lcd8_Config(0);   // intialize Lcd in 8bit mode via SPI
```

Spi_Lcd8_Out

Prototype	`void SPI_Lcd8_Out(unsigned short row, unsigned short column, char *text);`
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	`void SPI_Lcd8_Chr_CP(char out_char);`
Returns	Nothing.
Description	Prints character on Lcd at current cursor position. Both variables and literals can be passed as character. 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	`void SPI_Lcd8_Chr(unsigned short row, unsigned short column, char out_char);`
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	`void SPI_Lcd8_Chr_CP(char out_char);`
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	`void SPI_Lcd8_Cmd(char out_char);`
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 LCD Commands

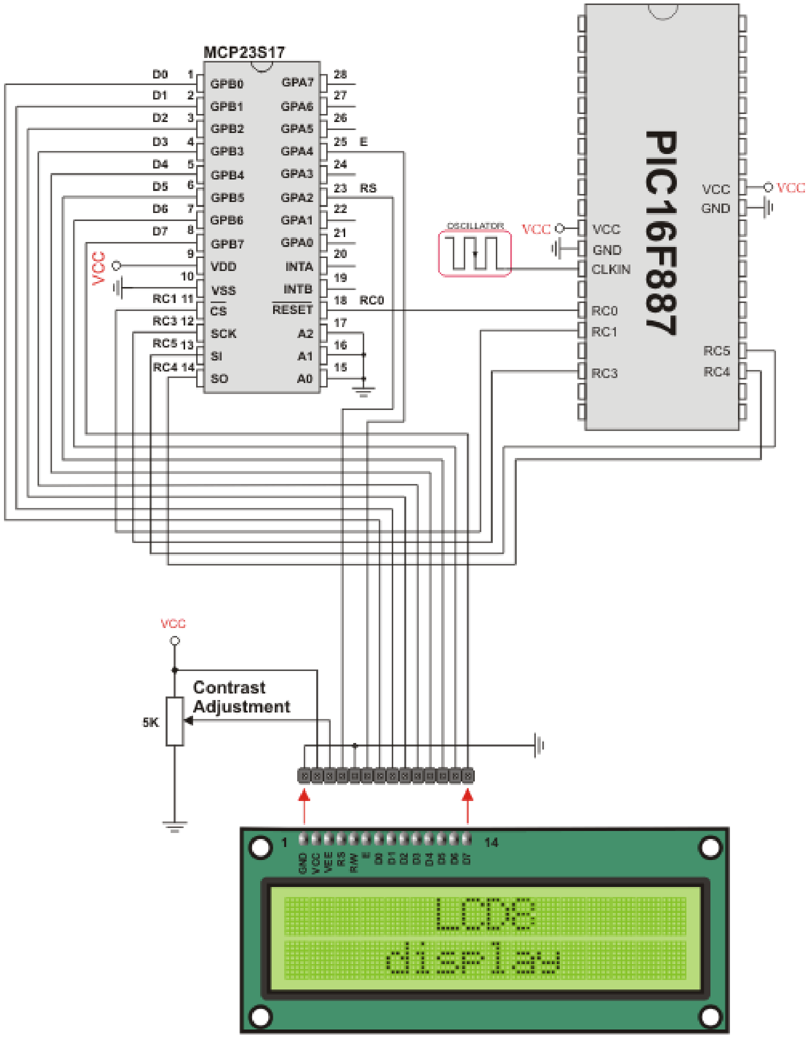
Lcd Command	**Purpose**
`LCD_FIRST_ROW`	Move cursor to the 1st row
`LCD_SECOND_ROW`	Move cursor to the 2nd row
`LCD_THIRD_ROW`	Move cursor to the 3rd row
`LCD_FOURTH_ROW`	Move cursor to the 4th row
`LCD_CLEAR`	Clear display
`LCD_RETURN_HOME`	Return cursor to home position, returns a shifted display to 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 in 8-bit mode via the SPI module, using serial to parallel convertor MCP23S17.

```
char *text = "mikroE";

// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

void main() {

  // If Port Expander Library uses SPI1 module
  SPI1_Init();      // Initialize SPI module used with PortExpander

  // If Port Expander Library uses SPI2 module
  //  SPI2_Init(); // Initialize SPI module used with PortExpander

  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
```

HW Connection



SPI LCD8 HW connection

SPI T6963C GRAPHIC LCD LIBRARY

The *mikroC PRO for PIC* 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.
For MCUs with two SPI modules it is possible to initialize both of them and then switch by using the `SPI_Set_Active()` routine.

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.

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

Note: The following low level library routines are implemented as macros. These macros can be found in the `SPI_T6963C.h` header file which is located in the SPI T6963C example projects folders.

- 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	`void SPI_T6963C_Config(unsigned int width, unsigned char height, unsigned char fntW, char DeviceAddress, unsigned char wr, unsigned char rd, unsigned char cd, unsigned char rst);`
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).

Description

```
schematic:
+---------------------+ /\
+ GRAPHICS PANEL #0    +   |
+                        +  |
+                        +  |
+                        +  |
+---------------------+   | PANEL 0
+ TEXT PANEL #0         +   |
+                        + \/
+---------------------+ /\
+ GRAPHICS PANEL #1    +   |
+                        +  |
+                        +  |
+                        +  |
+---------------------+   | PANEL 1
+ TEXT PANEL #1         +   |
+                        +  |
+---------------------+ \/
```

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

```
// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections
...
// Initialize SPI module
SPI1_Init();
SPI_T6963C_Config(240, 64, 8, 0, 0, 1, 3, 4);
```

Spi_T6963C_WriteData

Prototype	`void SPI_T6963C_writeData(unsigned char Ddata);`
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	`void SPI_T6963C_writeCommand(unsigned char Ddata);`
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	`void SPI_T6963C_setPtr(unsigned int p, unsigned char c);`
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	`void SPI_T6963C_waitReady(void);`
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	`void SPI_T6963C_fill(unsigned char v, unsigned int start, unsigned int len);`
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	`void SPI_T6963C_dot(int x, int y, unsigned char color);`
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	`void SPI_T6963C_write_char(unsigned char c, unsigned char x, unsigned char y, unsigned char mode);`
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	`void SPI_T6963C_write_text(unsigned char *str, unsigned char x, unsigned char y, unsigned char mode);`
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	`void SPI_T6963C_line(int x0, int y0, int x1, int y1, unsigned char pcolor);`
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	`void SPI_T6963C_rectangle(int x0, int y0, int x1, int y1, unsigned char pcolor);`
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	`void SPI_T6963C_box(int x0, int y0, int x1, int y1, unsigned char pcolor);`
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	`void SPI_T6963C_circle(int x, int y, long r, unsigned char pcolor);`
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	`void SPI_T6963C_image(const code char *pic);`
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 *mikroC PRO for PIC* pointer to const and pointer to RAM equivalency). Use the mikroC PRO's integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd. **Note**: Image dimension must match the display dimension.
Requires	Toshiba Glcd module needs to be initialized. See SPI_T6963C_Config routine.
Example	`SPI_T6963C_image(my_image);`

Spi_T6963C_Sprite

Prototype	`void SPI_T6963C_sprite(unsigned char px, unsigned char py, const code char *pic, unsigned char sx, unsigned char sy);`
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	`void SPI_T6963C_set_cursor(unsigned char x, unsigned char y);`
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	`void SPI_T6963C_clearBit(char b);`
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	`void SPI_T6963C_setBit(char b);`
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	`void SPI_T6963C_negBit(char b);`
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	`void SPI_T6963C_displayGrPanel(char n);`
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	`void SPI_T6963C_displayTxtPanel(char n);`
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	`void SPI_T6963C_setGrPanel(char n);`
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	`void SPI_T6963C_setTxtPanel(char n);`
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	`void SPI_T6963C_panelFill(unsigned char v);`
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	`void SPI_T6963C_grFill(unsigned char v);`
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	`void SPI_T6963C_txtFill(unsigned char v);`
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	`void SPI_T6963C_cursor_height(unsigned char n);`
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	`void SPI_T6963C_graphics(char n);`
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	`void SPI_T6963C_text(char n);`
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	`void SPI_T6963C_cursor(char n);`
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	`void SPI_T6963C_cursor_blink(char n);`
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, EasyPIC5 board and 16F887.

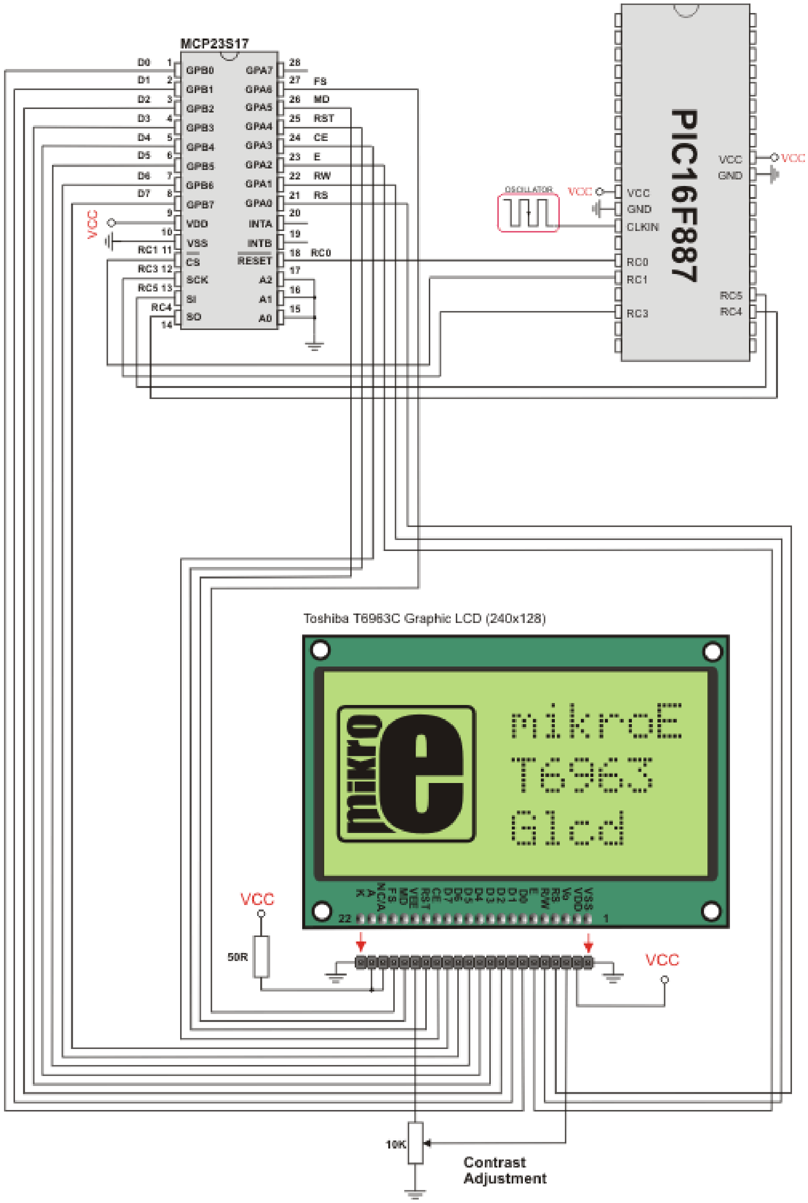
```
#include        "__SPIT6963C.h"

