

mikroElektronika

Development tools - Books - Compilers

www.mikroe.com

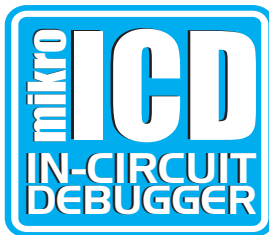
C COMPILER FOR MICROCHIP PIC MICROCONTROLLERS

mikroC

Making it simple

manual

User's



**SUPPORTED
from V6.0**

Develop your applications quickly and easily with the world's most intuitive C compiler for PIC Microcontrollers (families PIC12, PIC16, and PIC18).

Highly sophisticated IDE provides the power you need with the simplicity of a Windows based point-and-click environment.

With useful implemented tools, many practical code examples, broad set of built-in routines, and a comprehensive Help, mikroC makes a fast and reliable tool, which can satisfy needs of experienced engineers and beginners alike.

Reader's note**DISCLAIMER:**

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

HIGH RISK ACTIVITIES

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

Copyright © mikroElektronika 2003 - 2006.

This manual covers mikroC version 6.2.0.1 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
- Code sample
- Description of a bug

CONTACT US:

mikroElektronika

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

Fax: + 381 (11) 30 66 379

Web: www.mikroe.com

E-mail: office@mikroe.com

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

mikroC User's manual

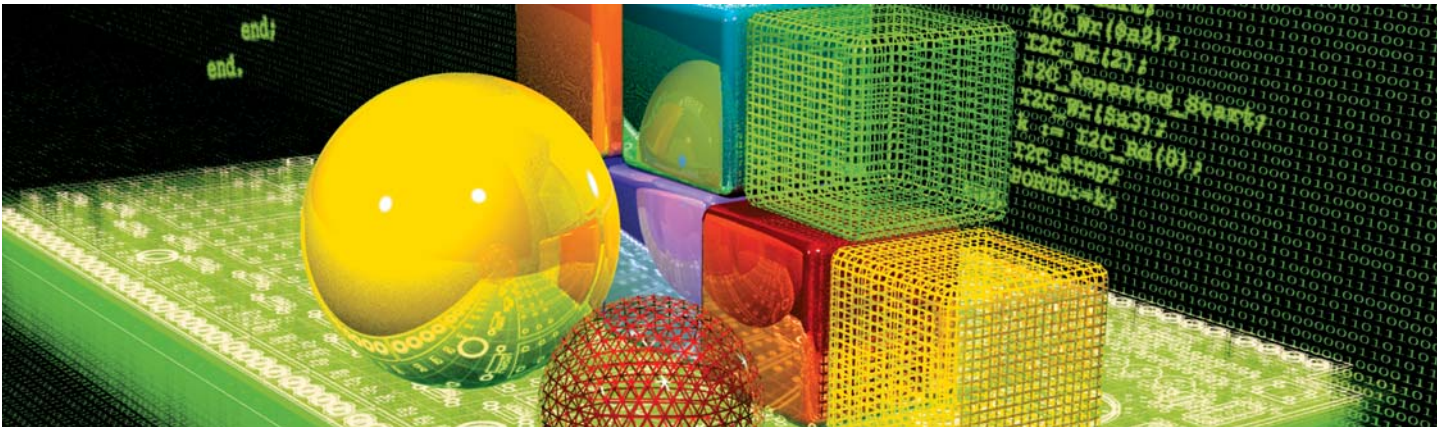


Table of Contents

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

CHAPTER 1: mikroC IDE	1
Quick Overview	1
Code Editor	3
Code Explorer	6
Debugger	7
Error Window	11
Statistics	12
Integrated Tools	15
Keyboard Shortcuts	19
CHAPTER 2: Building Applications	21
Projects	22
Source Files	24
Search Paths	24
Managing Source Files	25
Compilation	27
Output Files	27
Assembly View	27
Error Messages	28
CHAPTER 3: mikroC Language Reference	31
PIC Specifics	32
mikroC Specifics	34
ANSI Standard Issues	34
Predefined Globals and Constants	35
Accessing Individual Bits	35
Interrupts	36
Linker Directives	37
Code Optimization	38
Indirect Function Calls	39
Lexical Elements	40
mikro ICD (In-Circuit Debugger)	42
mikro ICD Debugger Options	44
mikro ICD Debugger Example	45
mikro ICD Overview	49
Tokens	53
Constants	54
Integer Constants	54
Floating Point Constants	56
Character Constants	57
String Constants	59

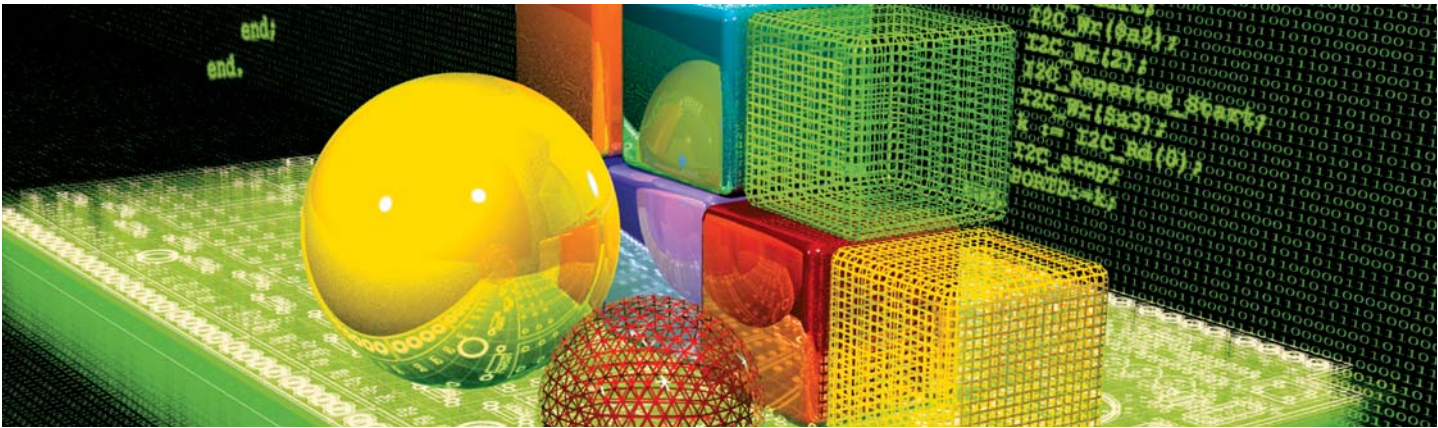
Enumeration Constants	60
Pointer Constants	60
Constant Expressions	60
Keywords	61
Identifiers	62
Punctuators	63
Objects and Lvalues	67
Scope and Visibility	69
Name Spaces	71
Duration	72
Types	74
Fundamental Types	75
Arithmetic Types	75
Enumeration Types	77
Void Type	79
Derived Types	80
Arrays	80
Pointers	83
Function Pointer	85
Pointer Arithmetic	87
Structures	91
Unions	96
Bit Fields	97
Types Conversions	99
Standard Conversions	99
Explicit Typcasting	101
Declarations	102
Linkage	104
Storage Classes	106
Type Qualifiers	108
Typedef Specifier	109
asm Declaration	110
Initialization	112
Functions	113
Function Declaration	113
Function Prototypes	114
Function Definition	115
Function Reentrancy	115
Function Calls	116
Ellipsis Operator	118
Operators	119
Precedence and Associativity	119
Arithmetic Operators	121
Relational Operators	123
Bitwise Operators	124

Logical Operators	126
Conditional Operator ? :	128
Assignment Operators	129
sizeof Operator	131
Expressions	132
Statements	134
Labeled Statements	134
Expression Statements	135
Selection Statements	135
Iteration Statements	138
Jump Statements	141
Compound Statements (Blocks)	143
Preprocessor	144
Preprocessor Directives	144
Macros	145
File Inclusion	149
Preprocessor Operators	150
Conditional Compilation	151

CHAPTER 4: mikroC Libraries **155**

Built-in Routines	156
Library Routines	160
ADC Library	162
CAN Library	164
CANSPI Library	176
Compact Flash Library	185
Compact Flash Flash FAT Library v2.xx	195
EEPROM Library	198
Ethernet Library	200
SPI Ethernet Library	212
Flash Memory Library	224
I2C Library	227
Keypad Library	232
LCD Library (4-bit interface)	236
LCD Custom Library (4-bit interface)	242
LCD8 Library (8-bit interface)	248
Graphic LCD Library	252
T6963C Graphic LCD Library	263
Manchester Code Library	279
Multi Media Card Library	285
OneWire Library	296
PS/2 Library	300
PWM Library	303

RS-485 Library	307
Software I2C Library	313
Software SPI Library	317
Software UART Library	320
Sound Library	323
SPI Library	325
USART Library	329
USB HID Library	333
Util Library	338
ANSI C Ctype Library	339
ANSI C Math Library	343
ANSI C Stdlib Library	349
ANSI C String Library	353
Conversions Library	359
Trigonometry Library	363
Sprint Library	364
SPI Graphic LCD Library	369
Port Expander Library	380
SPI LCD Library	388
SPI LCD8 (8-bit interface) Library	393
Spi T6963C Graphic LCD Library	398
Setjmp Library	414
Time Library	416
Contact Us	419

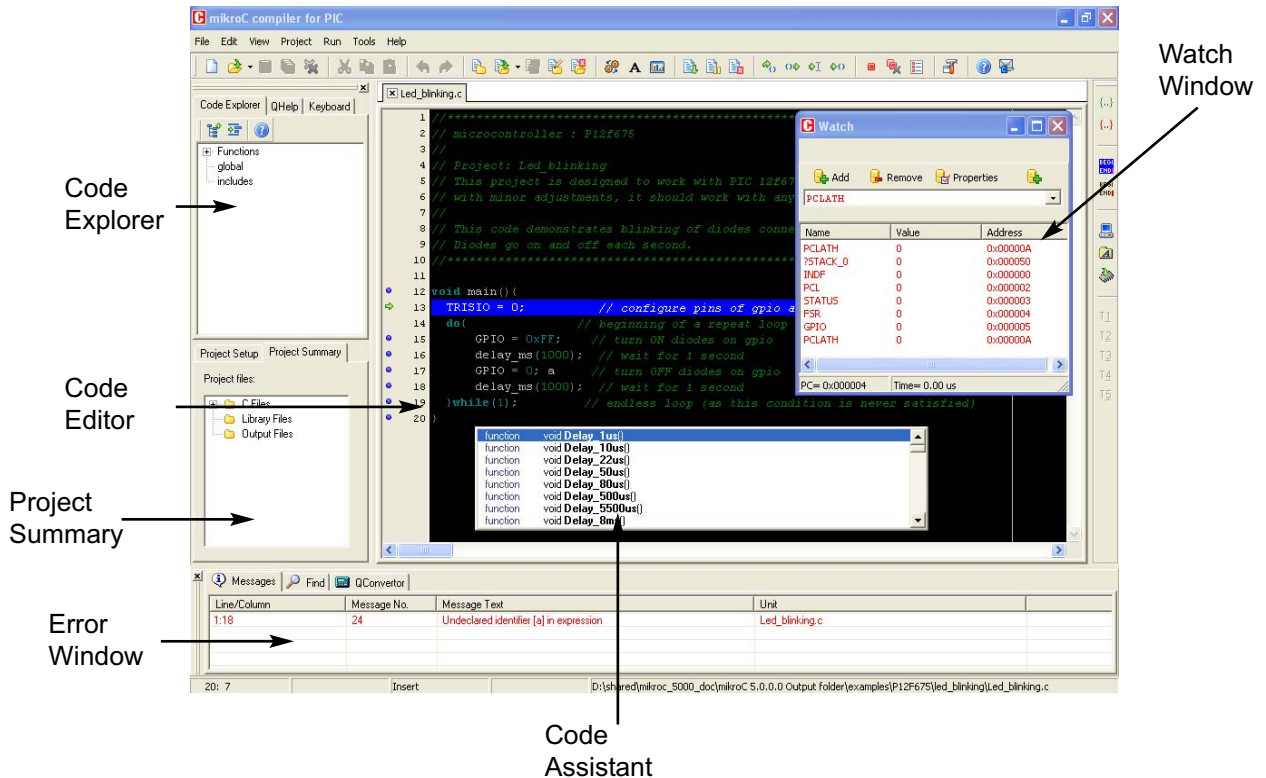


mikroC IDE

QUICK OVERVIEW

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

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 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.



mikroC allows you to quickly develop and deploy complex applications:

- Write your C source code using the highly advanced Code Editor
- Use the included mikroC libraries to dramatically speed up the development: data acquisition, memory, displays, conversions, communications...
- Monitor your program structure, variables, and functions in the Code Explorer. Generate commented, human-readable assembly, and standard HEX compatible with all programmers.
- Inspect program flow and debug executable logic with the integrated Debugger. Get detailed reports and graphs on code statistics, assembly listing, calling tree...
- We have provided plenty of examples for you to expand, develop, and use as building bricks in your projects.

CODE EDITOR

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

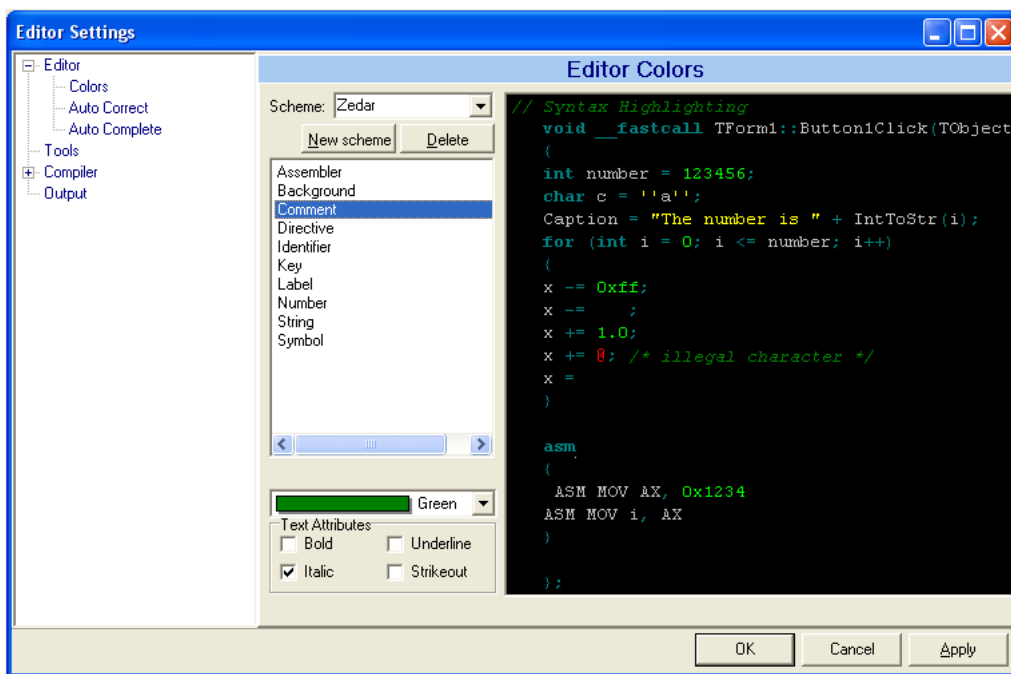
Advanced Editor features include:

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

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



Tools Icon.



Code Assistant [CTRL+SPACE]

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



Parameter Assistant [CTRL+SHIFT+SPACE]

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



Code Template [CTR+J]

You can insert the Code Template by typing the name of the template (for instance, *whileb*), then press CTRL+J, and the Code Editor will automatically generate the code. Or you can click a button from the Code toolbar and select a template from the list.

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

Auto Correct

The Auto Correct feature corrects common typing mistakes. To access the list of recognized typos, select Tools > Options from the drop-down menu, or click the Tools Icon, and then select the Auto Correct Tab. You can also add your own preferences to the list.



Comment /
Uncomment Icon.

Comment/Uncomment

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

Bookmarks

Bookmarks make navigation through large code easier.