/*
 * bitmap pictures stored in ROM
 */
extern const code char me[];
extern const code char einstein[];

// Port Expander module connections
sbit  SPExpanderRST at RC0_bit;
sbit  SPExpanderCS  at RC1_bit;
sbit  SPExpanderRST_Direction at TRISC0_bit;
sbit  SPExpanderCS_Direction  at TRISC1_bit;
// End Port Expander module connections

void main() {

 char txt1[] = " EINSTEIN WOULD HAVE LIKED mE";
 char txt[] =  " GLCD LIBRARY DEMO, WELCOME !";

 unsigned char   panel;          // current panel
 unsigned int    i;              // general purpose register
 unsigned char   curs;           // cursor visibility
 unsigned int    cposx, cposy;   // cursor x-y position

 TRISA = 0xFF;                   // Configure PORTA as input
 ANSEL  = 0;                     // Configure AN pins as digital I/O
 ANSELH = 0;

  // If Port Expander Library uses SPI1 module
  SPI1_Init();    // Initialize SPI module used with PortExpander

//  // If Port Expander Library uses SPI2 module
//  SPI2_Init(); // Initialize SPI module used with PortExpander

  /*
   * 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
   */

  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 = 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(1) {               // Endless loop

    /*
      * If PORTA_0 is pressed, toggle the display between graphic
panel 0 and graphic 1
     */
    if(RA0_bit) {
      panel++;
      panel &= 1;
      SPI_T6963C_displayGrPanel(panel);
      Delay_ms(300);
      }

    /*
     * If PORTA_1 is pressed, display only graphic panel
     */
    else if(RA1_bit) {
      SPI_T6963C_graphics(1);
      SPI_T6963C_text(0);
      Delay_ms(300);
      }

    /*
     * If PORTA_2 is pressed, display only text panel
     */
    else if(RA2_bit) {
      SPI_T6963C_graphics(0);
      SPI_T6963C_text(1);
      Delay_ms(300);
      }

    /*
     * If PORTA_3 is pressed, display text and graphic panels
     */
    else if(RA3_bit) {
      SPI_T6963C_graphics(1);
      SPI_T6963C_text(1);
      Delay_ms(300);
      }

    /*
```

```
         * If PORTA_4 is pressed, change cursor
         */
        else if(RA4_bit) {
          curs++;
          if(curs == 3) curs = 0;
          switch(curs) {
            case 0:
              // no cursor
              SPI_T6963C_cursor(0);
              break;
            case 1:
              // blinking cursor
              SPI_T6963C_cursor(1);
              SPI_T6963C_cursor_blink(1);
              break;
            case 2:
              // non blinking cursor
              SPI_T6963C_cursor(1);
              SPI_T6963C_cursor_blink(0);
              break;
            }
          Delay_ms(300);
          }

        /*
         * Move cursor, even if not visible
         */
        cposx++;
        if(cposx == SPI_T6963C_txtCols) {
          cposx = 0;
          cposy++;
         if(cposy == SPI_T6963C_grHeight / SPI_T6963C_CHARACTER_HEIGHT)
{
            cposy = 0;
            }
          }
        SPI_T6963C_set_cursor(cposx, cposy);

        Delay_ms(100);
        }
    }
```

HW Connection



Spi T6963C GLCD HW connection

T6963C GRAPHIC LCD LIBRARY

The *mikroC PRO for PIC* 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:
`extern sfr char T6963C_dataPort;`	T6963C Data Port.	`char T6963C_dataPort at PORTD;`
`extern sfr sbit T6963C_ctrlwr;`	Write signal.	`sbit T6963C_ctrlwr at RC2_bit;`
`extern sfr sbit T6963C_ctrlrd;`	Read signal.	`sbit T6963C_ctrlrd at RC1_bit;`
`extern sfr sbit T6963C_ctrlcd;`	Command/Data signal.	`sbit T6963C_ctrlcd at RC0_bit;`
`extern sfr sbit T6963C_ctrlrst;`	Reset signal.	`sbit T6963C_ctrlrst at RC4_bit;`
`extern sfr sbit T6963C_ctrlwr_Direction;`	Direction of the Write pin.	`sbit T6963C_ctrlwr_Direction at TRISC2_bit;`
`extern sfr sbit T6963C_ctrlrd_Direction;`	Direction of the Read pin.	`sbit T6963C_ctrlrd_Direction at TRISC1_bit;`
`extern sfr sbit T6963C_ctrlcd_Direction;`	Direction of the Data pin.	`sbit T6963C_ctrlcd_Direction at TRISC0_bit;`
`extern sfr sbit T6963C_ctrlrst_Direction;`	Direction of the Reset pin.	`sbit T6963C_ctrlrst_Direction at TRISC4_bit;`

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

Note: The following low level library routines are implemented as macros. These macros can be found in the `T6963C.h` header file which is located in the T6963C example projects folders.

- T6963C_clearBit
- T6963C_setBit
- T6963C_negBit
- T6963C_displayGrPanel
- T6963C_displayTxtPanel
- T6963C_setGrPanel
- T6963C_setTxtPanel
- T6963C_panelFill
- T6963C_grFill
- T6963C_txtFill
- T6963C_cursor_height
- T6963C_graphics
- T6963C_text
- T6963C_cursor
- T6963C_cursor_blink

T6963C_Init

Prototype	`void T6963C_init(unsigned int width, unsigned char height, unsigned char fntW);`
Returns	Nothing.
Description	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 #1         +   |
+                        +   |
+----------------------+ \/
```

Requires	Global variables: - `T6963C_dataPort:` Data Port - `T6963C_ctrlwr:` Write signal pin - `T6963C_ctrlrd:` Read signal pin - `T6963C_ctrlcd:` Command/Data signal pin - `T6963C_ctrlrst:` Reset signal pin - `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
char T6963C_dataPort at PORTD;
sbit T6963C_ctrlwr  at RC2_bit;
sbit T6963C_ctrlrd  at RC1_bit;
sbit T6963C_ctrlcd  at RC0_bit;
sbit T6963C_ctrlrst at RC4_bit;
sbit T6963C_ctrlwr_Direction  at TRISC2_bit;
sbit T6963C_ctrlrd_Direction  at TRISC1_bit;
sbit T6963C_ctrlcd_Direction  at TRISC0_bit;
sbit T6963C_ctrlrst_Direction at TRISC4_bit;
// End of T6963C module connections
// Signals not used by library, they are set in main function
sbit T6963C_ctrlce at RC3_bit;        // CE signal
sbit T6963C_ctrlfs at RC6_bit;        // FS signal
sbit T6963C_ctrlmd at RC5_bit;        // MD signal
sbit T6963C_ctrlce_Direction  at TRISC3_bit;// CE signal direc-
tion
sbit T6963C_ctrlfs_Direction  at TRISC6_bit;      // FS signal
direction
sbit T6963C_ctrlmd_Direction  at TRISC5_bit;      // MD signal
direction
// End 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	`void T6963C_writeData(unsigned char mydata);`
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	`void T6963C_writeCommand(unsigned char mydata);`
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	`void T6963C_setPtr(unsigned int p, unsigned char c);`
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	`void T6963C_waitReady(void);`
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	`void T6963C_fill(unsigned char v, unsigned int start, unsigned int len);`
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	`void T6963C_dot(int x, int y, unsigned char color);`
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	`void T6963C_write_char(unsigned char c, unsigned char x, unsigned char y, unsigned char mode);`
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 logical “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: 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	`void T6963C_write_text(unsigned char *str, unsigned char x, unsigned char y, unsigned char mode);`
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 logical “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	`void T6963C_line(int x0, int y0, int x1, int y1, unsigned char pcolor);`
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	`void T6963C_rectangle(int x0, int y0, int x1, int y1, unsigned char pcolor);`
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, 219, 107, T6963C_WHITE);`

T6963C_box

Prototype	`void T6963C_box(int x0, int y0, int x1, int y1, unsigned char pcolor);`
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	`void T6963C_circle(int x, int y, long r, unsigned char pcolor);`
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	`void T6963C_image(const code char *pic);`
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 mikroC PRO for PIC pointer to const and pointer to RAM equivalency). Use the mikroC PRO's integrated Glcd Bitmap Editor (menu option **Tools › Glcd Bitmap Editor**) to convert image to a constant array suitable for displaying on Glcd. **Note**: Image dimension must match the display dimension.
Requires	Toshiba Glcd module needs to be initialized. See the T6963C_init routine.
Example	`T6963C_image(mc);`

T6963C_sprite

Prototype	`void T6963C_sprite(unsigned char px, unsigned char py, const code char *pic, unsigned char sx, unsigned char sy);`
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	`void T6963C_set_cursor(unsigned char x, unsigned char y);`
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_clearBit

Prototype	`void T6963C_clearBit(char b);`
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 the T6963C_init routine.
Example	`// clear bits 0 and 1 on control port` `T6963C_clearBit(0x03);`

T6963C_setBit

Prototype	`void T6963C_setBit(char b);`
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 the T6963C_init routine.
Example	`// set bits 0 and 1 on control port` `T6963C_setBit(0x03);`

T6963C_negBit

Prototype	`void T6963C_negBit(char b);`
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 the T6963C_init routine.
Example	`// negate bits 0 and 1 on control port` `T6963C_negBit(0x03);`

T6963C_displayGrPanel

Prototype	`void T6963C_displayGrPanel(char n);`
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	`void T6963C_displayTxtPanel(char n);`
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	`void T6963C_setTxtPanel(char n);`
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 text panel 1 as current text panel.` `T6963C_setTxtPanel(1);`

T6963C_SetTxtPanel

Prototype	`void T6963C_setTxtPanel(char n);`
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	`void T6963C_panelFill(unsigned char v);`
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	`void T6963C_grFill(unsigned char v);`
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	`void T6963C_txtFill(unsigned char v);`
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	`void T6963C_cursor_height(unsigned char n);`
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	`void T6963C_graphics(char n);`
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	`void T6963C_text(char n);`
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	`void T6963C_cursor(char n);`
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	`void T6963C_cursor_blink(char n);`
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, EasyPIC5 board and 16F887.

```
#include         "__T6963C.h"

// T6963C module connections
char T6963C_dataPort at PORTD;                     // DATA port

sbit T6963C_ctrlwr  at RC2_bit;                    // WR write signal
sbit T6963C_ctrlrd  at RC1_bit;                    // RD read signal
sbit T6963C_ctrlcd  at RC0_bit;                    // CD command/data signal
sbit T6963C_ctrlrst at RC4_bit;                    // RST reset signal
sbit T6963C_ctrlwr_Direction  at TRISC2_bit;       // WR write signal
sbit T6963C_ctrlrd_Direction  at TRISC1_bit;       // RD read signal
sbit T6963C_ctrlcd_Direction  at TRISC0_bit;       // CD command/data signal
sbit T6963C_ctrlrst_Direction at TRISC4_bit;       // RST reset signal

// Signals not used by library, they are set in main function
sbit T6963C_ctrlce at RC3_bit;                     // CE signal
sbit T6963C_ctrlfs at RC6_bit;                     // FS signal
sbit T6963C_ctrlmd at RC5_bit;                     // MD signal
sbit T6963C_ctrlce_Direction  at TRISC3_bit;       // CE signal direction
sbit T6963C_ctrlfs_Direction  at TRISC6_bit;       // FS signal direction
sbit T6963C_ctrlmd_Direction  at TRISC5_bit;       // MD signal direction

// End T6963C module connections
```

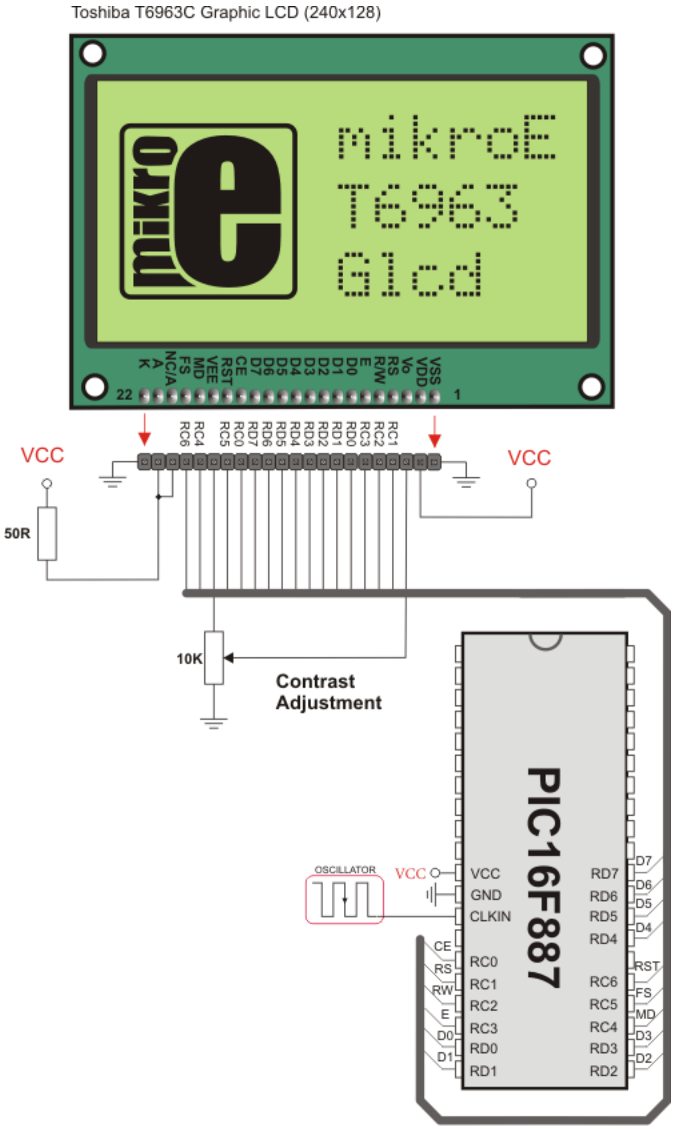
```
/*
 * bitmap pictures stored in ROM
 */
const code char mC[];
const code char einstein[];

void main() {
  char txt1[] = " EINSTEIN WOULD HAVE LIKED mE";
  char txt[] =  " GLCD LIBRARY DEMO, WELCOME !";

  unsigned char  panel;          // Current panel
  unsigned int   i;              // General purpose register
  unsigned char  curs;           // Cursor visibility
  unsigned int   cposx, cposy;   // Cursor x-y position

  TRISA0_bit = 1;                // Set RA0 as input
  TRISA1_bit = 1;                // Set RA1 as input
  TRISA2_bit = 1;                // Set RA2 as input
  TRISA3_bit = 1;                // Set RA3 as input
  TRISA4_bit = 1;                // Set RA4 as input

  T6963C_ctrlce_Direction = 0;
  T6963C_ctrlce = 0;             // Enable T6963C
  T6963C_ctrlfs_Direction = 0;
  T6963C_ctrlfs = 0;             // Font Select 8x8
  T6963C_ctrlmd_Direction = 0;
  T6963C_ctrlmd = 0;             // Column number select

  ANSEL  = 0;                    // Configure AN pins as digital I/O
  ANSELH = 0;

  // Initialize T6369C
  T6963C_init(240, 128, 8);

  /*
   * Enable both graphics and text display at the same time
   */
  T6963C_graphics(1);
  T6963C_text(1);

  panel = 0;
  i = 0;
  curs = 0;
  cposx = cposy = 0;
  /*
   * 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);

/*
 * 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(mC);

for(;;) {                        // Endless loop
  /*
   * If RA0 is pressed, display only graphic panel
   */
  if(RA0_bit) {
    T6963C_graphics(1);
```

```
          T6963C_text(0);
          Delay_ms(300);
          }

        /*
         * If RA1 is pressed, toggle the display between graphic panel
0 and graphic panel 1
         */
        else if(RA1_bit) {
          panel++;
          panel &= 1;
          T6963C_displayGrPanel(panel);
          Delay_ms(300);
          }

        /*
         * If RA2 is pressed, display only text panel
         */
        else if(RA2_bit) {
          T6963C_graphics(0);
          T6963C_text(1);
          Delay_ms(300);
          }

        /*
         * If RA3 is pressed, display text and graphic panels
         */
        else if(RA3_bit) {
          T6963C_graphics(1);
          T6963C_text(1);
          Delay_ms(300);
          }

        /*
         * If RA4 is pressed, change cursor
         */
        else if(RA4_bit) {
          curs++;
          if(curs == 3) curs = 0;
          switch(curs) {
            case 0:
              // no cursor
              T6963C_cursor(0);
              break;
            case 1:
              // blinking cursor
              T6963C_cursor(1);
              T6963C_cursor_blink(1);
              break;
            case 2:
```

```
            // non blinking cursor
            T6963C_cursor(1);
            T6963C_cursor_blink(0);
            break;
          }
        Delay_ms(300);
        }

      /*
       * Move cursor, even if not visible
       */
      cposx++;
      if(cposx == T6963C_txtCols) {
        cposx = 0;
        cposy++;
        if(cposy == T6963C_grHeight / T6963C_CHARACTER_HEIGHT) {
          cposy = 0;
          }
        }
      T6963C_set_cursor(cposx, cposy);

      Delay_ms(100);
      }
  }
```

HW Connection



T6963C GLCD HW connection

UART LIBRARY

UART hardware module is available with a number of PIC MCUs. mikroC PRO for PIC 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 PIC MCU with hardware integrated UART, for example 16F887. Then, simply use the functions listed below.

Note: Some PIC18 MCUs have multiple UART modules. Switching between the UART modules in the UART library is done by the UART_Set_Active function (UART module has to be previously initialized).

Note: In order to use the desired UART library routine, simply change the number 1 in the prototype with the appropriate module number, i.e. `UART2_Init(2400)`;

Library Routines

- UART1_Init
- UART1_Data_Ready
- UART1_Tx_Idle
- UART1_Read
- UART1_Read_Text
- UART1_Write
- UART1_Write_Text
- UART_Set_Active

Uart_Init

Prototype	`void UART1_Init(unsigned long baud_rate);`
Returns	Nothing.
Description	Initializes desired hardware UART module with the desired baud rate. Refer to the device data sheet for baud rates allowed for specific `Fosc`. If you specify the unsupported baud rate, compiler will report an error.
Requires	You need PIC MCU with hardware UART. `UART1_Init` needs to be called before using other functions from UART Library. Parameters: - `baud_rate`: requested baud rate Refer to the device data sheet for baud rates allowed for specific Fosc. **Note**: Calculation of the UART baud rate value is carried out by the compiler, as it would produce a relatively large code if performed on the libary level. Therefore, compiler needs to know the value of the parameter in the compile time. That is why this parameter needs to be a constant, and not a variable.
Example	This will initialize hardware UART1 module and establish the communication at 2400 bps: `UART1_Init(2400);`

Uart_Data_Ready

Prototype	`char UART1_Data_Ready();`
Returns	- 1 if data is ready for reading - 0 if there is no data in the receive register
Description	Use the function to test if data in receive buffer is ready for reading.
Requires	UART HW module must be initialized and communication established before using this function. See UART1_Init.