CTRL+<number> : Go to a bookmark

CTRL+SHIFT+<number> : Set a bookmark

Goto Line

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

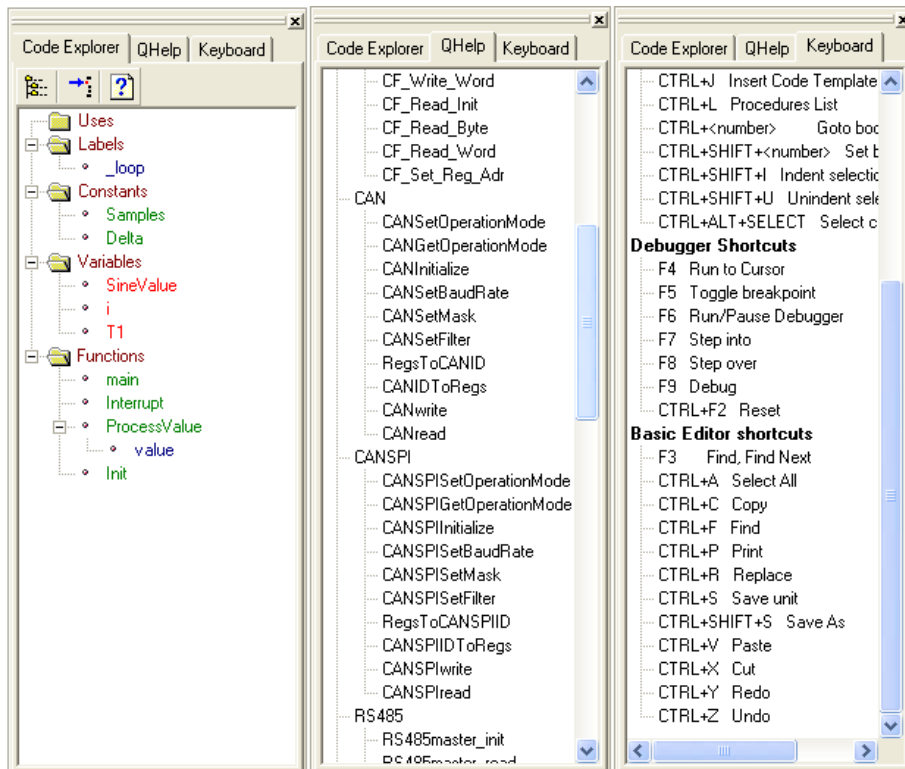
CODE EXPLORER

The Code Explorer is placed to the left of the main window by default, and gives a clear view of every declared item in the source code. You can jump to a declaration of any item by clicking it, or by clicking the Find Declaration icon. To expand or collapse treeview in Code Explorer, use the Collapse/Expand All icon.



Collapse/Expand All Icon.

Also, two more tabs are available in Code Explorer. QHelp Tab lists all the available built-in and library functions, for a quick reference. Double-clicking a routine in QHelp Tab opens the relevant Help topic. Keyboard Tab lists all the available keyboard shortcuts in mikroC.



DEBUGGER



Start Debugger

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

The Debugger simulates program flow and execution of instruction lines, but does not fully emulate PIC device behavior: it does not update timers, interrupt flags, etc.

After you have successfully compiled your project, you can run the Debugger by selecting Run > Debug from the drop-down menu, or by clicking the Debug Icon . Starting the Debugger makes more options available: Step Into, Step Over, Run to Cursor, etc. Line that is to be executed is color highlighted.



Pause Debugger

Debug [F9]

Start the Debugger.

Run/Pause Debugger [F6]

Run or pause the Debugger.



Step Into

Step Into [F7]

Execute the current 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.



Step Over

Step Over [F8]

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.



Step Out

Step Out [Ctrl+F8]

Execute the current C (single- or multi-cycle) instruction, then halt. If the instruction is within a routine, execute the instruction and halt at the first instruction following the call.



Run to Cursor

Run to cursor [F4]

Executes all instructions between the current instruction and the cursor position.



Toggle
Breakpoint.

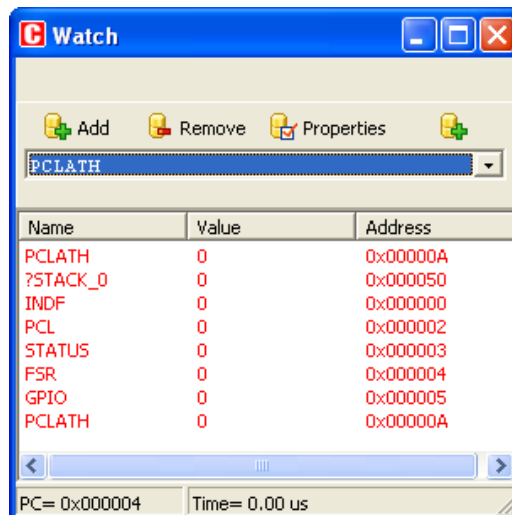
Toggle Breakpoint [F5]

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

Watch Window

Variables

The Watch Window allows you to monitor program items while running your program. It displays variables and special function registers of PIC MCU, their addresses and values. Values are updated as you go through the simulation.

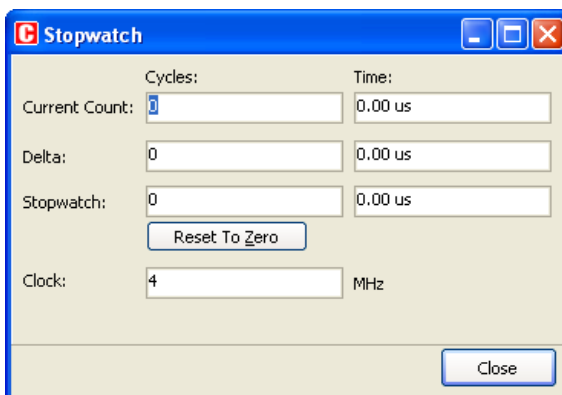


Double clicking one of the items opens a window in which you can assign a new value to the selected variable or register and change number formatting.

Stopwatch Window

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

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



View RAM Window

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

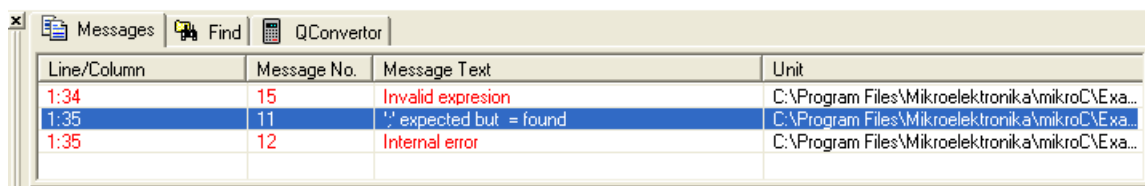
The View RAM Window displays the map of PIC's RAM, with recently changed items colored red. You can change value of any field by double-clicking it.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

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



Line/Column	Message No.	Message Text	Unit
1:34	15	Invalid expresion	C:\Program Files\Mikroelektronika\mikroC\Exa...
1:35	11	'' expected but = found	C:\Program Files\Mikroelektronika\mikroC\Exa...
1:35	12	Internal error	C:\Program Files\Mikroelektronika\mikroC\Exa...

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

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

STATISTICS

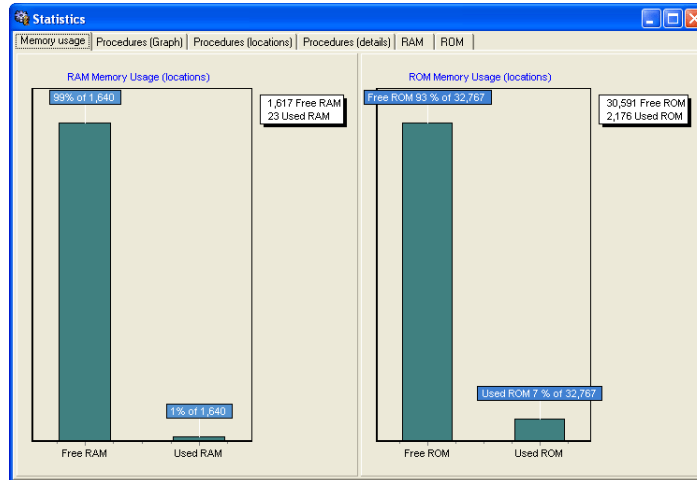


Statistics Icon.

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

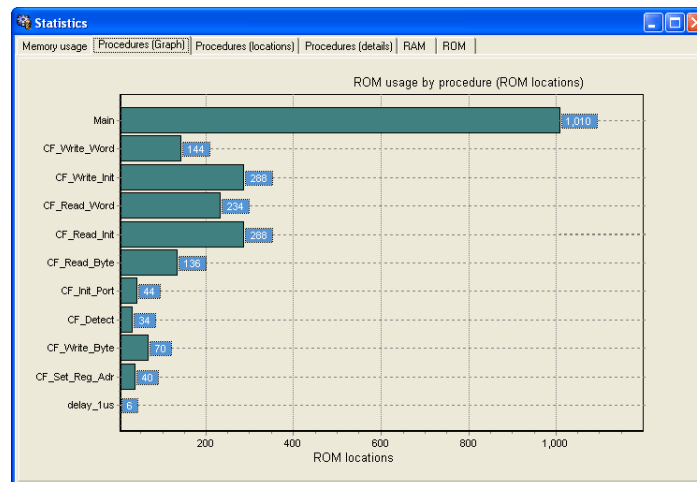
Memory Usage Window

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



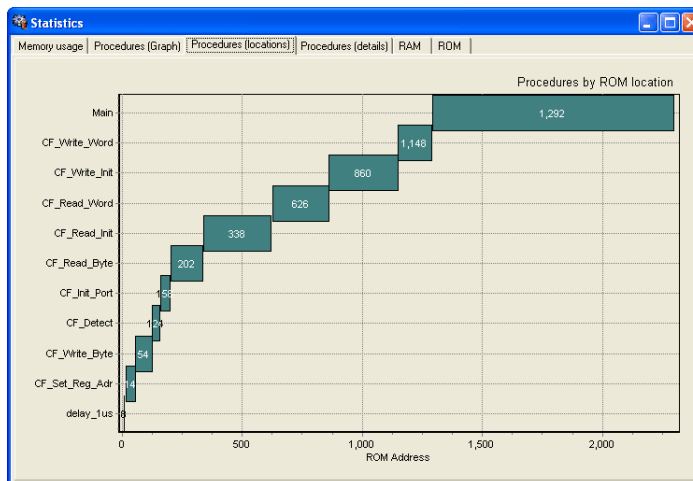
Procedures (Graph) Window

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



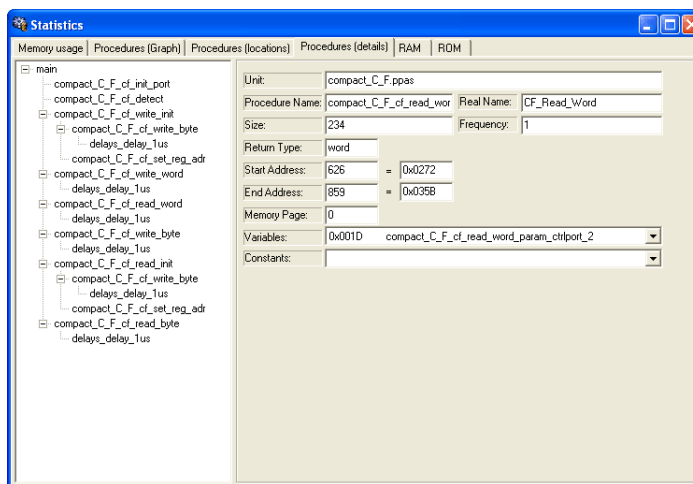
Procedures (Locations) Window

Displays how functions are distributed in microcontroller's memory.



Procedures (Details) Window

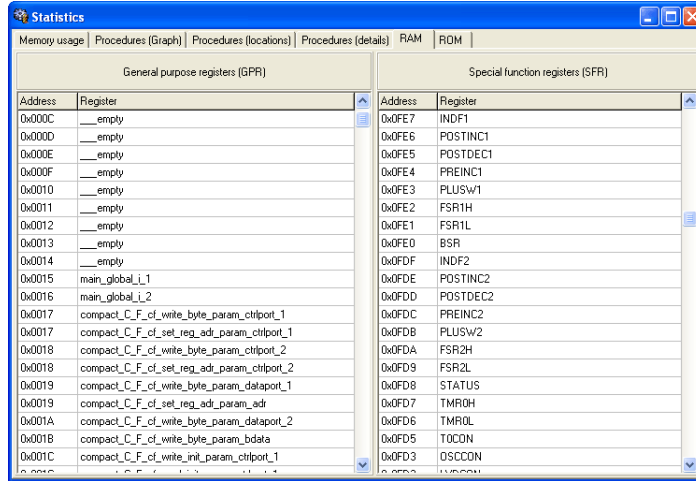
Displays complete call tree, along with details for each function:



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

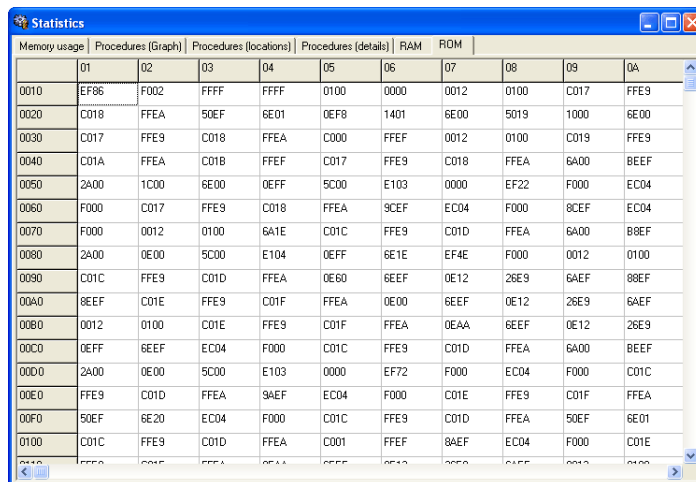
RAM Window

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



ROM Window

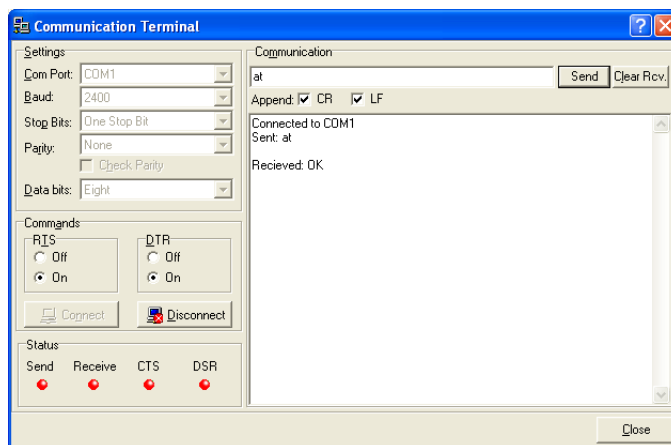
Lists op-codes and their addresses in form of a human readable hex code.



INTEGRATED TOOLS

USART Terminal

mikroC includes the USART (Universal Synchronous Asynchronous Receiver Transmitter) communication terminal for RS232 communication. You can launch it from the drop-down menu Tools > Terminal or by clicking the Terminal icon.