Example

```
// If data is ready, read it:
if (UART1_Data_Ready() == 1) {
 receive = UART1_Read();
 }
```

UART1_Tx_Idle

Prototype	`char UART1_Tx_Idle();`
Returns	- 1 if data is ready for reading - 0 if there is no data in the receive register
Description	Use the function to test if the transmit shift register is empty or not.
Requires	UART HW module must be initialized and communication established before using this function. See UART1_Init.

Example

```
// If the previous data has been shifted out, send next data:
if (UART1_Tx_Idle() == 1) {
  UART1_Write(_data);
 }
```

UART1_Read

Prototype	`char UART1_Read();`
Returns	Returns the received byte.
Description	Function receives a byte via UART. Use the function UART1_Data_Ready to test if data is ready first.
Requires	UART HW module must be initialized and communication established before using this function. See UART1_Init.

Example

```
// If data is ready, read it:
if (UART1_Data_Ready() == 1) {
 receive = UART1_Read();
 }
```

UART1_Read_Text

Prototype	`void UART1_Read_Text(char *Output, char *Delimiter, char Attempts);`
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 UART1_Init.

Example

```
Read text until the sequence “OK” is received, and send back what’ s
been received:

UART1_Init(4800);             // initialize UART1 module
Delay_ms(100);

 while (1) {
   if (UART1_Data_Ready() == 1) {   // if data is received
     UART1_Read_Text(output, "delim", 10); // reads text until
'delim' is found
     UART1_Write_Text(output); // sends back text
 }
}
```

UART1_Write

Prototype	`void UART1_Write(char _data);`
Returns	Nothing.
Description	The function transmits a byte via the UART module. Parameters: `_data`: data to be sent
Requires	UART HW module must be initialized and communication established before using this function. See UART1_Init.

Example

```
unsigned char _data = 0x1E;
...
UART1_Write(_data);
```

UART1_Write_Text

Prototype	`void UART1_Write_Text(char * UART_text);`
Returns	Nothing.
Description	Sends text (parameter UART_text) via UART. Text should be zero terminated.
Requires	UART HW module must be initialized and communication established before using this function. See UART1_Init.
Example	Read text until the sequence “OK” is received, and send back what’ s been received:

```
UART1_Init(4800);                // initialize UART1 module
Delay_ms(100);

 while (1) {
   if (UART1_Data_Ready() == 1) {     // if data is received
     UART1_Read_Text(output, "delim", 10); // reads text until
'delim' is found
     UART1_Write_Text(output);        // sends back text
 }
}
```

UART_Set_Active

Prototype	`void UART_Set_Active(char (*read_ptr)(), void (*write_ptr)(unsigned char data_), char (ready_ptr)(), char (*tx_idle_ptr)())`
Returns	Nothing.
Description	Sets active UART module which will be used by the UART library routines. Parameters: - `read_ptr`: UART1_Read handler - `write_ptr`: UART1_Write handler - `ready_ptr`: UART1_Data_Ready handler - `tx_idle_ptr`: UART1_Tx_Idle handler
Requires	Routine is available only for MCUs with two UART modules. Used UART module must be initialized before using this routine. See UART1_Init routine
Example	

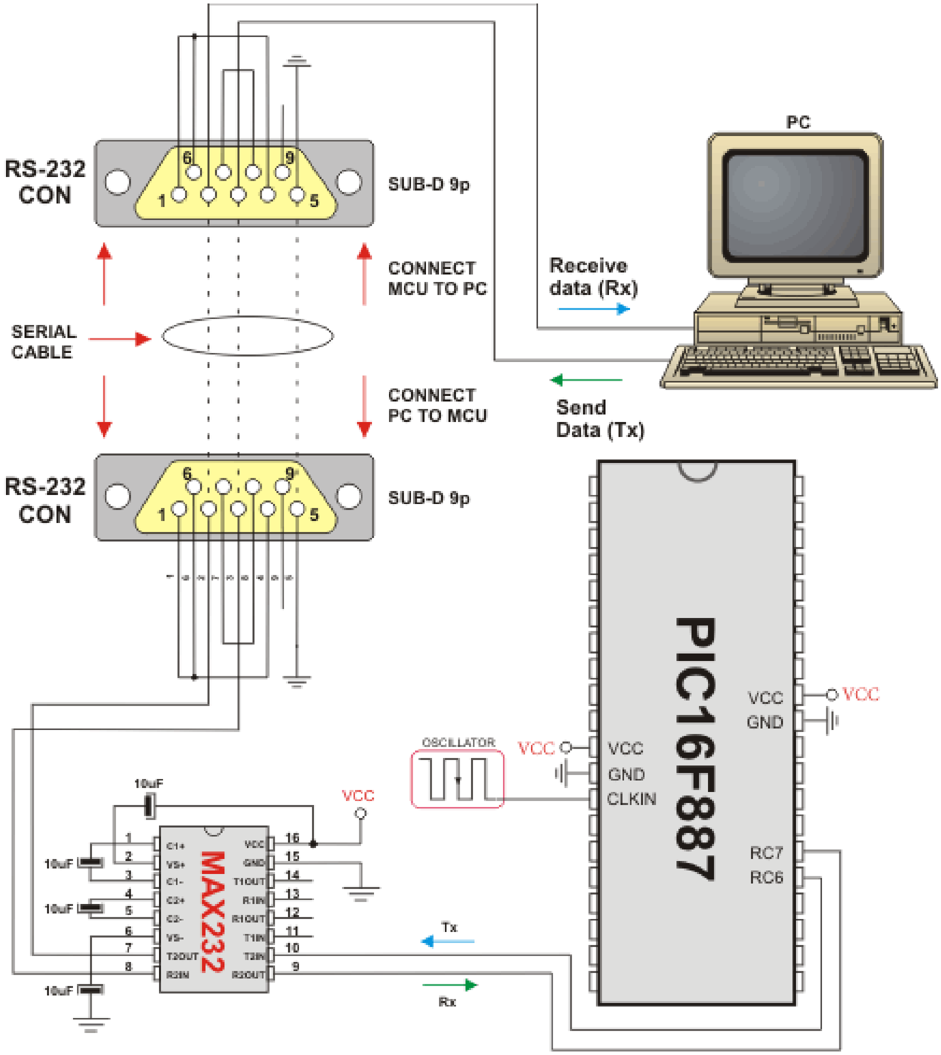
```
// Activate UART2 module
UART_Set_Active(&UART1_Read, &UART1_Write, &UART1_Data_Ready,
&UART1_Tx_Idle);
```

Library Example

The example demonstrates a simple data exchange via UART. When PIC MCU receives data, it immediately sends it back. If PIC is connected to the PC (see the figure below), you can test the example from the *mikroC PRO for PIC* terminal for RS-232 communication, menu choice **Tools › Terminal**.

```
char uart_rd;

void main() {

  UART1_Init(9600);                // Initialize UART module at 9600 
bps
  Delay_ms(100);                   // Wait for UART module to stabi-
lize

  while (1) {                       // Endless loop
   if (UART1_Data_Ready()) {        // If data is received,
     uart_rd = UART1_Read();        //   read the received data,
     UART1_Write(uart_rd);          //   and send data via UART
    }
  }
}
```

HW Connection



RS-232 HW connection

USB HID LIBRARY

Universal Serial Bus (USB) provides a serial bus standard for connecting a wide variety of devices, including computers, cell phones, game consoles, PDA's, etc.

mikroC PRO for PIC includes a library for working with human interface devices via Universal Serial Bus. A human interface device or HID is a type of computer device that interacts directly with and takes input from humans, such as the keyboard, mouse, graphics tablet, and the like.

Descriptor File

Each project based on the USB HID library should include a descriptor source file which contains vendor id and name, product id and name, report length, and other relevant information. To create a descriptor file, use the integrated USB HID terminal of mikroC PRO for PIC(**Tools › USB HID Terminal**). The default name for descriptor file is `USBdsc.c`, but you may rename it.

The provided code in the "Examples" folder works at 48MHz, and the flags should not be modified without consulting the appropriate datasheet first.

Library Routines

- Hid_Enable
- Hid_Read
- Hid_Write
- Hid_Disable

Hid_Enable

Prototype	`void Hid_Enable(unsigned readbuff, unsigned writebuff);`
Returns	Nothing.
Description	Enables USB HID communication. Parameters `readbuff` and `writebuff` are the Read Buffer and the Write Buffer, respectively, which are used for HID communication. This function needs to be called before using other routines of USB HID Library.
Requires	Nothing.
Example	`Hid_Enable(&rd, &wr);`

Hid_Read

Prototype	`unsigned char Hid_Read(void);`
Returns	Number of characters in the Read Buffer received from the host.
Description	Receives message from host and stores it in the Read Buffer. Function returns the number of characters received in the Read Buffer.
Requires	USB HID needs to be enabled before using this function. See Hid_Enable.
Example	`get = Hid_Read();`

Hid_Write

Prototype	`unsigned short Hid_Write(unsigned writebuff, unsigned short len);`
Returns	1 if data was successfuly sent, 0 if not.
Description	Function sends data from Write Buffer `writebuff` to host. Write Buffer is the same parameter as used in initialization; see Hid_Enable. Parameter `len` should specify a length of the data to be transmitted. Function call needs to be repeated as long as data is not successfuly sent.
Requires	USB HID needs to be enabled before using this function. See Hid_Enable.
Example	`// retry until success.` `while(!Hid_Write(&my_Usb_Buff, 1));`

Hid_Disable

Prototype	`void Hid_Disable(void);`
Returns	Nothing.
Description	Disables USB HID communication.
Requires	USB HID needs to be enabled before using this function. See Hid_Enable.
Example	`Hid_Disable();`

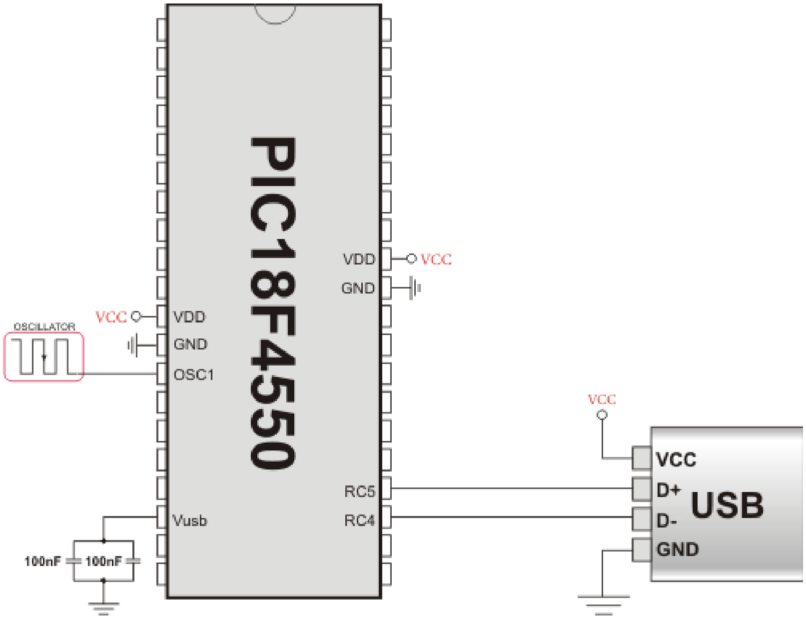
Library Example

The following example continually sends sequence of numbers 0..255 to the PC via Universal Serial Bus. `usbdsc.c` must be included in the project (via mikroC PRO for PIC IDE tool or via `#include` mechanism in source code).

```
unsigned short m, k;
unsigned short userRD_buffer[64];
unsigned short userWR_buffer[64];

void interrupt() {
  asm CALL _Hid_InterruptProc
  asm nop
}

void Init_Main() {
  // Disable all interrupts
  // Disable GIE, PEIE, TMR0IE, INT0IE,RBIE
  INTCON = 0;
  INTCON2 = 0xF5;
  INTCON3 = 0xC0;
  // Disable Priority Levels on interrupts
  RCON.IPEN = 0;
  PIE1 = 0;
  PIE2 = 0;
  PIR1 = 0;
  PIR2 = 0;

  // Configure all ports with analog function as digital
  ADCON1 |= 0x0F;

  // Ports Configuration
  TRISA = 0;
  TRISB = 0;
  TRISC = 0xFF;
  TRISD = 0xFF;
  TRISE = 0x07;
```

```
    LATA = 0;
    LATB = 0;
    LATC = 0;
    LATD = 0;
    LATE = 0;

    // Clear user RAM
    // Banks [ 00 .. 07] ( 8 x 256 = 2048 Bytes )
    asm {
      LFSR      FSR0, 0x000
      MOVLW     0x08
      CLRF      POSTINC0, 0
      CPFSEQ    FSR0H, 0
      BRA       $ - 2
    }

    // Timer 0
    T0CON   = 0x07;
    TMR0H   = (65536-156) >> 8;
    TMR0L   = (65536-156) & 0xFF;
    INTCON.T0IE = 1;              // Enable T0IE
    T0CON.TMR0ON = 1;
}


/** Main Program Routine **/

void main() {
    Init_Main();
    Hid_Enable(&userRD_buffer, &userWR_buffer);

    do {
      for (k = 0; k < 255; k++) {
        // Prepare send buffer
        userWR_buffer[ 0] = k;

        // Send the number via USB
        Hid_Write(&userWR_buffer, 1);
      }
    } while (1);

    Hid_Disable();
}
```

HW Connection



USB connection scheme

STANDARD ANSI C LIBRARIES

- ANSI C Ctype Library
- ANSI C Math Library
- ANSI C Stdlib Library
- ANSI C String Library

ANSI C Ctype Library

The *mikroC PRO for PIC* provides a set of standard ANSI C library functions for testing and mapping characters.

Note: Not all of the standard functions have been included.

Note: The functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming. Be sure to skim through the description before using standard C functions.

Library Functions

- isalnum
- isalpha
- iscntrl
- isdigit
- isgraph
- islower
- ispunct
- isspace
- isupper
- isxdigit
- toupper
- tolower

isalnum

Prototype	`unsigned short isalpha(char character);`
Description	Function returns 1 if the `character` is alphanumeric (A-Z, a-z, 0-9), otherwise returns zero.

isalpha

Prototype	`unsigned short isalpha(char character);`
Description	Function returns 1 if the `character` is alphabetic (A-Z, a-z), otherwise returns zero.

iscntrl

Prototype	`unsigned short iscntrl(char character);`
Description	Function returns 1 if the `character` is a control or delete character(decimal 0-31 and 127), otherwise returns zero.

isdigit

Prototype	`unsigned short isdigit(char character);`
Description	Function returns 1 if the `character` is a digit (0-9), otherwise returns zero.

isgraph

Prototype	`unsigned short isgraph(char character);`
Description	Function returns 1 if the `character` is a printable, excluding the space (decimal 32), otherwise returns zero.

islower

Prototype	`int islower(char character);`
Description	Function returns 1 if the `character` is a lowercase letter (a-z), otherwise returns zero.

ispunct

Prototype	`unsigned short ispunct(char character);`
Description	Function returns 1 if the `character` is a punctuation (decimal 32-47, 58-63, 91-96, 123-126), otherwise returns zero.

isspace

Prototype	`unsigned short isspace(char character);`
Description	Function returns 1 if the `character` is a white space (space, tab, CR, HT, VT, NL, FF), otherwise returns zero.

isupper

Prototype	`unsigned short isupper(char character);`
Description	Function returns 1 if the `character` is an uppercase letter (A-Z), otherwise returns zero.

isxdigit

Prototype	`unsigned short isxdigit(char character);`
Description	Function returns 1 if the `character` is a hex digit (0-9, A-F, a-f), otherwise returns zero.

toupper

Prototype	`unsigned short toupper(char character);`
Description	If the `character` is a lowercase letter (a-z), the function returns an uppercase letter. Otherwise, the function returns an unchanged input parameter.

tolower

Prototype	`unsigned short tolower(char character);`
Description	If the `character` is an uppercase letter (A-Z), function returns a lowercase letter. Otherwise, function returns an unchanged input parameter.

ANSI C Math Library

The *mikroC PRO for PIC* provides a set of standard ANSI C library functions for floating point math handling.

Note: Not all of the standard functions have been included.

Note: The functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming. Be sure to skim through the description before using standard C functions.

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	`double acos(double x);`
Description	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 Π (inclusive).

asin

Prototype	`double asin(double x);`
Description	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 -Π/2 and Π/2 (inclusive).

atan

Prototype	`double atan(double f);`
Description	Function computes the arc tangent of parameter f; that is, the value whose tangent is `f`. The return value is in radians, between -Π/2 and Π/2 (inclusive).

atan2

Prototype	`double atan2(double y, double x);`
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 -Π and Π (inclusive).

ceil

Prototype	`double ceil(double x);`
Description	Function returns value of parameter x rounded up to the next whole number.

cos

Prototype	`double cos(double f);`
Description	Function returns the cosine of `f` in radians. The return value is from -1 to 1.

cosh

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

eval_poly

Prototype	`static double eval_poly(double x, const double code * d, int n);`
Description	Function Calculates polynom for number `x`, with coefficients stored in `d[ ]`, for degree `n`.

exp

Prototype	`double exp(double x);`
Description	Function returns the value of `e` — the base of natural logarithms — raised to the power `x (i.e. ex)`.

fabs

Prototype	`double fabs(double d);`
Description	Function returns the absolute (i.e. positive) value of `d`.

floor

Prototype	`double floor(double x);`
Description	Function returns the value of parameter x rounded down to the nearest integer.

frexp

Prototype	`double frexp(double value, int *eptr);`
Description	Function splits a floating-point value into a normalized fraction and an integral power of 2. The return value is the normalized fraction and the integer exponent is stored in the object pointed to by `eptr`.

ldexp

Prototype	`double ldexp(double value, int newexp);`
Description	Function returns the result of multiplying the floating-point number `num` by 2 raised to the power `n` (i.e. returns `x * 2n`).

log

Prototype	`double log(double x);`
Description	Function returns the natural logarithm of x (i.e. `loge(x)`).

log10

Prototype	`double log10(double x);`
Description	Function returns the base-10 logarithm of x (i.e. `log10(x)`).

modf

Prototype	`double modf(double val, double * iptr);`
Description	Returns argument `val` split to the fractional part (function return val) and integer part (in number iptr).

pow

Prototype	`double pow(double x, double y);`
Description	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 `unsigned long`.

sin

Prototype	`double sin(double f);`
Description	Function returns the sine of `f` in radians. The return value is from -1 to 1.

sinh

Prototype	`double sinh(double x);`
Description	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	`double sqrt(double x);`
Description	Function returns the non negative square root of `x`.

tan

Prototype	`double tan(double x);`
Description	Function returns the tangent of `x` in radians. The return value spans the allowed range of floating point in the *mikroC PRO for PIC*.

tanh

Prototype	`double tanh(double x);`
Description	Function returns the hyperbolic tangent of `x`, defined mathematically as `sinh(x)/cosh(x)`.

ANSI C Stdlib Library

The *mikroC PRO for PIC* provides a set of standard ANSI C library functions of general utility.

Note: Not all of the standard functions have been included.

Note: Functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming. Be sure to skim through the description before using standard C functions.