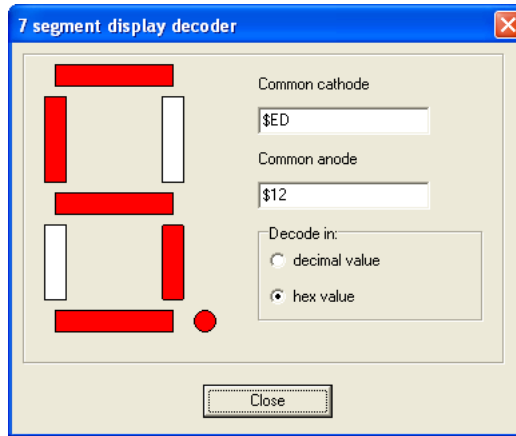
ASCII Chart

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

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

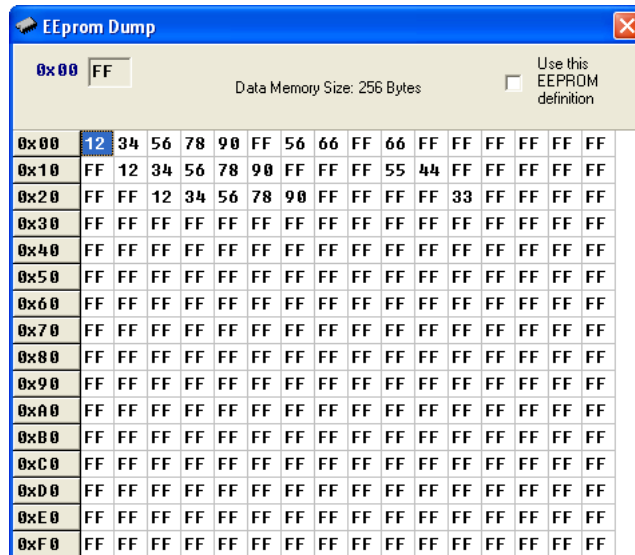
7 Segment Display Decoder

The 7seg Display Decoder is a convenient visual panel which returns decimal/hex value for any viable combination you would like to display on 7seg. Click on the parts of 7 segment image to get the desired value in the edit boxes. You can launch it from the drop-down menu Tools > 7 Segment Display.



EEPROM Editor

EEPROM Editor allows you to easily manage EEPROM of PIC microcontroller.



mikroBootloader

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

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

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

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

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

Integrating User Code and Boot Code

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

KEYBOARD SHORTCUTS

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

IDE Shortcuts

F1	Help
CTRL+N	New Unit
CTRL+O	Open
CTRL+F9	Compile
CTRL+F11	Code Explorer on/off
CTRL+SHIFT+F5	View breakpoints

Basic Editor shortcuts

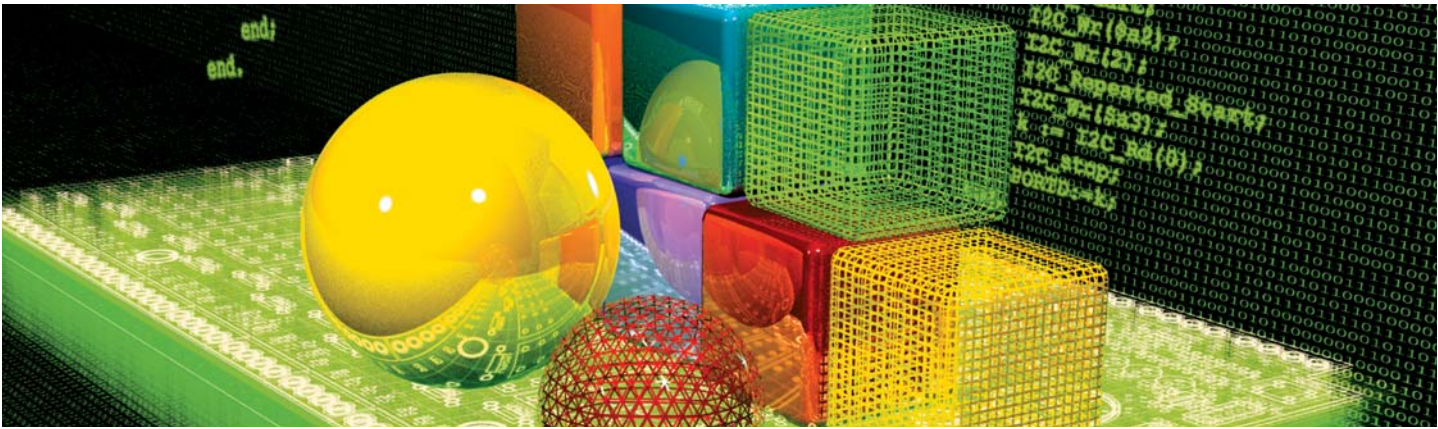
F3	Find, Find Next
CTRL+A	Select All
CTRL+C	Copy
CTRL+F	Find
CTRL+P	Print
CTRL+R	Replace
CTRL+S	Save unit
CTRL+SHIFT+S	Save As
CTRL+V	Paste
CTRL+X	Cut
CTRL+Y	Redo
CTRL+Z	Undo

Advanced Editor shortcuts

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

Debugger Shortcuts

F4	Run to Cursor
F5	Toggle breakpoint
F6	Run/Pause Debugger
F7	Step into
F8	Step over
F9	Debug
CTRL+F2	Reset



Building Applications

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

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

PROJECTS

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

Project file carries the following information:

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



New Project.

New Project

The easiest way to create project is by means of New Project Wizard, drop-down menu Project > New Project. Just fill the dialog with desired values (project name and description, location, device, clock, config word) and mikroC will create the appropriate project file. Also, an empty source file named after the project will be created by default.



Edit Project.

Editing Project

Later, you can change project settings from drop-down menu Project > Edit Project. You can rename the project, modify its description, change chip, clock, config word, etc. To delete a project, simply delete the folder in which the project file is stored.



Add to Project.

Add/Remove Files from Project

Project can contain any number of source files (extension `.c`). The list of relevant source files is stored in the project file (extension `.ppc`). To add source file to your project, select Project > Add to Project from drop-down menu. Each added source file must be self-contained, i.e. it must have all the necessary definitions after preprocessing. To remove file(s) from your project, select Project > Remove from Project from drop-down menu.

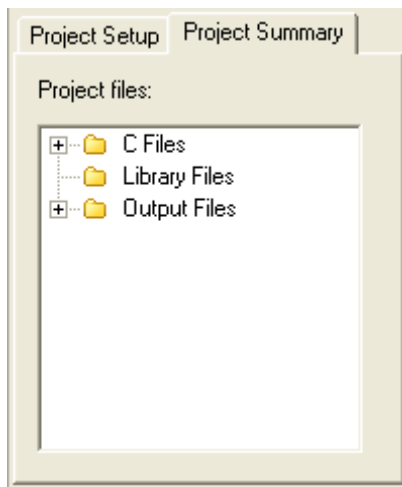


Remove from Project.

Note: For inclusion of header files, use the preprocessor directive `#include`.

Extended functionality of the Project Files tab

By using the Project Files' new features, you can reach all the output files (.lst, .asm) by a single click. You can also include in project the library files (.mcl), for libraries, either your own or compiler default, that are project-specific.



Libraries (.mcl) now have different, more compact format, compared to mikroC version 2. This, however, means that library formats are now incompatible. The users that are making transition from version 2 to 5, must re-build all their previously written libraries in order to use them in the new version. All the source code written and tested in previous versions should compile correctly on version 5.0, except for the `asm{}` blocks, which are commented in the asm section of help.

SOURCE FILES

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

Use the preprocessor directive `#include` to include headers. Do not rely on preprocessor to include other source files — see Projects for more information.

Search Paths

Paths for source files (`.c`)

You can specify your own custom search paths. This can be configured by selecting Tools > Options from drop-down menu and then tab window Advanced.

In project settings, you can specify either absolute or relative path to the source file. If you specify a relative path, mikroC will look for the file in following locations, in this particular order:

1. the project folder (folder which contains the project file `.ppc`),
2. your custom search paths,
3. mikroC installation folder > “uses” folder.

Paths for Header Files (.h)

Header files are included by means of preprocessor directive `#include`. If you place an explicit path to the header file in preprocessor directive, only that location will be searched.

You can specify your own custom search paths: select **Tools > Options** from the drop-down menu and then select **Search Path**.

In project settings, you can specify either absolute or relative path to the header. If you specify a relative path, mikroC will look for the file in following locations, in this particular order:

1. the project folder (folder which contains the project file `.ppc`),
2. mikroC installation folder > “include” folder,
3. your custom search paths.

Managing Source Files



New File.

Creating a new source file

To create a new source file, do the following:

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

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



Open File Icon.

Opening an Existing File

Select File > Open from drop-down menu, or press CTRL+O, or click the Open File icon. The Select Input File dialog opens. In the dialog, browse to the location of the file you want to open and select it. Click the Open button.

The selected file is displayed in its own tab. If the selected file is already open, its current Editor tab will become active.



Print File Icon.

Printing an Open File

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



Save File Icon.

Saving File

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



Save File As.

Saving File Under a Different Name

Make sure that window containing the file you want to save is the active window. Select File > Save As from drop-down menu, or press SHIFT+CTRL+S. The New File Name dialog will be displayed. In the dialog, browse to the folder where you want to save the file. In the File Name field, modify the name of the file you want to save. Click the Save button.



Close File.

Closing a File

Make sure that tab containing the file you want to close is the active tab. Select File > Close from drop-down menu, or right click the tab of the file you want to close in Code Editor. If the file has been changed since it was last saved, you will be prompted to save your changes.

COMPILATION



Compile Icon.

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

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

Output Files

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

Intel HEX file (.hex)

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

Binary mikro Compiled Library (.mcl)

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

List File (.lst)

Overview of PIC memory allotment: instruction addresses, registers, routines, etc.

Assembler File (.asm)

Human readable assembly with symbolic names, extracted from the List File.

Assembly View



View Assembly Icon.

After compiling your program in mikroC, you can click View Assembly Icon or select Project > View Assembly from drop-down menu to review generated assembly code (.asm file) in a new tab window. Assembly is human readable with symbolic names. All physical addresses and other information can be found in Statistics or in list file (.lst).

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

ERROR MESSAGES

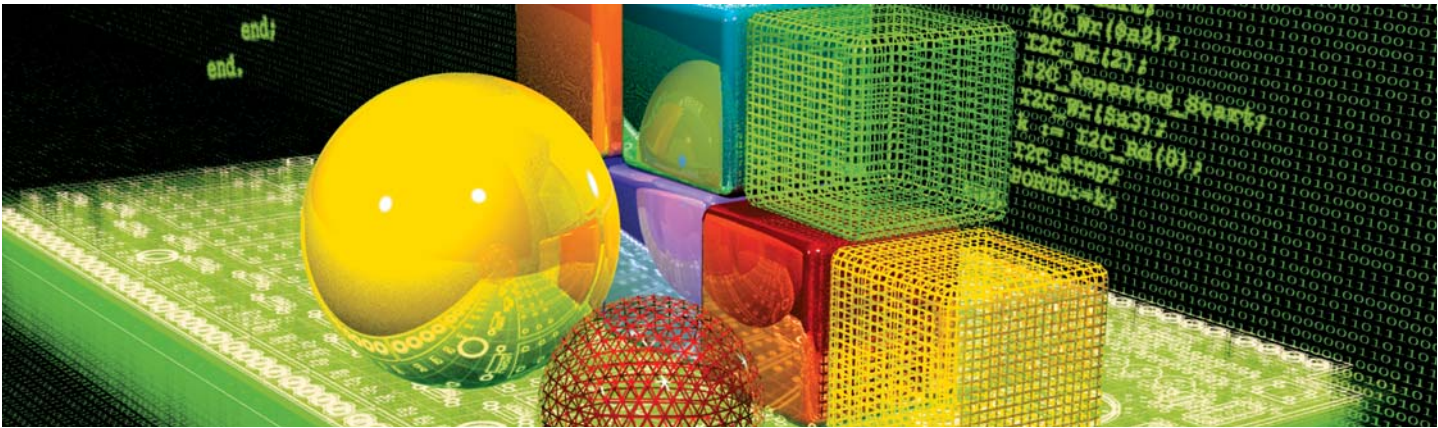
Error Messages

- Specifier needed
- Invalid declarator
- Expected '(' or identifier
- Integer const expected
- Array dimension must be greater than 0
- Local objects cannot be extern
- Declarator error
- Bad storage class
- Arguments cannot be of void type
- Specifier/qualifier list expected
- Address must be greater than 0
- Identifier redefined
- case out of switch
- default label out of switch
- switch exp. must evaluate to integral type
- continue outside of loop
- break outside of loop or switch
- void func cannot return values
- Unreachable code
- Illegal expression with void
- Left operand must be pointer
- Function required
- Too many chars
- Undefined struct
- Nonexistent field
- Aggregate init error
- Incompatible types
- Identifier redefined
- Function definition not found
- Signature does not match
- Cannot generate code for expression
- Too many initializers of subaggregate
- Nonexistent subaggregate
- Stack Overflow: func call in complex expression
- Syntax Error: expected %s but %s found
- Array element cannot be function
- Function cannot return array

- Inconsistent storage class
- Inconsistent type
- %s tag redefined
- Illegal typecast
- %s is not a valid identifier
- Invalid statement
- Constant expression required
- Internal error %s
- Too many arguments
- Not enough parameters
- Invalid expression
- Identifier expected, but %s found
- Operator [%s] not applicable to this 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

Compiler Warning Messages

- Highly inefficient code: func call in complex expression
- Inefficient code: func call in complex expression



mikroC Language Reference

C offers unmatched power and flexibility in programming microcontrollers. mikroC adds even more power with an array of libraries, specialized for PIC HW modules and communications. This chapter should help you learn or recollect C syntax, along with the specifics of programming PIC microcontrollers. If you are experienced in C programming, you will probably want to consult mikroC Specifics first.

PIC SPECIFICS

In order to get the most from your mikroC 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 is capable of handling very complex data types, PIC may choke on them, especially if you are working on some of the older models. This can dramatically increase the time needed for performing even simple operations. Universal advice is to use the smallest possible type in every situation. It applies to all programming in general, and doubly so with microcontrollers.

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

Nested Calls Limitations

Nested call represents a function call within function body, either to itself (recursive calls) or to another function. Recursive function calls are supported by mikroC 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 limits the number of non-recursive nested calls to:

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

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.

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 3 or 4 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 linker directive `absolute`. Variables that are approached only directly should be moved to banks 3 and 4 for increased efficiency.

mikroC SPECIFICS

ANSI Standard Issues

Divergence from the ANSI C Standard

mikroC diverges from the ANSI C standard in few areas. Some of these modifications are improvements intended to facilitate PIC programming, while others are result of PICmicro hardware limitations:

Function recursion is supported with limitations because of no easily-usable stack and limited memory. See PIC Specifics.

Pointers to variables and pointers to constants are not compatible, i.e. no assigning or comparison is possible between the two.

mikroC treats identifiers declared with const qualifier as “true constants” (C++ style). This allows using const objects in places where ANSI C would expect a constant expression. If aiming at portability, use the traditional preprocessor defined constants. See Type Qualifiers and Constants.

mikroC allows C++ style single-line comments using two adjacent slashes (`//`).

Features under construction: anonymous structures and unions.

Implementation-defined Behavior

Certain sections of the ANSI standard have implementation-defined behavior. This means that the exact behavior of some C code can vary from compiler to compiler. Throughout the help are sections describing how the mikroC compiler behaves in such situations. The most notable specifics include: Floating-point Types, Storage Classes, and Bit Fields.

Predefined Globals and Constants

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

All PIC SFR registers are implicitly declared as global variables of `volatile unsigned short`. These identifiers have external linkage, and are visible in the entire project. When creating a project, mikroC will include an appropriate `.def` file, containing declarations of available SFR and constants (such as `TOIE`, `INTF`, etc). Identifiers are all in uppercase, identical to nomenclature in Microchip datasheets. For the complete set of predefined globals and constants, look for “Defs” in your mikroC installation folder, or probe the Code Assistant for specific letters (Ctrl+Space in Editor).

Accessing Individual Bits

mikroC allows you to access individual bits of 8-bit variables, types `char` and `unsigned short`. Simply use the direct member selector (`.`) with a variable, followed by one of identifiers `F0`, `F1`, ..., `F7`. For example:

```
// If RB0 is set, set RC0:  
if (PORTB.F0) PORTC.F0 = 1;
```

There is no need for any special declarations; this kind of selective access is an intrinsic feature of mikroC and can be used anywhere in the code. Identifiers `F0–F7` are not case sensitive and have a specific namespace.