Library Functions

- abs
- atof
- atoi
- atol
- div
- ldiv
- uldiv
- labs
- max
- min
- rand
- srand
- xtoi

abs

Prototype	`int abs(int a);`
Description	Function returns the absolute (i.e. positive) value of `a`.

atof

Prototype	`double atof(char *s)`
Description	Function converts the input string `s` into a double precision value and returns the value. Input string `s` should conform to the floating point literal format, with an optional whitespace at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (including a null character).

atoi

Prototype	`int atoi(char *s);`
Description	Function converts the input string `s` into an integer value and returns the value. The input string `s` should consist exclusively of decimal digits, with an optional whitespace and a sign at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (including a null character).

atol

Prototype	`long atol(char *s)`
Description	Function converts the input string `s` into a long integer value and returns the value. The input string `s` should consist exclusively of decimal digits, with an optional whitespace and a sign at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (including a null character).

div

Prototype	`div_t div(int number, int denom);`
Description	Function computes the result of division of the numerator `number` by the denominator `denom`; the function returns a structure of type `div_t` comprising quotient (`quot`) and remainder (`rem`), see Div Structures.

ldiv

Prototype	`ldiv_t ldiv(long number, long denom);`
Description	Function is similar to the div function, except that the arguments and result structure members all have type `long`. Function computes the result of division of the numerator `number` by the denominator `denom`; the function returns a structure of type `ldiv_t` comprising quotient (`quot`) and remainder (`rem`), see Div Structures.

uldiv

Prototype	`uldiv_t uldiv(unsigned long number, unsigned long denom);`
Description	Function is similar to the div function, except that the arguments and result structure members all have type `unsigned long`. Function computes the result of division of the numerator `number` by the denominator `denom`; the function returns a structure of type `uldiv_t` comprising quotient (`quot`) and remainder (`rem`), see Div Structures.

labs

Prototype	`long labs(long x);`
Description	Function returns the absolute (i.e. positive) value of long integer `x`.

max

Prototype	`int max(int a, int b);`
Description	Function returns greater of the two integers, `a` and `b`.

min

Prototype	`int min(int a, int b);`
Description	Function returns lower of the two integers, `a` and `b`.

rand

Prototype	`int rand();`
Description	Function returns a sequence of pseudo-random numbers between 0 and 32767. The function will always produce the same sequence of numbers unless srand is called to seed the start point.

srand

Prototype	`void srand(unsigned x);`
Description	Function uses `x` as a starting point for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. No values are returned by this function.

xtoi

Prototype	`unsigned xtoi(register char *s);`
Description	Function converts the input string `s` consisting of hexadecimal digits into an integer value. The input parameter `s` should consist exclusively of hexadecimal digits, with an optional whitespace and a sign at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (including a null character).

Div Structures

```
 typedef struct divstruct {
              int quot;
              int rem;
} div_t;


 typedef struct ldivstruct {
              long quot;
              long rem;
} ldiv_t;


 typedef struct uldivstruct {
              unsigned long quot;
              unsigned long rem;
} uldiv_t;
```

ANSI C String Library

The *mikroC PRO for PIC* provides a set of standard ANSI C library functions useful for manipulating strings and RAM memory.

Note: Not all of the standard functions have been included.

Note: Functions have been mostly implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming. Be sure to skim through the description before using standard C functions.

Library Functions

- memchr
- memcmp
- memcpy
- memmove
- memset
- strcat
- strchr
- strcmp
- strcpy
- strlen
- strncat
- strncpy
- strspn
- strncmp
- strstr
- strcspn
- strpbrk
- strrchr

memchr

Prototype	`void *memchr(void *p, char n, unsigned int v);`
Description	Function locates the first occurrence of `n` in the initial `v` bytes of memory area starting at the address `p`. The function returns the pointer to this location or `0` if the `n` was not found. For parameter `p` you can use either a numerical value (literal/variable/constant) indicating memory address or a dereferenced value of an object, for example `&mystring` or `&P0`.

memcmp

Prototype	`int memcmp(void *s1, void *s2, int n);`
Description	Function compares the first `n` characters of objects pointed to by `s1` and `s2` and returns zero if the objects are equal, or returns a difference between the first differing characters (in a left-to-right evaluation). Accordingly, the result is greater than zero if the object pointed to by `s1` is greater than the object pointed to by `s2` and vice versa.

memcpy

Prototype	`void *memcpy(void *d1, void *s1, int n);`
Description	Function copies `n` characters from the object pointed to by `s1` into the object pointed to by `d1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns address of the object pointed to by `d1`.

memmove

Prototype	`void *memmove(void *to, void *from, register int n);`
Description	Function copies `n` characters from the object pointed to by `from` into the object pointed to by `to`. Unlike memcpy, the memory areas `to` and `from` may overlap. The function returns address of the object pointed to by `to`.

memset

Prototype	`void *memset(void *p1, char character, int n)`
Description	Function copies the value of the `character` into each of the first `n` characters of the object pointed by `p1`. The function returns address of the object pointed to by `p1`.

strcat

Prototype	`char *strcat(char *to, char *from);`
Description	Function appends a copy of the string `from` to the string `to`, overwriting the null character at the end of `to`. Then, a terminating null character is added to the result. If copying takes place between objects that overlap, the behavior is undefined. `to` string must have enough space to store the result. The function returns address of the object pointed to by `to`.

strchr

Prototype	`char *strchr(char *ptr, char chr);`
Description	Function locates the first occurrence of character `chr` in the string `ptr`. The function returns a pointer to the first occurrence of character `chr`, or a null pointer if `chr` does not occur in `ptr`. The terminating null character is considered to be a part of the string.

strcmp

Prototype	`int strcmp(char *s1, char *s2);`
Description	Function compares strings `s1` and `s2` and returns zero if the strings are equal, or returns a difference between the first differing characters (in a left-to-right evaluation). Accordingly, the result is greater than zero if `s1` is greater than `s2` and vice versa.

strcpy

Prototype	`char *strcpy(char *to, char *from);`
Description	Function copies the string `from` into the string `to`. If copying is successful, the function returns to. If copying takes place between objects that overlap, the behavior is undefined.

strlen

Prototype	`int strlen(char *s);`
Description	Function returns the length of the string `s` (the terminating null character does not count against string’s length).

strncat

Prototype	`char *strncat(char *to, char *from, int size);`
Description	Function appends not more than `size` characters from the string `from` to `to`. The initial character of `from` overwrites the null character at the end of `to`. The terminating null character is always appended to the result. The function returns `to`.

strncpy

Prototype	`char *strncpy(char *to, char *from, int size);`
Description	Function copies not more than `size` characters from string `from` to `to`. If copying takes place between objects that overlap, the behavior is undefined. If `from` is shorter than `size` characters, then `to` will be padded out with null characters to make up the difference. The function returns the resulting string `to`.

strspn

Prototype	`int strspn(char *str1, char *str2);`
Description	Function returns the length of the maximum initial segment of `str1` which consists entirely of characters from `str2`. The terminating null character at the end of the string is not compared.

strncmp

Prototype	`int strncmp(char *s1, char *s2, char len);`

Description	Function lexicographically compares not more than `len` characters (characters that follow the null character are not compared) from the string pointed by `s1` to the string pointed by `s2`. The function returns a value indicating the `s1` and `s2` relationship:
	`Value      Meaning`
	`< 0        s1 "less than" s2`
	`= 0        s1 "equal to" s2`
	`> 0        s1 "greater than" s2`

strstr

Prototype	`char *strstr(char *s1, char *s2);`
Description	Function locates the first occurrence of the string `s2` in the string `s1` (excluding the terminating null character). The function returns pointer to first occurrence of `s2` in `s1`; if no string was found, function returns `0`. If `s2` is a null string, the function returns `0`.

strcspn

Prototype	`char *strcspn(char * s1, char *s2);`
Description	Function computes the length of the maximum initial segment of the string pointed to by `s1` that consists entirely of characters that are not in the string pointed to by `s2`. The function returns the length of the initial segment.

strpbrk

Prototype	`char *strpbrk(char * s1, char *s2);`
Description	Function searches `s1` for the first occurrence of any character from the string `s2`. The terminating null character is not included in the search. The function returns pointer to the matching character in `s1`. If `s1` contains no characters from `s2`, the function returns `0`.

strrchr

Prototype	`char *strrchr(char * ptr, unsigned int chr);`
Description	Function searches the string `ptr` for the last occurrence of character `chr.` The null character terminating `ptr` is not included in the search. The function returns pointer to the last `chr` found in `ptr`; if no matching character was found, function returns `0`.

MISCELLANEOUS LIBRARIES

- Button Library
- Conversions Library
- Sprint Library
- Setjmp Library
- Time Library
- Trigonometry Library

BUTTON LIBRARY

The Button library contains miscellaneous routines useful for a project development.

Library Routines

- Button

Button

Prototype	`unsigned short Button(unsigned short *port, unsigned short pin, unsigned short time, unsigned short active_state);`
Returns	Returns 0 or 255.
Description	Function eliminates the influence of contact flickering upon pressing a button (debouncing). Parameter `port` specifies the location of the button; parameter `pin` is the pin number on designated `port` and goes from 0..7; parameter `time` is a debounce period in milliseconds; parameter `active_state` can be either 0 or 1, and it determines if the button is active upon logical zero or logical one.
Requires	Button pin must be configured as input.
Example	Example reads RB0, to which the button is connected; on transition from 1 to 0 (release of button), PORTD is inverted:

```
do {
  if (Button(&PORTB, 0, 1, 1)) oldstate = 1;
  if (oldstate && Button(&PORTB, 0, 1, 0)) {
    PORTD = ~PORTD;
    oldstate = 0;
  }
} while(1);
```

CONVERSIONS LIBRARY

The *mikroC PRO for PIC* 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
- LongToStr
- LongWordToStr
- FloatToStr

The following functions convert decimal values to BCD and vice versa:

- Dec2Bcd
- Bcd2Dec16
- Dec2Bcd16

ByteToStr

Prototype	`void ByteToStr(unsigned short input, char *output);`
Returns	Nothing.
Description	Converts input byte to a string. The output string has fixed width of 4 characters including null character at the end (string termination). 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	Destination string should be at least 4 characters in length.

Example	`unsigned short t = 24;`
	`char txt[4];`
	`...`
	`ByteToStr(t, txt);  // txt is " 24" (one blank here)`

ShortToStr

Prototype	`void ShortToStr(short input, char *output);`
Returns	Nothing.
Description	Converts input signed short number to a string. The output string has fixed width of 5 characters including null character at the end (string termination). 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	Destination string should be at least 5 characters in length.