Provided you are familiar with the particular chip, you can also access bits by name:

```
INTCON.TMR0F = 0; // Clear TMR0F
```

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

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

Interrupts

Interrupts can be easily handled by means of reserved word `interrupt`. mikroC implicitly declares function `interrupt` which cannot be redeclared. Its prototype is:

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

Write your own definition (function body) to handle interrupts in your application. mikroC saves the following SFR on stack when entering `interrupt` and pops them back upon return:

PIC12 and PIC16 family: `W`, `STATUS`, `FSR`, `PCLATH`

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

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

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**.

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).

Linker Directives

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

Directive `absolute`

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

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

Be careful when using `absolute` directive, as you may overlap two variables by mistake. 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,  
// so changing i changes jjjj highest byte at the same time
```

Directive `org`

Directive `org` specifies the starting address of 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 linker. Directive `org` cannot be applied to an interrupt routine.

Here is a simple example:

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

Code Optimization

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

Constant folding

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

Constant propagation

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

Copy propagation

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

Value numbering

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

"Dead code" elimination

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

Stack allocation

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

Local vars optimization

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

Better code generation and local optimization

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

Indirect Function Calls

If the linker encounters an indirect function call (by a pointer to function), it assumes that any one 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 function `func_name` is implemented. This module must also include declarations of all functions listed in the `called_func` list.

All functions listed in the `called_func` list will be linked if 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.

LEXICAL ELEMENTS

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

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

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

Whitespace

Whitespace is the 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, the two sequences

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

and

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

are lexically equivalent and parse identically to give the six tokens.

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

Comments

Comments are pieces of text used to annotate a program, and are technically another form of whitespace. Comments are for the programmer's use only; they are stripped from the source text before parsing. There are two ways to delineate comments: the C method and the C++ method. Both are supported by mikroC.

C comments

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

In mikroC,

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

parses as:

```
int i;
```

Note that mikroC does not support the nonportable token pasting strategy using `/**/`. For more on token pasting, refer to Preprocessor topics.

C++ comments

mikroC 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:

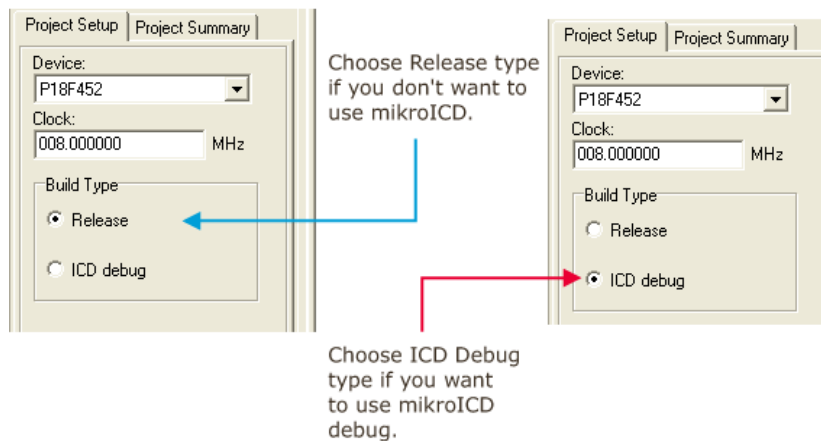
```
int i;  
int j;
```

mikro ICD (In-Circuit Debugger)

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

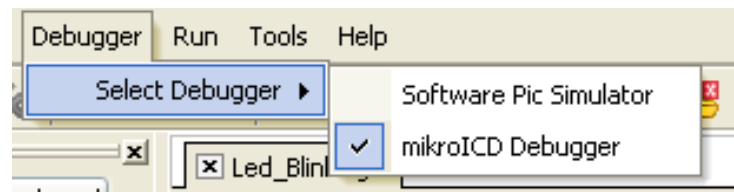
Step No. 1

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



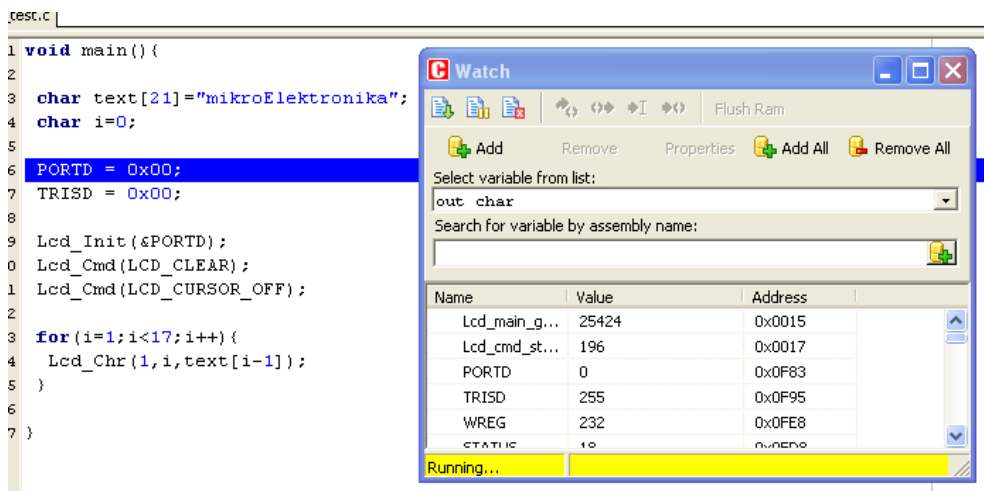
Step No. 2

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



Step No. 3

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



mikro ICD Debugger Options

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

mikro ICD Debugger Example

Step No. 1

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

```
void main(){

    char text[ 21]="mikroElektronika";
    char i=0;

    PORTD = 0x00;
    TRISD = 0x00;

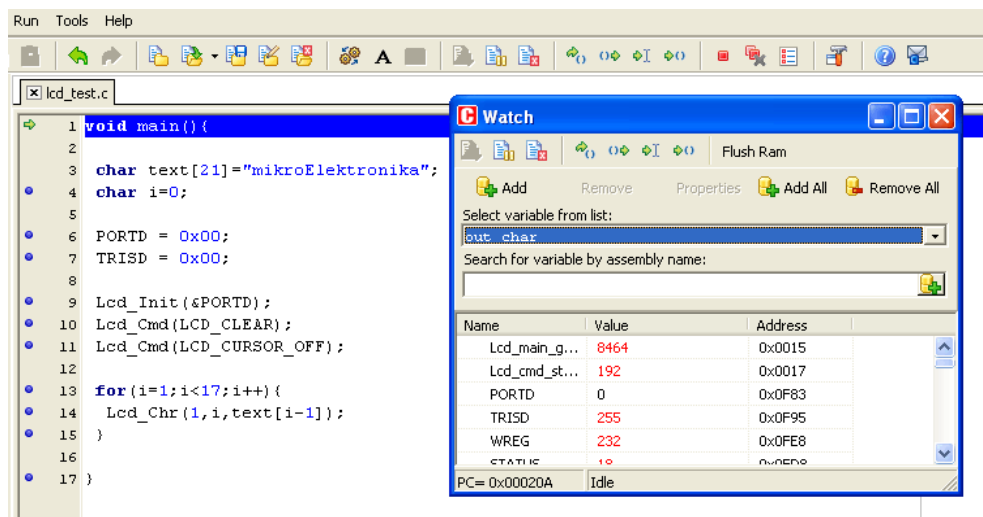
    Lcd_Init(&PORTD);
    Lcd_Cmd(LCD_CLEAR);
    Lcd_Cmd(LCD_CURSOR_OFF);

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

}
```

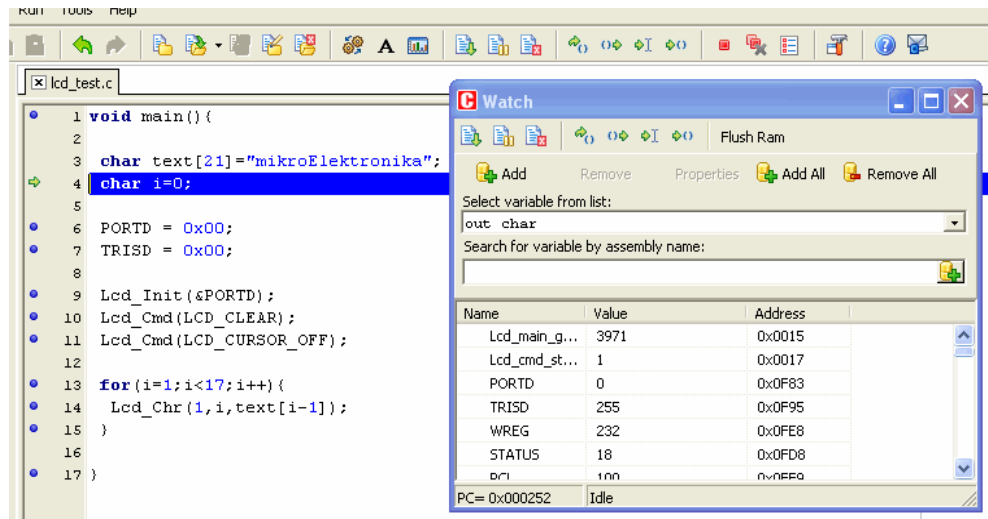
Step No. 2

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



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.



The screenshot shows the MikroC IDE with the source code for `lcd_test.c` on the left and the Watch window on the right. The code is as follows:

```

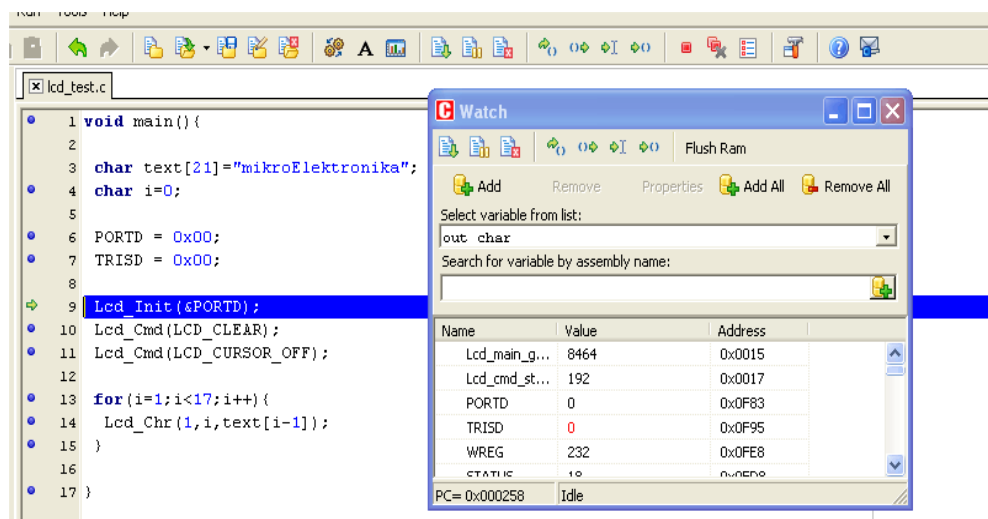
1 void main() {
2
3 char text[21]="mikroElektronika";
4 char i=0;
5
6 PORTD = 0x00;
7 TRISD = 0x00;
8
9 Lcd_Init (&PORTD);
10 Lcd_Cmd(LCD_CLEAR);
11 Lcd_Cmd(LCD_CURSOR_OFF);
12
13 for (i=1;i<17;i++) {
14   Lcd_Chrc(1,i,text[i-1]);
15 }
16
17 }

```

The Watch window displays the following variables and their values:

Name	Value	Address
Lcd_main_g...	3971	0x0015
Lcd_cmd_st...	1	0x0017
PORTD	0	0x0F83
TRISD	0	0x0F95
WREG	232	0x0FE8
STATUS	18	0x0FD8
PC	0x000252	Idle

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



The screenshot shows the MikroC IDE with the source code for `lcd_test.c` on the left and the Watch window on the right. The code is as follows:

```

1 void main() {
2
3 char text[21]="mikroElektronika";
4 char i=0;
5
6 PORTD = 0x00;
7 TRISD = 0x00;
8
9 Lcd_Init (&PORTD);
10 Lcd_Cmd(LCD_CLEAR);
11 Lcd_Cmd(LCD_CURSOR_OFF);
12
13 for (i=1;i<17;i++) {
14   Lcd_Chrc(1,i,text[i-1]);
15 }
16
17 }

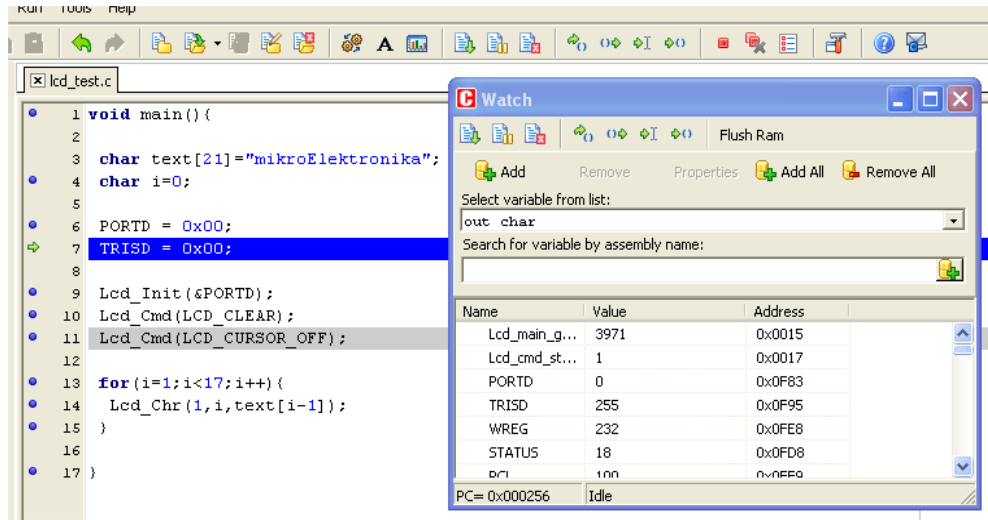
```

The Watch window displays the following variables and their values:

Name	Value	Address
Lcd_main_g...	8464	0x0015
Lcd_cmd_st...	192	0x0017
PORTD	0	0x0F83
TRISD	0	0x0F95
WREG	232	0x0FE8
STATIC	10	none
PC	0x000258	Idle

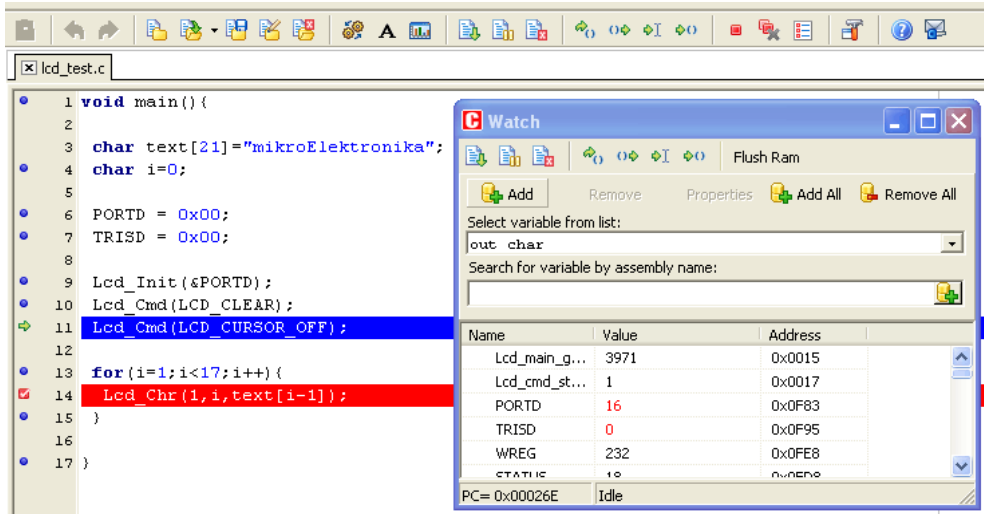
Step No. 4

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



Step No. 5

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



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 breakpoints are used, next breakpoints that will be used are software breakpoint. Those breakpoints are placed inside mikro ICD, and they simulate hardware breakpoints. Software breakpoints are much slower than hardware breakpoints. This differences between hardware and software differences are not visible in mikro ICD software but their different timings are quite notable, so it is important to know that there is two types of breakpoints.

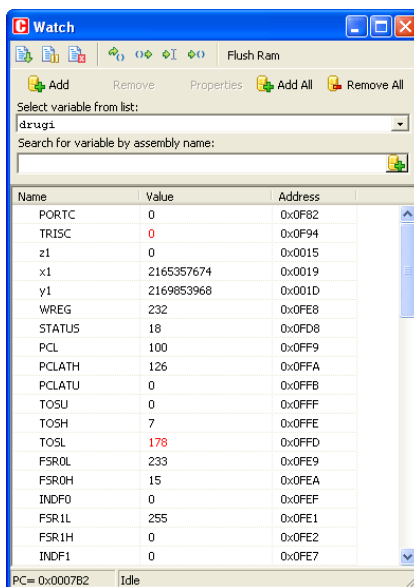


mikro ICD (In-Circuit Debugger) Overview

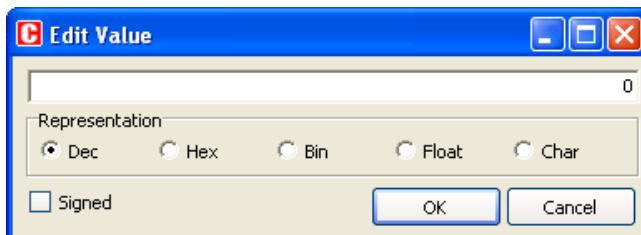
Watch Window

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

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



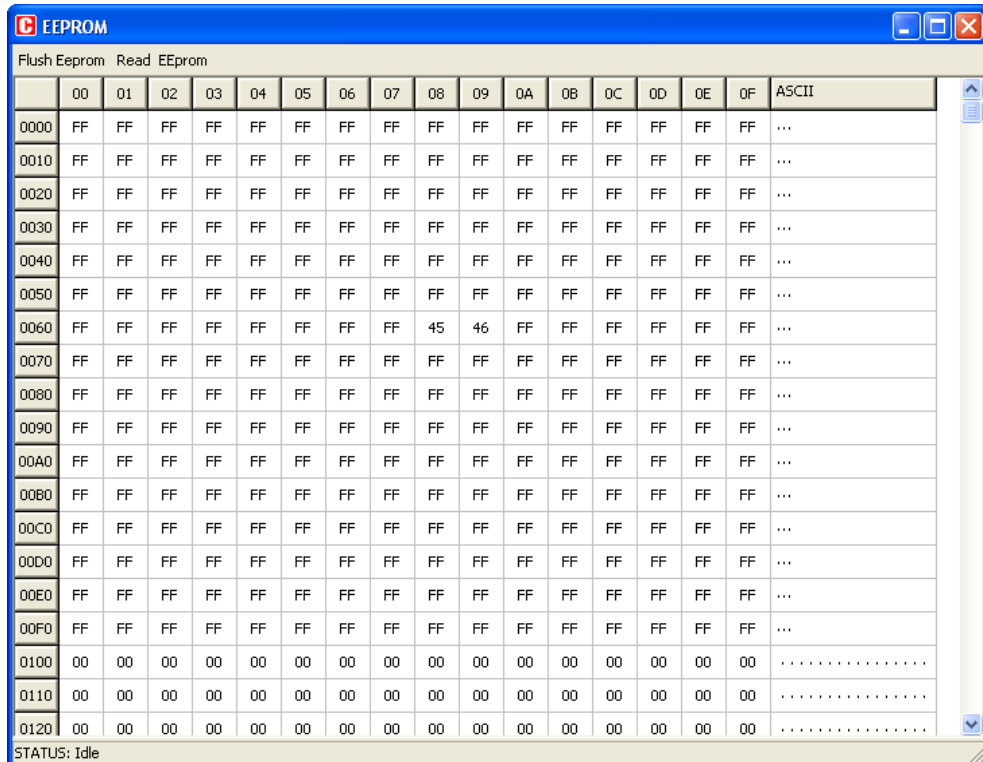
Double clicking an item opens the Edit Value window in which you can assign a new value to the selected variable/register. Also, you can change view to binary, hex, char, or decimal for the selected item.



View EEPROM Window

mikro ICD EEPROM Window is available from the drop-down menu, **View > Debug Windows > View EEPROM.**

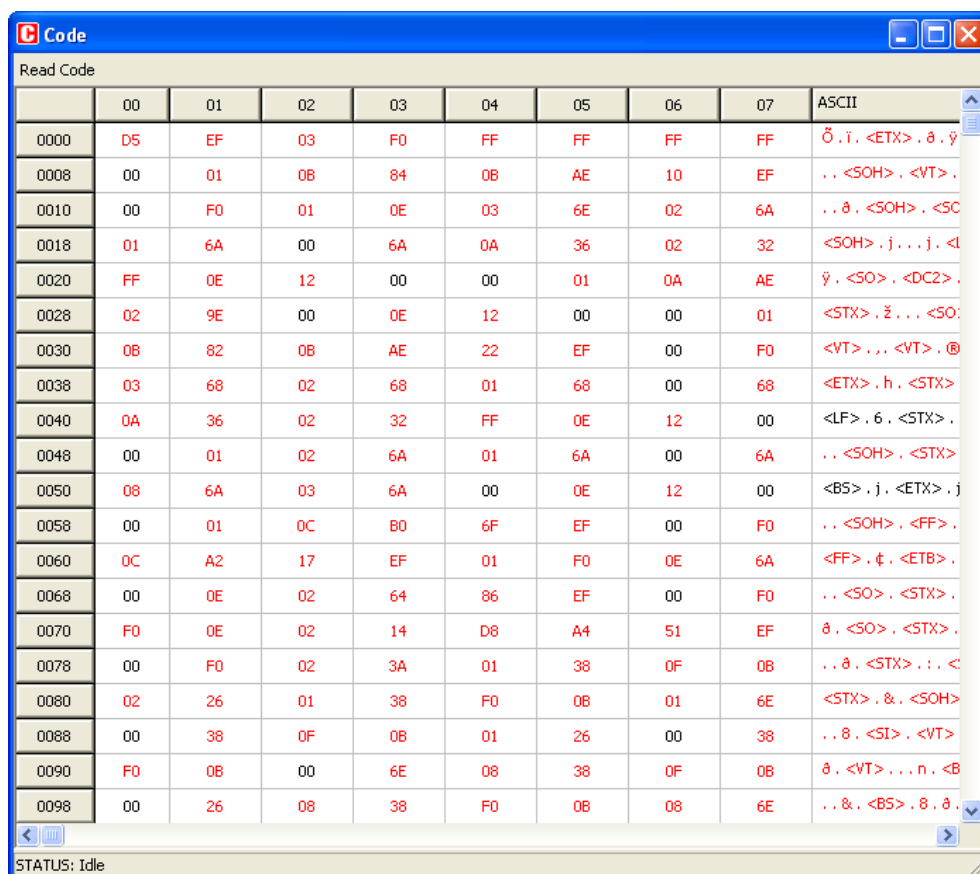
The EEPROM window shows current values written into PIC internal EEPROM memory. There are two action buttons concerning EEPROM watch window - **Flush EEPROM** and **Read EEPROM.** **Flush EEPROM** writes data from EEPROM window into PIC internal EEPROM memory. **Read EEPROM** reads data from PIC internal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



View Code Window

mikro ICD View Code Window is available from the drop-down menu, **View > Debug Windows > View Code**.

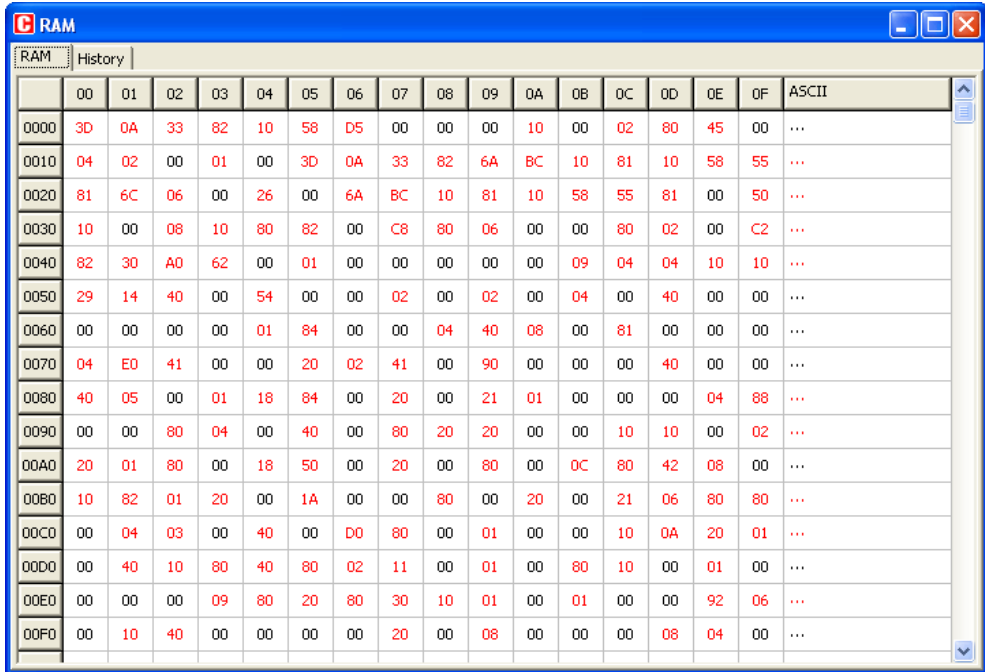
The View Code window shows code (hex code) written into PIC. There is action button concerning View Code watch window - **Read Code**. **Read Code** reads code from PIC and loads it up in View Code Window.



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	3D	0A	33	82	10	58	D5	00	00	00	10	00	02	80	45	00	...
0010	04	02	00	01	00	3D	0A	33	82	6A	BC	10	81	10	58	55	...
0020	81	6C	06	00	26	00	6A	BC	10	81	10	58	55	81	00	50	...
0030	10	00	08	10	80	82	00	C8	80	06	00	00	80	02	00	C2	...
0040	82	30	A0	62	00	01	00	00	00	00	00	09	04	04	10	10	...
0050	29	14	40	00	54	00	00	02	00	02	00	04	00	40	00	00	...
0060	00	00	00	00	01	84	00	00	04	40	08	00	81	00	00	00	...
0070	04	E0	41	00	00	20	02	41	00	90	00	00	00	40	00	00	...
0080	40	05	00	01	18	84	00	20	00	21	01	00	00	00	04	88	...
0090	00	00	80	04	00	40	00	80	20	20	00	00	10	10	00	02	...
00A0	20	01	80	00	18	50	00	20	00	80	00	0C	80	42	08	00	...
00B0	10	82	01	20	00	1A	00	00	80	00	20	00	21	06	80	80	...
00C0	00	04	03	00	40	00	D0	80	00	01	00	00	10	0A	20	01	...
00D0	00	40	10	80	40	80	02	11	00	01	00	80	10	00	01	00	...
00E0	00	00	00	09	80	20	80	30	10	01	00	01	00	00	92	06	...
00F0	00	10	40	00	00	00	00	20	00	08	00	00	00	08	04	00	...

Common Errors

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

TOKENS

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

mikroC recognizes following kinds of tokens:

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

Token Extraction Example

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

```
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 wrote the code might have intended to write

```
inter = a + (++b)
```

but it won't work that way. The compiler would parse it as the following seven tokens:

```
inter      // identifier
=          // assignment operator
a          // identifier
++         // postincrement operator
+         // addition operator
b          // identifier
;         // semicolon separator
```

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

CONSTANTS

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

mikroC 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 numeric value and the 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 the constant to be represented as a `long`. Similarly, the suffix `U` (or `u`) forces the constant to be `unsigned`. You can use both `L` and `U` suffixes on the same constant in any order or case: `ul`, `Lu`, `UL`, etc.

In the absence of any suffix (`U`, `u`, `L`, or `l`), 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 the constant has a `U` or `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 the constant has an `L` or `l` suffix, its data type will be the first of the following that can accommodate its value: `long int`, `unsigned long int`.

If the constant has both `U` and `L` suffixes, (`ul`, `lu`, `UL`, `LU`, `uL`, `Lu`, `LU`, or `UL`), its data type will be `unsigned long int`.

Decimal Constants

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.

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

< -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 a 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).

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter `e` (or `E`) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

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

mikroC limits floating-point constants to range

`±1.17549435082E38 .. ±6.80564774407E38`.

mikroC floating-point constants are of type `double`. Note that mikroC'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 C, single-character constants have data type `int`. Multi-character constants are referred to as string constants or string literals. For more information refer to String Constants.

Escape Sequences

The backslash character (`\`) is used to introduce an escape sequence, which allows the 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 the ASCII symbol or control code corresponding to that value; for example, `'\x3F'` for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type `char` (0 to 0xFF for mikroC). Larger numbers will generate the compiler error “Numeric constant too large”.

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 `\\` to represent an ASCII backslash, as used in operating system paths.

The following table shows the available escape sequences in mikroC:

Sequence	Value	Char	What it does
\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

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 the 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, e.g.

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

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

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

is an 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 assist 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 of enum declarations.

Pointer Constants

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

Constant Expressions

A constant expression is an expression that always evaluates to a constant and consists only of constants (literals) or symbolic constants. It is evaluated at compile-time and it must evaluate to a constant that is in the range of representable values for its type. Constant expressions are evaluated just as regular expressions are.

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.

You can use a constant expression anywhere that 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 the alphabetical listing of keywords in C:

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

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

IDENTIFIERS

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

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

Case Sensitivity

mikroC identifiers are *not* case sensitive at present, so that `Sum`, `sum`, and `suM` represent an equivalent identifier. However, future versions of mikroC will offer the option of activating/suspending case sensitivity. The only exceptions at present are the reserved words `main` and `interrupt` which must be written in lowercase.

Uniqueness and Scope

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

PUNCTUATORS

The mikroC punctuators (also known as separators) include brackets, parentheses, braces, comma, semicolon, colon, asterisk, equal sign, and 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 expansion:

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

For more information, refer to Expressions and Operators Precedence.

Braces

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

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

The closing brace serves as a terminator for the compound statement, so a semicolon is not required after the }, except in structure declarations. Often, the semicolon is illegal, as in

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

For more information, refer to Compound Statements.

Comma

The comma (,) separates the elements of a function argument list:

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

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

```
/* call func with two args */
func(i, j);

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


Semicolon

The 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, mikroC 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 Statements.

Colon

Use the colon (:) to indicate a labeled statement. For example:

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

Labels are discussed in Labeled Statements.

Asterisk (Pointer Declaration)

The asterisk (*) in a declaration denotes the creation of a pointer to a type:

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

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

```
i = *char_ptr;
```

For more information, see Pointers.

Equal Sign

The equal sign (=) separates variable declarations from initialization lists:

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

The equal sign is also used as the assignment operator in expressions:

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

For more information, see Assignment Operators.

Pound Sign (Preprocessor Directive)

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

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

OBJECTS AND LVALUES

Objects

An object is a specific region of memory that can hold a fixed or variable value (or set of values). To prevent confusion, this use of the word object is different from the more general term used in object-oriented languages. Our definition 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. This name can be a simple identifier, or it can be a complex expression that uniquely references the object.

Objects and Declarations

Declarations establish the 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 compiler deduces these attributes from implicit or explicit declarations in the source code. Commonly, only the type is explicitly specified and the storage class specifier assumes 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

An *lvalue* is an object locator: an expression that designates an object. An example of an 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, the *l* stood for “*left*”, meaning that an 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 an 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 identifier is the 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 depend on how and where identifiers are declared.

Block Scope

The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the enclosing block). Parameter declarations with a function definition also have block scope, limited to the scope of the function body.

File Scope

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 Scope

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:` followed by a statement. Label names must be unique within a function.

Function Prototype Scope

Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.

Visibility

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

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

Technically, visibility cannot exceed scope, but scope *can* exceed visibility. Take a look at 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 the scope within which an identifier must be unique. C uses four distinct categories of identifiers:

Goto label names

These must be unique within the function in which they are declared.

Structure, union, and enumeration tags

These must be unique within the block in which they are defined. Tags declared outside of any function must be unique.

Structure and union member names

These 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.

Variables, typedefs, functions, and enumeration members

These must be unique within 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 scope rules.

For example:

```
int blue = 73;

{ // open a block
  enum colors { black, red, green, blue, violet, white } c;
  /* enumerator blue 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 storage class, defines the 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 model 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 mikroC, static duration objects are not initialized to zero (or null) in the absence of any explicit initializer.