Example	`short t = -24;`
	`char txt[5];`
	`...`
	`ShortToStr(t, txt);  // txt is " -24" (one blank here)`

WordToStr

Prototype	`void WordToStr(unsigned input, char *output);`
Returns	Nothing.
Description	Converts input word to a string. The output string has fixed width of 6 characters including null character at the end (string termination). 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	Destination string should be at least 6 characters in length.

Example	`unsigned t = 437;`
	`char txt[ 6] ;`
	`...`
	`WordToStr(t, txt);  // txt is "  437" (two blanks here)`

IntToStr

Prototype	`void IntToStr(int input, char *output);`
Returns	Nothing.
Description	Converts input signed integer number to a string. The output string has fixed width of 7 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters: - `input:` signed integer number to be converted - `output:` destination string
Requires	Destination string should be at least 7 characters in length.

Example	`int j = -4220;`
	`char txt[ 7] ;`
	`...`
	`IntToStr(j, txt);  // txt is " -4220" (one blank here)`

LongintToStr

Prototype	`void LongToStr(long input, char *output);`
Returns	Nothing.
Description	Converts input signed long integer number to a string. The output string has fixed width of 12 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters: - `input:`signed long integer number to be converted - `output:` destination string
Requires	Destination string should be at least 12 characters in length.
Example	

```
long jj = -3700000;
char txt[12];
...
LongToStr(jj, txt);
// txt is "   -3700000" (three blanks here)
```

LongWordToStr

Prototype	`void LongWordToStr(unsigned long input, char *output);`
Returns	Nothing.
Description	Converts input unsigned long integer number to a string. The output string has fixed width of 11 characters including null character at the end (string termination). The output string is right justified and the remaining positions on the left (if any) are filled with blanks. Parameters: - `input:` unsigned long integer number to be converted - `output:` destination string
Requires	Destination string should be at least 11 characters in length.
Example	

```
unsigned long jj = 3700000;
char txt[11];
...
LongToStr(jj, txt);
// txt is "   3700000" (three blanks here)
```

FloatToStr

Prototype	`unsigned char FloatToStr(float fnum, unsigned char *str);`
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	Destination string should be at least 14 characters in length.

Example	`float ff1 = -374.2;`
	`float ff2 = 123.456789;`
	`float ff3 = 0.000001234;`
	`char txt[ 15] ;`
	`...`
	`FloatToStr(ff1, txt);  // txt is "-374.2"`
	`FloatToStr(ff2, txt);  // txt is "123.4567"`
	`FloatToStr(ff3, txt);  // txt is "1.234e-6"`

Dec2Bcd

Prototype	`unsigned short Dec2Bcd(unsigned short decnum);`
Returns	Converted BCD value.
Description	Converts input unsigned short integer number to its appropriate BCD representation. Parameters: - `decnum:` unsigned short integer number to be converted
Requires	Nothing.

Example	`unsigned short a, b;`
	`...`
	`a = 22;`
	`b = Dec2Bcd(a);     // b equals 34`

Bcd2Dec16

Prototype	`unsigned Bcd2Dec16(unsigned bcdnum);`
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	`unsigned a, b;`
	`...`
	`a = 0x1234;           // a equals 4660`
	`b = Bcd2Dec16(a);     // b equals 1234`

Dec2Bcd16

Prototype	`unsigned Dec2Bcd16(unsigned decnum);`
Returns	Converted BCD value.
Description	Converts unsigned 16-bit decimal value to its BCD equivalent. Parameters: - `decnum` unsigned 16-bit decimal number to be converted
Requires	Nothing.

Example	`unsigned a, b;`
	`...`
	`a = 2345;`
	`b = Dec2Bcd16(a);     // b equals 9029`

PRINTOUT LIBRARY

The *mikroC PRO for PIC* provides the PrintOut routine for easy data formatting and printing.

Note: Library works with PIC18 family only.

Library Routines

- PrintOut

PrintOut

Prototype	`void PrintOut(void (*prntoutfunc)(char ch), const char *f,...);`
Returns	Nothing.
Description	`PrintOut` is used to format data and print them in a way defined by the user through a print handler function. Parameters: - `prntoutfunc`: print handler function - `f`:format string The `f` argument is a format string and may be composed of characters, escape sequences, and format specifications. Ordinary characters and escape sequences are copied to the print handler in order in which they are interpreted. Format specifications always begin with a percent sign (`%`) and require additional arguments to be included in the function call. The format string is read from left to right. The first format specification encountered refers to the first argument after the `f` parameter and then converts and outputs it using the format specification. The second format specification accesses the second argument after `f`, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications. The format specifications have the following format: `% [flags] [width] [.precision]  [{ l | L }]  conversion_type` Each field in the format specification can be a single character or a number which specifies a particular format option. The `conversion_type` field is where a single character specifies that an argument is interpreted as a character, string, number, or pointer, as shown in the following table:

Description

`conversion_type`	**Argument Type**	**Output Format**
d	`int`	Signed decimal number
u	`unsigned int`	Unsigned decimal number
o	`unsigned int`	Unsigned octal number
x	`unsigned int`	Unsigned hexadecimal number using 0123456789abcdef
X	`unsigned int`	Unsigned hexadecimal number using 0123456789ABCEDF
f	`double`	Floating-point number using the format [-]dddd.dddd
e	`double`	Floating-point number using the format [-]d.dddde[-]dd
E	`double`	Floating-point number using the format [-]d.ddddE[-]dd
g	`double`	Floating-point number using either e or f format, whichever is more compact for the specified value and precision
c	`int`	int is converted to an `unsigned char`, and the resulting character is written
s	`char *`	String with a terminating null character
p	`void *`	Pointer value, the X format is used
%	`<none>`	A % is written. No argument is converted. The complete conversion specification shall be %%.

Description

The `flags` field is where a single character is used to justify the output and to print +/- signs and blanks, decimal points, and octal and hexadecimal prefixes, as shown in the following table.

`flags`	**Meaning**
`+`	Left justify the output in the specified field width.
`-`	Prefix the output value with + or - sign if the output is a signed type.
`space (' ')`	Prefix the output value with a blank if it is a signed positive value. Otherwise, no blank is prefixed.
`#`	Prefix a non-zero output value with `0`, `0x`, or `0X` when used with `o`, `x`, and `X` field types, respectively. When used with the `e`, `E`, `f`, `g`, and `G` field types, the `#` flag forces the output value to include a decimal point. In any other case the `#` flag is ignored.
`*`	Ignore format specifier.

The `width` field is a non-negative number that specifies a minimum number of printed characters. If a number of characters in the output value is less than width, blanks are added on the left or right (when the - flag is specified) in order to pad to the minimum width. If the width is prefixed with 0, then zeros are padded instead of blanks. The `width` field never truncates a field. If the length of the output value exceeds the specified width, all characters are output.

The `precision` field is a non-negative number that specifies the number of characters to print, number of significant digits, or number of decimal places. The precision field can cause truncation or rounding of the output value in the case of a floating-point number as specified in the following table.

`flags`	**MeaningMeaning of the `precision` field**
`d, u, o, x, X`	The precision field is where you specify the minimum number of digits that will be included in the output value. Digits are not truncated if the number of digits in an argument exceeds that defined in the precision field. If the number of digits in the argument is less than the precision field, the output value is padded on the left with zeros.
`f`	The precision field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
`e, E`	The precision field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
`g`	The precision field is where you specify the maximum number of significant digits in the output value.
`c, C`	The precision field has no effect on these field types.
`s`	The precision field is where you specify the maximum number of characters in the output value. Excess characters are not output.

Description	The optional characters `l` or `L` may immediately precede `conversion_type` to respectively specify long versions of the integer types `d`, `i`, `u`, `o`, `x`, and `X`. You must ensure that the argument type matches that of the format specification. You can use type casts to ensure that the proper type is passed to `printout`.
Requires	Nothing.
Example	Print mikroElektronika example's header file to UART.

Example

```
void PrintHandler(char c){

  UART1_Write(c);

}

void main(){
  UART1_Init(9600);
  Delay_ms(100);

  PrintOut(PrintHandler, "/*\r\n"
                            " * Project name:\r\n"
                            "      PrintOutExample (Sample usage
of PrintOut() function)\r\n"
                            " * Copyright:\r\n"
                            "      (c) MikroElektronika,
2006.\r\n"
                            " * Revision History:\r\n"
                            "      20060710:\r\n"
                            "        - Initial release\r\n"
                            " * Description:\r\n"
                            "      Simple demonstration on usage
of the PrintOut() function\r\n"
                            " * Test configuration:\r\n"
                            "      MCU:
PIC18F8520\r\n"
                            "      Dev.Board:        BigPIC5\r\n"
                            "      Oscillator:       HS,
%10.3fMHz\r\n"
                            "      Ext. Modules:     None.\r\n"
                            "      SW:                mikroC PRO
for PIC\r\n"
                            " * NOTES:\r\n"
                            "      None.\r\n"
                            " */\r\n", Get_Fosc_kHz()/1000.);

}
```

SETJMP LIBRARY

This library contains functions and types definitions for bypassing the normal function call and return discipline. The type declared is `jmp_buf` which is an array type suitable for holding the information needed to restore a calling environment.

Type declaration is contained in sejmp16.h and setjmp18.h header files for PIC16 and PIC18 family mcus respectively. These headers can be found in the include folder of the compiler. The implementation of this library is different for PIC16 and PIC18 family mcus. For PIC16 family Setjmp and Longjmp are implemented as macros defined in setjmp16.h header file and for PIC18 family as functions defined in setjmp library file.

Note: Due to PIC16 family specific of not being able to read/write stack pointer, the program execution after Longjmp ivocation occurs depends on the stack content. That is why, for PIC16 family only, implementation of Setjmp and Longjmp functions is not ANSI C standard compliant.

Library Routines

- Setjmp
- Longjmp

Setjmp

Prototype	`int setjmp(jmp_buf env);`
Returns	if the return is from direct invocation it returns 0 if the return is from a call to the longjmp it returns nonzero value
Description	This function saves calling position in `jmp_buf` for later use by longjmp. The parameter `env`: array of type (jmp_buf) suitible for holding the information needed for restoring calling environment.
Requires	Nothing.
Example	`setjmp(buf);`

Longjmp

Prototype	`void longjmp(jmp_buf env, int val);`
Returns	longjmp causes setjmp to return val, if val is 0 it will return 1.
Description	Restores calling environment saved in `jmp_buf` by most recent invocation of setjmp macro. If there has been no such invocation, or function conatinig the invocation of setjmp has terminated in the interim, the behaviour is undefined.Parameter `env`: array of type (jmp_buf) holding the information saved by corresponding setjmp invocation, `val`: char value, that will return corresponding setjmp.
Requires	Invocation of Longjmp must occur before return from the function in which Setjmp was called encounters.
Example	`longjmp(buf, 2);`

Library Example

Example demonstrates function cross calling using setjmp and longjmp functions. When called, Setjmp() saves its calling environment in its jmp_buf argument for later use by the Longjmp(). Longjmp(), on the other hand, restores the environment saved by the most recent invocation of the Setjmp() with the corresponding jmp_buf argument. The given example is for P16.
#include <Setjmp16.h>

```
#include <Setjmp16.h>

jmp_buf buf;               // Note: Program flow diagrams are indexed
according
                           // to the sequence of execution

void func33(){             //  2<----------|
                           //                 |
  asm nop;                 //                 |
  longjmp(buf, 2);         //  3-------------->|
  asm nop;                 //                |   |
                           //                |   |
}                          //                |   |
                           //                |   |
void func(){               //  1<------ |    |   |
                           //           |    |   |
  portb = 3;               //           |    |   |
  if (setjmp(buf) == 2)    //  3<--------------|
    portb = 1;             //  4-->|    |   |
  else                     //       |   |    |
    func33();              //  2---------->|
  asm nop;                 //       |   |
                           //  4<--|   |
}                          // 5-------|------>depends on stack content
                           //           |
void main() {              //           |
                           //           |
  PORTB = 0;               //           |
  TRISB = 0;               //           |
                           //           |
  asm nop;                 //           |
                           //           |
  func();                  //  1------>|
                           //
  asm nop;                 //
}
```

SPRINT LIBRARY

The *mikroC PRO for PIC* provides the standard ANSI C Sprintf function for easy data formatting.

Note: In addition to ANSI C standard, the Sprint Library also includes two limited versions of the `sprintf` function (`sprinti` and `sprintl`). These functions take less ROM and RAM and may be more convenient for use in some cases.

Functions

- sprintf
- sprintl
- sprinti

sprintf

Prototype	`sprintf(char *wh, const char *f,...);`
Returns	The function returns the number of characters actually written to destination string.
Description	`sprintf` is used to format data and print them into destination string. Parameters: - `wh:` destination string - `f`: format string The `f` argument is a format string and may be composed of characters, escape sequences, and format specifications. Ordinary characters and escape sequences are copied to the destination string in the order in which they are interpreted. Format specifications always begin with a percent sign (`%`) and require additional arguments to be included in the function call. The format string is read from left to right. The first format specification encountered refers to the first argument after `f` and then converts and outputs it using the format specification. The second format specification accesses the second argument after `f`, and so on. If there are more arguments than format specifications, then these extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications. The format specifications have the following format: `% [flags] [width] [.precision] [{ l	L }] conversion_type`

Description

Each field in the format specification can be a single character or a number which specifies a particular format option. The `conversion_type` field is where a single character specifies that the argument is interpreted as a character, string, number, or pointer, as shown in the following table:

`conversion_type`	**Argument Type**	**Output Format**
d	`int`	Signed decimal number
u	`unsigned int`	Unsigned decimal number
o	`unsigned int`	Unsigned octal number
x	`unsigned int`	Unsigned hexadecimal number using 0123456789abcdef
X	`unsigned int`	Unsigned hexadecimal number using 0123456789ABCEDF
f	`double`	Floating-point number using the format [-]dddd.dddd
e	`double`	Floating-point number using the format [-]d.dddde[-]dd
E	`double`	Floating-point number using the format [-]d.ddddE[-]dd
g	`double`	Floating-point number using either e or f format, whichever is more compact for the specified value and precision
c	`int`	int is converted to an `unsigned char`, and the resulting character is written
s	`char *`	String with a terminating null character
p	`void *`	Pointer value, the X format is used
%	`<none>`	A % is written. No argument is converted. The complete conversion specification shall be %%.

Description

The `flags` field is where a single character is used to justify the output and to print +/- signs and blanks, decimal points, and octal and hexadecimal prefixes, as shown in the following table.

`flags`	**Meaning**
`+`	Left justify the output in the specified field width.
`-`	Prefix the output value with + or - sign if the output is a signed type.
`space (' ')`	Prefix the output value with a blank if it is a signed positive value. Otherwise, no blank is prefixed.
`#`	Prefix a non-zero output value with `0`, `0x`, or `0X` when used with `o`, `x`, and `X` field types, respectively. When used with the `e`, `E`, `f`, `g`, and `G` field types, the `#` flag forces the output value to include a decimal point. In any other case the `#` flag is ignored.
`*`	Ignore format specifier.

The `width` field is a non-negative number that specifies a minimum number of printed characters. If a number of characters in the output value is less than width, blanks are added on the left or right (when the - flag is specified) in order to pad to the minimum width. If the width is prefixed with 0, then zeros are padded instead of blanks. The `width` field never truncates a field. If the length of the output value exceeds the specified width, all characters are output.

The `precision` field is a non-negative number that specifies the number of characters to print, number of significant digits, or number of decimal places. The precision field can cause truncation or rounding of the output value in the case of a floating-point number as specified in the following table.

`flags`	**MeaningMeaning of the `precision` field**
`d, u, o, x, X`	The precision field is where you specify the minimum number of digits that will be included in the output value. Digits are not truncated if the number of digits in an argument exceeds that defined in the precision field. If the number of digits in the argument is less than the precision field, the output value is padded on the left with zeros.
`f`	The precision field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
`e, E`	The precision field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
`g`	The precision field is where you specify the maximum number of significant digits in the output value.
`c, C`	The precision field has no effect on these field types.
`s`	The precision field is where you specify the maximum number of characters in the output value. Excess characters are not output.

Description	The optional characters `l` or `L` may immediately precede `conversion_type` to respectively specify long versions of the integer types `d, i, u, o, x,` and `X`. You must ensure that the argument type matches that of the format specification. You can use type casts to ensure that the proper type is passed to `sprintf`.

sprintl

Prototype	`sprintl(char *wh, const char *f,...);`
Returns	The function returns the number of characters actually written to destination string.
Description	The same as sprintf, except it doesn't support float-type numbers.

sprinti

Prototype	`sprinti(char *wh, const char *f,...);`
Returns	The function returns the number of characters actually written to destination string.
Description	The same as sprintf, except it doesn't support long integers and float-type numbers.

Library Example

This is a demonstration of the standard C library sprintf routine usage. Three different representations of the same floating poing number obtained by using the sprintf routine are sent via UART.

```
double ww = -1.2587538e+1;
char  buffer[15];

// Function for sending string to UART
void UartWriteText(char *txt) {
  while(*txt)
    UART1_Write(*txt++);
}

// Function for sending const string to UART
void UartWriteConstText(const char *txt) {
  while(*txt)
    UART1_Write(*txt++);
}

void main(){

  UART1_Init(4800);           // Initialize UART module at 4800 bps
  Delay_ms(10);

   UartWriteConstText("Floating point number representation"); //
Write message on UART

  sprintf(buffer, "%12e", ww);   // Format ww and store it to buffer
  UartWriteConstText("\r\ne format:");      // Write message on UART
  UartWriteText(buffer);                     // Write buffer on UART

  sprintf(buffer, "%12f", ww);   // Format ww and store it to buffer
  UartWriteConstText("\r\nf format:");      // Write message on UART
  UartWriteText(buffer);                     // Write buffer on UART

  sprintf(buffer, "%12g", ww);  // Format ww and store it to buffer
  UartWriteConstText("\r\ng format:");      // Write message on UART
  UartWriteText(buffer);                     // Write buffer on UART
}
```

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. Type declaration is contained in timelib.h which can be found in the mikroC PRO for PIC Time Library Demo example folder.

Library Routines

- Time_dateToEpoch
- Time_epochToDate
- Time_dateDiff

Time_dateToEpoch

Prototype	`long Time_dateToEpoch(TimeStruct *ts);`
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

```
#include         "timelib.h"
...
TimeStruct       ts1;
long     epoch;
...
/*
 * what is the epoch of the date in ts ?
 */
epoch = Time_dateToEpoch(&ts1);
```

Time_epochToDate

Prototype	`void Time_epochToDate(long e, TimeStruct *ts);`
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

```
#include        "timelib.h"
...
TimeStruct       ts2;
long     epoch;
...
/*
 * what date is epoch 1234567890 ?
 */
epoch = 1234567890;
Time_epochToDate(epoch, &ts2);
```

Time_dateDiff

Prototype	`long Time_dateDiff(TimeStruct *t1, TimeStruct *t2);`
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. Result is positive if `t1` is before `t2`, result is null if `t1` is the same as `t2` and result is 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) **Note**: This function is implemented as macro in the timelib.h header file which can be found in the mikroC PRO for PIC Time Library Demo example folder.
Requires	Nothing.
Example	

```
#include        "timelib.h"
...
TimeStruct       ts1, ts2;
long     diff;
...
/*
 * how many seconds between these two dates contained in ts1 and
ts2 buffers?
 */
 diff = Time_dateDiff(&ts1, &ts2);
```

Library Example

This example demonstrates Time Library usage.

```
#include        "timelib.h"

TimeStruct ts1, ts2;
long epoch;
long diff;

void main() {
  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);
}
```

TRIGONOMETRY LIBRARY

The *mikroC PRO for PIC* 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	`int sinE3(unsigned angle_deg);`
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	see below

```
int res;
...
res = sinE3(45);   // result is 707
```

cosE3

Prototype	`int cosE3(unsigned angle_deg);`
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	`int res;`
	`...`
	`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