An object can have static duration *and* local scope – see the example on the following page.

Local Duration

Local duration objects are also known as automatic objects. They are created on the stack (or in a register) when the enclosing block or 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 is usually redundant, because `auto` is the 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. The converse is not true: a local scope object *can* have static duration.

Here is an example of two objects with local scope, but with different duration:

```
void f() {
    /* local duration var; init a upon every call to f */
    int a = 1;

    /* static duration var; init b only upon 1st 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

C is strictly typed language, which means that every object, function, and expression need to have a strictly defined type, known in the time of compilation. Note that C 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 accesses,
- in many type-checking situations, to ensure that illegal assignments are trapped.

mikroC supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers in 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 on those values. The compile-time operator, `sizeof`, lets you determine the size in bytes of any standard or user-defined type.

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

Type Categories

The *fundamental* types represent types that cannot be separated 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 these.

The *derived* types are also known as structured types. The derived types include pointers to other types, arrays of other types, function types, structures, and unions.

FUNDAMENTAL TYPES

Arithmetic Types

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

Integral Types

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

The table below is the 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 `unsigned` prefix, `signed` is automatically assumed for integral types. The only exception is the `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 be applied only to the `int`. The keywords `short` and `long` used on their own mean `short int` and `long int`, respectively.

Floating-point Types

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

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

Below is the overview of arithmetic types:

Type	Size	Range
(unsigned) char	8-bit	0 .. 255
signed char	8-bit	- 128 .. 127
(signed) short (int)	8-bit	- 128 .. 127
unsigned short (int)	8-bit	0 .. 255
(signed) int	16-bit	-32768 .. 32767
unsigned (int)	16-bit	0 .. 65535
(signed) long (int)	32-bit	-2147483648 .. 2147483647
unsigned long (int)	32-bit	0 .. 4294967295
float	32-bit	$\pm 1.17549435082E-38$.. $\pm 6.80564774407E38$
double	32-bit	$\pm 1.17549435082E-38$.. $\pm 6.80564774407E38$
long double	32-bit	$\pm 1.17549435082E-38$.. $\pm 6.80564774407E38$

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 list of discreet values, enumerators. The enumerators listed inside the braces are also known as enumeration constants. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator is set to zero, and each succeeding enumerator is set to one more than its predecessor.

Variables of `enum` type are declared 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, `colors`, a variable `c` of this type, and a set of enumerators with constant integer values (`black = 0, red = 1, ...`). In C, a variable of an enumerated type can be assigned any value of 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 then increase 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.

Anonymous Enum Type

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

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

As 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. For more information, consult Name Spaces.

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. For example:

```
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`, meaning that they can point to any type. These pointers are sometimes called *generic*.

DERIVED TYPES

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

Arrays

Array is the simplest and most commonly used structured type. 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 identifier:

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

This declares an array named as *array_name* composed of elements of *type*. The *type* can be scalar type (except *void*), user-defined type, pointer, enumeration, or another array. Result of the *constant-expression* within the brackets determines the 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 the number of elements in the array.

Each of the elements of an array is numbered from 0 through the number of elements minus one. If the number 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] ;           /* an array of 10 integers */
float vector_two[ MAX] ;       /* an array of 50 floats   */
float vector_three[ MAX - 20] ; /* an array of 30 floats   */
```


Array Initialization

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:

```
/* 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. Vice versa is possible, when the trailing “excess” elements will be assigned 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 in Expressions

When name of the 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 2-dimensional array:

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

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]`. First element of the 5th row would be `m[0][5]`.

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

```
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 C, address of an object in memory can be obtained by means of unary operator `&`. To reach the pointed object, we use 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 at 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.

Note: Currently, mikroC does not support pointers to functions, but this feature will be implemented in future versions.

Pointer Declarations

Pointers are declared same as any other variable, but with `*` ahead of identifier. 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 mikroC.

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 the scoping, duration, and visibility rules apply 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 objects.

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. mikroC 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 a `void` pointer to a non-void pointer – refer to Void Type 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. Instead of zero, the mnemonic `NULL` (defined in the standard library header files, such as `stdio.h`) can be used for legibility. All pointers can be successfully tested for equality or inequality to `NULL`.

For example:

```
int *pn = 0; /* Here's one null pointer */
int *pn = NULL; /* This is an equivalent declaration */

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

/* .. or like this: */
if ( pn == NULL ) { ... }
```

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. Check *Indirect Function Calls*.

Assign an address to a Function Pointer

It's quite easy to assign the address of a function to a function pointer. You simply take the name of a suitable and known function or member function. It's optional to use the address operator & in front of the function's name.

```
//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 C is limited to:

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

The internal arithmetic performed on pointers depends on the memory model 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 C. When name of the 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 is true:

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

According to these guidelines, we can write:

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

Also, you need to be careful with operator precedence:

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

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

You can use a simple assignment operator (=) 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 `void` type).

Assigning the integer constant 0 to a pointer assigns a null pointer value to it. The mnemonic `NULL` (defined in the standard library header files, such as `stdio.h`) can be used for legibility.

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

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

if (pa > pb) { ...
    // this will be executed as 4 is greater than 2
}
```


You can also compare pointers to zero value – this tests if pointer actually points to anything. All pointers can be successfully tested for equality or inequality to NULL:

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

Note: Comparing pointers pointing to different objects/arrays can be performed at programmer’s responsibility — 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 pointer points to an element of an array and if the result is a pointer pointing into the same array (or one element beyond it).

If a pointer is declared to point to *type*, adding an integral value to the pointer advances the pointer by that number of objects of type. Informally, you can think of `P+n` as advancing the pointer `P` by $(n * \text{sizeof}(type))$ bytes, as long as the pointer remains within the legal range (first element to one beyond the last element). If *type* has size of 10 bytes, then adding 5 to a pointer to *type* advances the pointer 50 bytes in memory. In case of `void` type, size of the step is one byte.

For example:

```
int a[ 10 ];           // array a containing 10 elements of 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[1]
```

There is no such element as “one past the last element”, of course, but a pointer is allowed to assume such a 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 a pointer pointing to one element past an array, which is legal. However, applying the 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 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. Difference will equal the distance between the two pointed addresses, in bytes.

For example:

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

Structures

A structure is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere.

Unlike arrays, structures are considered single objects. The mikroC structure type lets you handle complex data structures almost as easily as single variables.

Note: mikroC 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 the structure; *member-declarator-list* is a list of structure members, actually a list of variable declarations. Variables of structured type are declared 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 a declared structure. Here is an example:

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

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

Note that you can omit structure tag, but then you cannot declare additional objects of this type 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. Referring to declarations from the previous example:

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

/* Initialize already declared circles o1 and o2: */
o1 = {1, {0, 0}};           // r is 1, center is at (0, 0)
o2 = {4, { 1.2, -3 }};     // r is 4, center is at (1.2, -3)
```

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 you omit the structure tag, you get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited `struct-id-list` to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere.

It is possible to create a typedef while declaring a structure, with or without a tag:

```
typedef struct { ... } Mystruct;
Mystruct s, *ps, arrs[10];
```

Structure Assignment

Variables of 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 same structured type only if they were both defined by the same instruction or were defined 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

You can get size of the structure in memory by means of operator `sizeof`. Size of the structure does not necessarily need to be 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(); // func1() returns a structure
mystruct *func2(); // 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 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 struct type *S*. Then if *m* is a member identifier of type *M* declared in *s*, the expression

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

is of type *M*, and represents the member object *m* in *s*.

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

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

is of 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[10]; 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 an lvalue, provided that *s* is an 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 structure B contains a field whose type is 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 a; double d;
} s, *sptr;

//...

s.i = 3;           // assign 3 to the i member of B
s.a.j = 2;        // assign 2 to the j member of A
sptr->d = 1.23;    // assign 1.23 to the d member of B
sptr->a.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 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 the syntactic and functional features of structure types. The key difference is that a union allows only one of its members to be “active” at any given time, the most recently changed member.

Note: mikroC does not support anonymous unions (ANSI divergence).

Union Declaration

Unions are declared same 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. Let’s have a simple example:

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

The identifier `mu`, of type `union myunion`, can be used to hold a 2-byte `int`, a 4-byte `double`, or a single-byte `char`, but only one of these at any given time.

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 an `int` object, and 3 bytes are unused when `mu` holds a `char`.

Union Member Access

Union members can be accessed with the structure member selectors (`.` and `->`), but care is needed. Check the example on the following page.

Referring to declarations from the previous example:

```
mu.d = 4.016;
Lcd_Out_Cp(FloatToStr(mu.d)); // OK: displays mu.d = 4.016
Lcd_Out_Cp(IntToStr(mu.i)); // peculiar result

pm->i = 3;
Lcd_Out_Cp(IntToStr(mu.i)); // OK: displays mu.i = 3
```

The second `Lcd_Out_Cp` 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 does not provide a 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. Bit fields 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 mikroC'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. 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 identifier 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 artificial “padding”, thus skipping irrelevant bits.

For example, if we need to manipulate only bits 2–4 of a register as one block, we could create a structure:

```
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 {
    prescaler : 2; timeronoff : 1; postscaler : 4;} mybitfield;
```

which declares structured type `mybitfield` containing three components: `prescaler` (bits 0 and 1), `timeronoff` (bit 2), and `postscaler` (bits 3, 4, 5, and 6).

Bit Fields Access

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

```
// Declare a bit field TimerControl:
mybitfield TimerControl;

void main() {
    TimerControl.prescaler = 0;
    TimerControl.timeronoff = 1;
    TimerControl.postscaler = 3;
    T2CON = TimerControl;
}
```

TYPES CONVERSIONS

C is 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 is changing it to the same object of another type (i.e. applying another type to a given object). C defines a set of standard conversions for built-in types, provided by compiler when necessary.

Conversion is required in following situations:

- if statement requires an expression of particular type (according to language definition), and we use an expression of different type,
- if 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 initializing an object (in declaration) with an object of different type.

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

Standard Conversions

Standard conversions are built in C. These conversions are performed automatically, whenever required in the program. They can be also explicitly required by means of typecast operator (refer to Explicit Typecasting).

The basic rule of automatic (implicit) conversion is that the operand of simpler type is converted (promoted) to the type of more complex operand. Then, type of the result is that of more complex operand.

Arithmetic Conversions

When you use an arithmetic expression, such as `a+b`, where `a` and `b` are of different arithmetic types, mikroC 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 set the high byte to zero when converted to `int`.

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

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

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 `unsigned 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 they are `float` (equivalent with `double` and `long double` in mikroC).

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`, the other operand is converted to `long`
4. Otherwise, if either operand is `unsigned`, 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. = 3.0
3.*5/4        // = (3.*5)/4 = (3.*5.)/4 = 15./4 = 15./4. = 3.75
```

Pointer Conversions

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 Types Conversions (Typecasting)

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

```
(type) object
```

For example:

```
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

Introduction to Declarations

Declaration introduces one or several names to a program – it informs the compiler what the name represents, what is its type, what are allowed operations 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 the 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 an extern specifier, and has no initializer or body (in case of func.),
- it is a 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.

Let's have an 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;
}

int i; /* Definition of variable i */
int i; /* Error: i is already defined! */
```

Declarations and Declarators

A declaration is a list of names. The names are sometimes referred to as declarators or identifiers. 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*. Specifier *storage-class* can take 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.

Here is an example of variable declaration:

```
/* 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
```

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. The term translation unit refers to a source code file together with any included files, but less any 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.

Linkage is the 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 the 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; 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 Rule:

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. For each identifier in a translation unit declared with internal linkage, no more than one external definition can be given. An external definition is an external declaration that also defines an object or function; that is, it also allocates 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.

mikroC allows later declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. Here's an example:

```
int a[];           // No size
struct mystruct;  // Tag only, no member declarators
.
.
.
int a[3] = { 1, 2, 3}; // Supply size and initialize
struct mystruct {
    int i, j;
};                // Add member declarators
```

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 compiler deduces these attributes from implicit or explicit declarations in the source code.

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

```
storage-class type identifier
```

The storage class specifiers in mikroC are:

```
auto  
register  
static  
extern
```

Auto

Use the `auto` modifier to define a local variable as having a local duration. This is the default for local variables and is rarely used. You cannot use `auto` with globals. See also Functions.

Register

By default, mikroC stores variables within internal microcontroller memory. Thus, modifier `register` technically has no special meaning. mikroC compiler simply ignores requests for register allocation.

Static

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

Local name declared with `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

Name declared with `extern` specifier has external linkage, unless it has been previously declared as having internal linkage. Declaration is not a definition if it has `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 a variable, or body of a function, is defined in a separate source code module. Functions declared with `extern` are visible throughout all source files in a program, unless you redefine the function as `static`.

See Linkage for more information.

Type Qualifiers

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

Qualifier `const`

Qualifier `const` implies that the declared object will not change its value during runtime. In declarations with `const` qualifier, you need to initialize all the objects in the declaration.

Effectively, mikroC treats objects declared with `const` qualifier same as literals or preprocessor constants. Compiler will report an error if trying to change an object declared with `const` qualifier.

For example:

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

Qualifier `volatile`

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

Typedef Specifier

Specifier `typedef` introduces a synonym for a specified type. You can use `typedef` declarations to construct shorter or more meaningful names for types already defined by the language or for types that you have declared. You cannot use the `typedef` specifier inside a function definition.

The specifier `typedef` stands first in the declaration:

```
typedef <type-definition> synonym;
```

The `typedef` keyword assigns the *synonym* to the *<type-definition>*. The *synonym* needs to be a valid identifier.

Declaration starting with the `typedef` specifier does not introduce an object or function of a given type, but rather a new name for a given type. That is, the `typedef` declaration is identical to “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 C.

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 `typedef` declaration, as in any declaration, you can declare several types at once. For example:

```
typedef int *Pti, Array[10];
```

Here, `Pti` is synonym for type “pointer to `int`”, and `Array` is synonym for type “array of 10 `int` elements”.

asm Declaration

C allows embedding assembly in the source code by means of asm declaration. Declarations `_asm` and `__asm` are also allowed in mikroC, and have the same meaning. Note that you cannot use numerals as absolute addresses for SFR or GPR variables in assembly instructions. You may use symbolic names instead (listing will display these names as well as addresses).

You can group assembly instructions by the `asm` keyword (or `_asm`, or `__asm`):

```
asm {  
    block of assembly instructions  
}
```

C comments (both single-line and multi-line) are allowed in embedded assembly code. Assembly-style comments starting with semicolon are not allowed.

If you plan to use a certain C variable in embedded assembly only, be sure to at least initialize it in C code; otherwise, linker will issue an error. This does not apply to predefined globals such as `PORTB`.

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

```
unsigned myvar;  
void main() {  
    asm {  
        MOVLW 10 // just a test  
        MOVLW test_main_global_myvar_1  
    }  
}
```

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

```
myvar := 0;
```

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

Migration from older versions of mikroC

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

Variable mangling is altered, and is now more in C-manner. For example, for variable named :

- `_myVar`, if it is global.
- `FARG_+XX`, if it is local (this is myVar's actual position in the local function frame).
- `_myVar_L0(+XX)`, if it is a local static variable (+XX to access further individual bytes).

The only types whose name remains the same in asm as it is in C are constants, e.g. `INTCON`, `PORTB`, `WREG`, `GIE`, etc.

Accessing individual bytes is different as well. For example, if you have a global variable "g_var", that is of type long (i.e. 4 bytes), you are to access it like this:

```

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

```

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

```

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

```

For objects located in RAM:

```

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

```

Initialization

At the time of declaration, you can set the initial value of a declared object, i.e. initialize it. Part of the declaration which specifies the initialization is called the 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 the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

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 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

mikroC 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 a function can be used in expressions – technically, function call is considered an operator like any other.

C allows a function to create results other than its return value, referred to as *side effects*. Often, 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 info on functions' limitations on PIC micros.

Function Declaration

Functions are declared in your source files or made available by linking precompiled libraries. Declaration syntax of a 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. The *type* represents the type of function result, and can be any standard or user-defined type. For functions that do not return value, you should use `void` type. The *type* can be omitted in global function declarations, and function will assume `int` type by default.

Function *type* can also be a pointer. For example, `float*` means that the function result is a pointer to `float`. Generic pointer, `void*` is also allowed. Function *cannot* return 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 function calls for validity. If the list is empty, function does not take any arguments. Also, if the list is `void`, 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 with variable declaration, each argument in the list needs its own type specifier and a possible qualifier `const` or `volatile`.

Function Prototypes

A given function can be defined only once in a program, but can be declared several times, provided the declarations are compatible. If you write a nondefining declaration of a function, i.e. without the function body, you do not have to specify the formal arguments. This kind of declaration, commonly known as the *function prototype*, allows better control over argument number and type checking, and type conversions.

Name of the parameter in function prototype has its scope limited to the prototype. This allows different parameter names 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 greatly aid 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, you can that file to get the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

Function Definition

Function definition consists of its declaration and a 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. This means that function declarations cannot be nested.

To return the function result, use the `return` statement. Statement `return` in functions of `void` type cannot have a parameter – in fact, you can omit the `return` statement 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 coordinates (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

Limited reentrancy for functions is allowed. The functions that don't have their own function frame (no arguments and local variables) can be called both from the interrupt and the "main" thread. Functions that have input arguments and/or local variables can be called only from one of the before mentioned program threads. Check *Indirect Function Calls*.

Function Calls

A function is called with actual arguments placed in the same sequence as their matching formal parameters. Use a 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 disagree, implicit type conversions rules apply. Actual arguments can be of any complexity, but you should not depend on their order of evaluation, because it is not specified.

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

If the function is without result (type `void`) or you don't need the result, you can write the function call as a self-contained expression.

In C, scalar parameters are always passed to function by value. A function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine. You can pass scalar object by the address by declaring a formal parameter to be a pointer. Then, use the indirection operator `*` to access the pointed object.

Argument Conversions

When a function prototype has not been previously declared, mikroC converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard Conversions. When a function prototype is in scope, mikroC converts the given argument to the type of the declared parameter as if by assignment.

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. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Note: If your function prototype does not match the actual function definition, mikroC 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 a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught.

The compiler is also able to force arguments to the proper type. Suppose you have the following code:

```
int limit = 32;
char ch = 'A';
long res;

extern long func(long par1, long par2);           // prototype

main() {
    //...
    res = func(limit, ch);                       // function call
}
```

Since it has the function prototype for `func`, this program 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 have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to `func` would not match in size or content what `func` was expecting, leading to problems.

Ellipsis ('...') Operator

An ellipsis ('...') consists of three successive periods with no whitespace intervening. You can use an ellipsis 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, an `int` and a `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.

mikroC recognizes following operators:

- Arithmetic Operators
- Assignment Operators
- Bitwise Operators
- Logical Operators
- Reference/Indirect Operators (see Pointer Arithmetic)
- Relational Operators
- Structure Member Selectors (see Structure Member Access)

- Comma Operator , (see Comma Expressions)
- Conditional Operator ? :

- Array subscript operator [] (see Arrays)
- Function call operator () (see Function Calls)

- sizeof Operator

- Preprocessor Operators # and ## (see Preprocessor Operators)

Operators Precedence and Associativity

There are 15 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Table on the following page sums all mikroC operators.

Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left-to-right or right-to-left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

Precedence	Operands	Operators	Associativity
15	2	() [] . ->	left-to-right
14	1	! ~ ++ -- + - * & (type) sizeof	right-to-left
13	2	* / %	left-to-right
12	2	+ -	left-to-right
11	2	<< >>	left-to-right
10	2	< <= > >=	left-to-right
9	2	== !=	left-to-right
8	2	&	left-to-right
7	2	^	left-to-right
6	2		left-to-right
5	2	&&	left-to-right
4	2		left-to-right
3	3	? :	left-to-right
2	2	= *= /= %= += -= &= ^= = <<= >>=	right-to-left
1	2	,	left-to-right

Arithmetic Operators

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results. Type char technically represents small integers, so char variables can be used as operands in arithmetic operations.

All of arithmetic operators associate from left to right.

Operator	Operation	Precedence
+	addition	12
-	subtraction	12
*	multiplication	13
/	division	13
%	returns the remainder of integer division (cannot be used with floating points)	13
+ (unary)	unary plus does not affect the operand	14
- (unary)	unary minus changes the sign of operand	14
++	increment adds one to the value of the operand	14
--	decrement subtracts one from the value of the operand	14

Note: Operator * is context sensitive and can also represent the pointer reference operator. See Pointers for more information.

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 as we are working with floats
```

Remainder operand % works only with integers; sign of result is equal to the sign of first operand:

```
/* for example: */
9 % 3;         // equals 0
7 % 3;         // equals 1
-7 % 3;        // equals -1
```

We can use arithmetic operators 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 operand *before* the evaluation. When used as suffix, operators ++ and -- add or subtract one from the value of operand *after* the evaluation.

For example:

```
int j = 5; j = ++k;
/* k = k + 1, j = k, which gives us j = 6, k = 6 */

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 the expression evaluates to 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 was designated in such a way to allow complex expressions without parentheses to have expected meaning:

`a + 5 >= c - 1.0 / e` // i.e. $(a + 5) \geq (c - (1.0 / e))$

Always bear in mind that relational operators return either 0 or 1. Consider the following examples:

```
8 == 13 > 5           // returns 0: 8==(13>5), 8==1, 0
14 > 5 < 3           // returns 1: (14>5)<3, 1<3, 1
a < b < 5            // returns 1: (a<b)<5, (0 or 1)<5, 1
```

Bitwise Operators

Use the bitwise operators to modify the 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
<code>&</code>	bitwise AND; returns 1 if both bits are 1, otherwise returns 0	9
<code> </code>	bitwise (inclusive) OR; returns 1 if either or both bits are 1, otherwise returns 0	9
<code>^</code>	bitwise exclusive OR (XOR); returns 1 if the bits are complementary, otherwise 0	10
<code>~</code>	bitwise complement (unary); inverts each bit	10
<code>>></code>	bitwise shift left; moves the bits to the left, see below	10
<code><<</code>	bitwise shift right; moves the bits to the right, see below	10

Note: Operator `&` can also be the pointer reference operator. Refer to Pointers for more information.

Bitwise operators `&`, `|`, and `^` perform logical operations on appropriate pairs of bits of their operands. 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 */
```

Bitwise Shift Operators

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

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

```
000001 << 5;      /* equals 000040 */  
0x3801 << 4;      /* equals 0x8010, overflow! */
```

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

```
0xFF56 >> 4;      /* equals 0xFFF5 */  
0xFF56u >> 4;     /* equals 0x0FF5 */
```

Bitwise vs. Logical

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 */
```

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
<code>&&</code>	logical AND	5
<code> </code>	logical OR	4
<code>!</code>	logical negation	14

Precedence of logical, relational, and arithmetic operators was chosen in such a way to allow complex expressions without parentheses to have 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 is not 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 the expressions evaluate 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 with complex logical expressions is that the evaluation of consecutive logical operands stops 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 vs. 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
```

Expression1 evaluates 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 the value of either *expression2* or *expression3* depending upon which evaluates. The fact that only one of these two expressions evaluates is very important if you expect them 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 of arithmetic type; *expression2* and *expression3* are subject to the usual arithmetic conversions, which determines the resulting type.
2. Both of compatible struct or union types. The resulting type is the structure or union type of *expression2* and *expression3*.
3. Both of void type. The resulting type is void.

4. Both of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
5. One operand is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
6. One operand 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` operand.

Assignment Operators

Unlike many other programming languages, C treats value assignment as an operation (represented by an operator) rather than instruction.

Simple Assignment Operator

For a common value assignment, we use a simple assignment operator (=) :

```
expression1 = expression2
```

Expression1 is an object (memory location) to which we assign value of *expression2*. Operand *expression1* has to be a lvalue, and *expression2* can be any expression. The assignment expression itself is not an lvalue.

If *expression1* and *expression2* are of different types, result of the *expression2* will be converted to the type of *expression1*, if necessary. Refer to Type Conversions for more information.

Compound Assignment Operators

C allows more complex assignments by means of compound assignment operators. 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 these associate from right to left. Spaces separating compound operators (e.g. + =) will generate 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 a qualified or unqualified arithmetic type and *expression2* is an arithmetic type.
2. *expression1* has a qualified or unqualified version of a 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 the left has all the qualifiers of the type pointed to by the 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 the left has all the qualifiers of the type pointed to by the right.
5. *expression1* is a pointer and *expression2* is a null pointer constant.

Sizeof Operator

Prefix unary operator `sizeof` returns an integer constant that gives the size in bytes of how much memory space is used by its operand (determined by its type, with some exceptions).

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 an lvalue that designates a bit field object.

Sizeof Applied to Expression

If applied to expression, size of the operand is determined without evaluating the expression (and therefore without side effects). 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. Unit for type size is the `sizeof(char)` which is equivalent to one byte. Operation `sizeof(char)` gives the result 1, whether the char is signed or unsigned.

```
sizeof(char)           /* returns 1 */
sizeof(int)           /* returns 2 */
sizeof(unsigned long) /* 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 */
```

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. Operator `sizeof` cannot be applied to a function.

EXPRESSIONS

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

Expressions can produce an lvalue, an 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 you to use comma as a sequence operator to form the 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 same as `(a, b), c`. This allows us to write sequences with any number of expressions:

```
expression_1, expression_2, ... expression_n;
```

this 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  
   initializes 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 the comma punctuator which separates elements in a function argument list and initializer lists. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. For example,

```
func(i, (j = 1, j + 4), k);
```

calls function `func` with three arguments (`i, 5, k`), not four.

STATEMENTS

Statements specify the flow of control as a 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

Every statement in program can be labeled. 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 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;
```

mikroC executes an expression statement by evaluating the *expression*. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls.

The *null statement* is a special case, consisting of a single semicolon (;). The null statement does nothing, and is therefore useful in situations where the mikroC syntax expects a statement but your program does not need one. For example, 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 from alternative courses of action by testing certain values. There are two types of selection statements in C: `if` and `switch`.

If Statement

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

```
if (expression) statement1 [else statement2]
```

When *expression* evaluates to true, *statement1* executes. If *expression* 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 the `if` and the `else`.

Nested if statements

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

```
if (expression1) statement1
else if (expression2)
    if (expression3) statement2
    else statement3          /* this belongs to: if (expression3) */
else statement4            /* this belongs to: if (expression2) */
```

Note: The `#if` and `#else` preprocessor statements (directives) look similar to the `if` and `else` statements, but have very different effects. They control which source file lines are compiled and which are ignored. See Preprocessor for more information.

Switch Statement

Use the `switch` statement to pass control to a specific program branch, based on a certain condition. Syntax of `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 the available *constant-expressions* following the keyword `case`. If the match is found, `switch` passes control to that matching case, at which point the *statement* following the match evaluates. Note that *constant-expressions* must evaluate to integer. There cannot be two same *constant-expressions* evaluating to same value.

Parantheses around *expression* are mandatory.

Upon finding a match, program flow continues normally: 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 variable `i` has value between 1 and 3, 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 the `switch`, terminate each case with `break`.

Conditional `switch` statements can be nested – labels `case` and `default` are then assigned to the innermost enclosing `switch` statement.

Here is a simple example with `switch`. Let's assume we have a variable 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 appropriate routine:

```
switch (state) {  
    case 0: Lo(); break;  
    case 1: Mid(); break;  
    case 2: Hi(); break;  
    default: Message("Invalid state!");  
}
```

Iteration Statements

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

While Statement

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

```
while (expression) statement
```

The *statement* executes repeatedly until the value of *expression* is false. The test takes place before *statement* executes. Thus, if *expression* evaluates to false on the first pass, the loop does not execute.

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 a loop can be a null statement. For example:

```
while (*q++ = *p++);
```

Do Statement

The `do` statement executes until the condition becomes false. Syntax of `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 a 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. Syntax of `for` statement is:

```
for ([ init-exp ] ; [ condition-exp ] ; [ increment-exp ] ) statement
```

Before the first iteration of the loop, expression *init-exp* sets the starting variables for the loop. You cannot pass declarations in *init-exp*.

Expression *condition-exp* is checked before the first entry into the block; *statement* is executed repeatedly until the value of *condition-exp* is false. After each iteration of the loop, *increment-exp* increments a loop counter. Consequently, `i++` is functionally the same as `++i`.

All the expressions are optional. If *condition-exp* 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 ( ; ; ) { ... }
```

The only way to break out of this loop is by means of `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 ] ;
```

You can also do it like this:

```
/* valid, but ugly */  
for (s = 0, i = 0; i < n; s += a[ i ] * b[ i ], i++);
```

but this 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 we used a null statement (`;`) for a loop body.

Jump Statements

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

Break Statement

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

`Break` is commonly used in `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

You can use the `continue` statement within loops (`while`, `do`, `for`) 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 loop continuation condition is true.

Goto Statement

Use the `goto` statement to unconditionally jump to a local label — for more information on labels, refer to Labeled Statements. Syntax of `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.

You can use `goto` to break out from any level of nested control structures. But, `goto` cannot be used to jump into block while skipping that block's initializations – for example, jumping into loop's body, etc.

Use of `goto` statement is generally discouraged as practically every algorithm can be realized without it, resulting in legible structured programs. One possible application of `goto` statement is breaking out from deeply nested control structures:

```
for (...) {
    for (...) {
        ...
        if (disaster) goto Error;
        ...
    }
}
.
.
.
Error: /* error handling code */
```

Return Statement

Use the `return` statement to exit from the current function back to the calling routine, optionally returning a value. Syntax is:

```
return [ expression ] ;
```

This will evaluate the *expression* and return the result. Returned value will be automatically converted to the expected function type, if needed. The *expression* is optional; if omitted, function will return a random value from memory.

Note: Statement `return` in functions of void type cannot have an *expression* – in fact, you can omit the `return` statement altogether if it is the last statement in the function body.

Compound Statements (Blocks)

A compound statement, or block, is a list (possibly empty) of statements enclosed in matching braces `{ }`. Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a 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, `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 specified file to a certain point in code,
- replacing specific lexical symbols with other symbols,
- conditional compiling which conditionally includes or omits parts of code.

Note that preprocessor analyzes text at token level, not at individual character level. Preprocessor is controlled by means of preprocessor directives and preprocessor operators.

Preprocessor Directives

Any line in source code with a leading # is taken as a *preprocessing directive* (or *control line*), unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

The 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 preprocessor detects preprocessor directives and parses the tokens embedded in them. Directive is in effect from its declaration to the end of the program file.

mikroC supports standard preprocessor directives:

# (null directive)	#if
#define	#ifndef
#elif	#ifndef
#else	#include
#endif	#line
#error	#undef

Note: #pragma directive is under construction.

Line Continuation with Backslash

If you need to break directive into multiple lines, you can do it by ending the line with a backslash (\):

```
#define MACRO This directive continues to \  
              the following line.
```

Macros

Macros provide a mechanism for 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 noted later). Such replacements are known as *macro expansions*. The *token_sequence* is sometimes called body of the 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 the macro expansion. The *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 for the possibility of nested macros: The expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, such a 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 are not expanded

A macro won't be expanded during its own expansion (so `#define MACRO MACRO` won't expand indefinitely).

Let's have 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 the new definition is exactly the same token-by-token definition as the existing one. The preferred strategy where 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 there can be no whitespace between the *macro_identifier* and the “(”. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays 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 the two lists is different.

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 the *actual_arg_list*. As 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 of the arguments in macro body – this will avoid possible problems with operator precedence.

Undefining Macros

You can undefine a macro using the `#undef` directive.

```
#undef macro_identifier
```

Directive `#undef` detaches any previous token sequence from the *macro_identifier*; the macro definition has been forgotten, and the *macro_identifier* is undefined. No macro expansion occurs within `#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 a 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 `.c`) — see Projects for more information.

The syntax of `#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 the header file at that point in the source code. The placement of the `#include` can therefore influence the scope and duration of any identifiers in the included file.

The difference between the two formats lies in the searching algorithm employed in trying to locate the include file.

If `#include` directive was used with the `<header_name>` version, the search is made successively in each of the following locations, in this particular order:

1. mikroC installation folder > “include” folder,
2. your custom search paths.

The `"header_name"` version specifies a user-supplied include file; mikroC will look for the header file in following locations, in this particular order:

1. the project folder (folder which contains the project file `.ppc`),
2. mikroC installation folder > “include” folder,
3. your custom search paths.

Explicit Path

If you place an explicit path in the `header_name`, only that directory will be searched. For example:

```
#include "C:\my_files\test.h"
```

Note: There is also a third version of `#include` directive, rarely used, which assumes that neither `<` nor `"` appears as the first non-whitespace character following `#include`:

```
#include macro_identifier
```

It assumes a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the `<header_name>` or `"header_name"` formats.

Preprocessor Operators

The `#` (pound sign) is a preprocessor directive when it occurs as the first non-whitespace character on a line. Also, `#` and `##` perform operator replacement and merging during the preprocessor scanning phase.

Operator `#`

In C preprocessor, 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 result of preprocessing, you can 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_Out_Cp(#val " : "); \  
                        Lcd_Out_Cp(IntToStr(val));
```

(note the backslash as a line-continuation symbol)

Now, the following code,

```
LCD_PRINT(temp)
```

will be preprocessed to this:

```
Lcd_Out_Cp("temp" " : "); Lcd_Out_Cp(IntToStr(temp));
```

Operator

Operator ## is used for token pasting: you can paste (or merge) two tokens together by placing ## in between them (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. This is commonly used for constructing identifiers.

For example, we could define macro `SPLICE` for pasting two tokens into one identifier:

```
#define SPLICE(x,y) x ## _ ## y
```

Now, the call `SPLICE(cnt, 2)` expands to identifier `cnt_2`.

Note: mikroC does not support the older nonportable method of token pasting using `(1/**/r)`.

Conditional Compilation

Conditional compilation directives are typically used to make source programs easy to change and easy to compile in different execution environments. mikroC 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 are 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 the #if has a nonzero value, the line group immediately following the #if directive is retained in the translation unit.

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 the #if and #endif directives, but at most one #else directive is allowed. The #else directive, if present, must be the last directive before #endif.

The sections can be any program text that has meaning to the compiler or the preprocessor. The preprocessor selects a single section by evaluating the *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 `#if` directive.

The net result of the preceding scenario is that only one code *section* (possibly empty) will be compiled.

Directives `#ifdef` and `#ifndef`

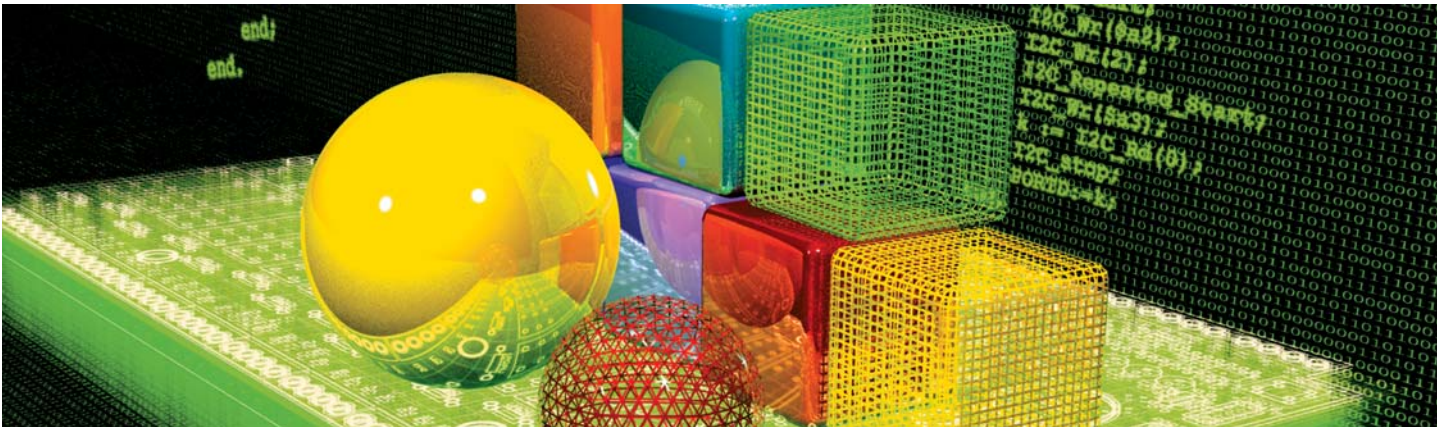
You can use the `#ifdef` and `#ifndef` directives anywhere `#if` can be used. The `#ifdef` and `#ifndef` conditional directives let you test whether an identifier is currently defined or not. The line

```
#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 the `#if`, `#elif`, `#else`, and `#endif`.

An identifier defined as NULL is considered to be defined.



mikroC Libraries

mikroC provides a number of built-in and library routines which help you develop your application faster and easier. Libraries for ADC, CAN, USART, SPI, I2C, 1-Wire, LCD, PWM, RS485, Serial Ethernet, Toshiba GLCD, Port Expander, Serial GLCD, numeric formatting, bit manipulation, and many other are included along with practical, ready-to-use code examples.

BUILT-IN ROUTINES

mikroC 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` and `Delay_Cyc`, 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`

Lo

Prototype	<code>unsigned short Lo(long number);</code>
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 scalar type (i.e. Arithmetic Types and Pointers).
Example	<pre>d = 0x1AC30F4; tmp = Lo(d); // Equals 0xF4</pre>

Hi

Prototype	<code>unsigned short Hi(long number);</code>
Returns	Returns next to the lowest byte of number, bits 8..15.
Description	Function returns next to the lowest byte of number. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Requires	Arguments must be scalar type (i.e. Arithmetic Types and Pointers).
Example	<pre>d = 0x1AC30F4; tmp = Hi(d); // Equals 0x30</pre>

Higher

Prototype	<code>unsigned short Higher(long number);</code>
Returns	Returns next to the highest byte of number, bits 16..23.
Description	Function returns next to the highest byte of number. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Requires	Arguments must be scalar type (i.e. Arithmetic Types and Pointers).
Example	<pre>d = 0x1AC30F4; tmp = Higher(d); // Equals 0xAC</pre>

Highest

Prototype	<code>unsigned short Highest(long number);</code>
Returns	Returns the highest byte of number, bits 24..31.
Description	Function returns the highest byte of number. Function does not interpret bit patterns of number – it merely returns 8 bits as found in register. This is an “inline” routine; code is generated in the place of the call, so the call doesn’t count against the nested call limit.
Requires	Arguments must be scalar type (i.e. Arithmetic Types and Pointers).
Example	<pre>d = 0x1AC30F4; tmp = Highest(d); // Equals 0x01</pre>

Delay_us

Prototype	<code>void Delay_us(const time_in_us);</code>
Description	Creates a software delay in duration of <code>time_in_us</code> microseconds (a constant). Range of applicable constants depends on the oscillator frequency.
Example	<code>Delay_us(10); /* Ten microseconds pause */</code>

Delay_ms

Prototype	<code>void Delay_ms(const time_in_ms);</code>
Description	Creates a software delay in duration of <code>time_in_ms</code> milliseconds (a constant). Range of applicable constants depends on the oscillator frequency.
Example	<code>Delay_ms(1000); /* One second pause */</code>

Vdelay_ms

Prototype	<code>void Vdelay_ms(unsigned time_in_ms);</code>
Description	Creates a software delay in duration of <code>time_in_ms</code> milliseconds (a variable). Generated delay is not as precise as the delay created by <code>Delay_ms</code> .
Example	<code>pause = 1000; // ... Vdelay_ms(pause); // ~ one second pause</code>

Delay_Cyc

Prototype	<code>void Delay_Cyc(char Cycles_div_by_10);</code>
Description	<p>Creates a delay based on MCU clock. Delay lasts for 10 times the input parameter in MCU cycles. Input parameter needs to be in range 3 .. 255.</p> <p>Note that <code>Delay_Cyc</code> is library function rather than a built-in routine; it is presented in this topic for the sake of convenience.</p>
Example	<code>Delay_Cyc(10); /* Hundred MCU cycles pause */</code>

Clock_Khz

Prototype	<code>unsigned Clock_Khz(void);</code>
Returns	Device clock in KHz, rounded to the nearest integer.
Description	Returns device clock in KHz, rounded to the nearest integer.
Example	<code>clk = Clock_Khz();</code>

Clock_Mhz

Prototype	<code>unsigned Clock_Mhz(void);</code>
Returns	Device clock in MHz, rounded to the nearest integer.
Description	Returns device clock in MHz, rounded to the nearest integer.
Example	<code>clk = Clock_Mhz();</code>

LIBRARY ROUTINES

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

Currently available libraries are:

Hardware/PIC-specific Libraries

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

- SPI LCD Library
- SPI LCD8 Library
- SPI T6963C Graphic LCD Library

Standard ANSI C Libraries

- ANSI C Ctype Library
- ANSI C Math Library
- ANSI C Stdlib Library
- ANSI C String Library

Miscellaneous Libraries

- Conversions Library
- Trigonometry Library
- sprint Library
- Setjmp Library
- Time 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	<code>unsigned Adc_Read(char channel);</code>
Returns	10-bit unsigned value read from the specified ADC channel.
Description	<p>Initializes PIC's internal ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12TAD).</p> <p>Parameter <code>channel</code> represents the channel from which the analog value is to be acquired. For channel-to-pin mapping please refer to documentation for the appropriate PIC MCU.</p>
Requires	<p>PIC MCU with built-in ADC module. You should consult the Datasheet documentation for specific device (most devices from PIC16/18 families have it).</p> <p>Before using the function, be sure to configure the appropriate TRISA bits to designate the pins as input. Also, configure the desired pin as analog input, and set Vref (voltage reference value).</p> <p>The function is currently unsupported by the following PICmicros: P18F2331, P18F2431, P18F4331, and P18F4431.</p>
Example	<pre>unsigned tmp; ... tmp = Adc_Read(1); /* read analog value from channel 1 */</pre>

Library Example

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

```

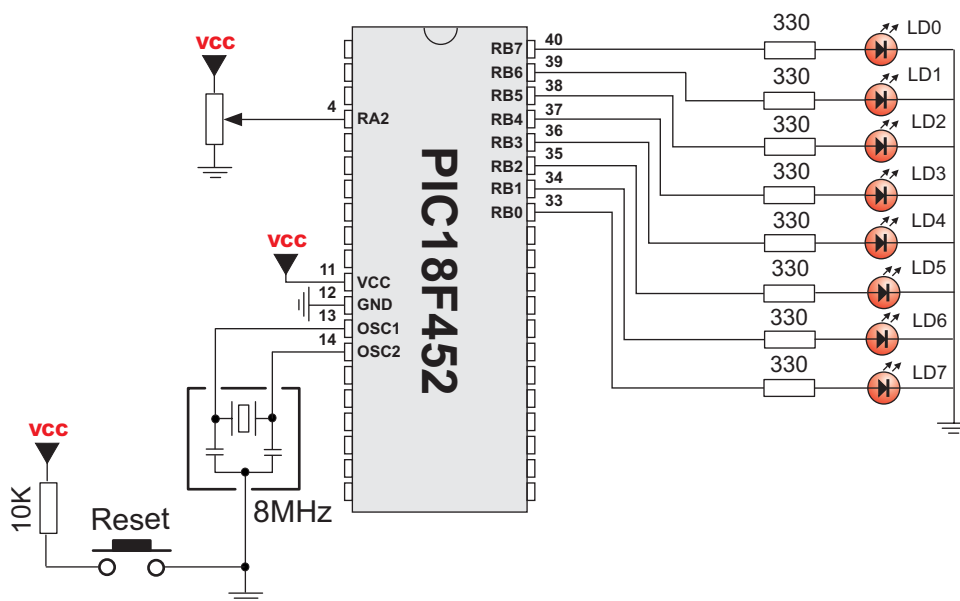
unsigned temp_res;

void main() {
    ADCON1 = 0x80;    // Configure analog inputs and Vref
    TRISA = 0xFF;    // PORTA is input
    TRISB = 0x3F;    // Pins RB7, RB6 are outputs
    TRISD = 0;       // PORTD is output

    do {
        temp_res = Adc_Read(2);    // Get results of AD conversion
        PORTD = temp_res;          // Send lower 8 bits to PORTD
        PORTB = temp_res >> 2;    // Send 2 most significant bits to RB7, RB6
    } while(1);
}

```

Hardware Connection



CAN Library

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

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

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

CAN supports two message formats:

Standard format, with 11 identifier bits, and

Extended format, with 29 identifier bits

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

Note: Be sure to check CAN constants necessary for using some of the functions. See page 145.

Library Routines

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

Following routines are for the internal use by compiler only:

```
RegsToCANID  
CANIDToRegs
```

CANSetOperationMode

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

CANGetOperationMode

Prototype	<code>char CANGetOperationMode(void);</code>
Returns	Current opmode.
Description	Function returns current operational mode of CAN module.
Requires	CAN routines are currently supported only by P18XXX8 PICmicros. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	<code>if (CANGetOperationMode() == CAN_MODE_NORMAL) { ... };</code>

CANInitialize

Prototype	void CANInitialize(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);
Description	<p>Initializes CAN. All pending transmissions are aborted. Sets all mask registers to 0 to allow all messages. The Config mode is internally set by this function. Upon a execution of this function Normal mode is set.</p> <p>Filter registers are set according to flag value:</p> <pre>if (CAN_CONFIG_FLAGS & CAN_CONFIG_VALID_XTD_MSG != 0) // Set all filters to XTD_MSG else if (config & CONFIG_VALID_STD_MSG != 0) // Set all filters to STD_MSG else // Set half the filters to STD, and the rest to XTD_MSG</pre> <p>Parameters:</p> <p>SJW as defined in 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).</p>
Requires	CAN routines are currently supported only by P18XXX8 PICmicros. Microcontroller must be connected to CAN transceiver (MCP2551 or similar) which is connected to CAN bus.
Example	<pre>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</pre>

CANSetBaudRate

Prototype	<code>void CANSetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);</code>
Description	<p>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.</p> <p>Parameters:</p> <p>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)</p>
Requires	CAN must be in Config mode; otherwise the function will be ignored.
Example	<pre>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);</pre>

CANSetMask

Prototype	<code>void CANSetMask(char CAN_MASK, long value, char CAN_CONFIG_FLAGS);</code>
Description	<p>Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers.</p> <p>Parameters: CAN_MASK is one of predefined constant values (see CAN constants); value is the mask register value; CAN_CONFIG_FLAGS selects type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG.</p>
Requires	CAN must be in Config mode; otherwise the function will be ignored.
Example	<pre>// Set all mask bits to 1, i.e. all filtered bits are relevant: 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. */</pre>

CANSetFilter

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

CANRead

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

CANWrite

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

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:

```
#define CAN_MODE_BITS      0xE0    // Use it to access mode bits
#define CAN_MODE_NORMAL    0
#define CAN_MODE_SLEEP     0x20
#define CAN_MODE_LOOP      0x40
#define CAN_MODE_LISTEN    0x60
#define 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:

```
#define CAN_CONFIG_DEFAULT          0xFF    // 11111111

#define CAN_CONFIG_PHSEG2_PRG_BIT   0x01
#define CAN_CONFIG_PHSEG2_PRG_ON   0xFF    // XXXXXXX1
#define CAN_CONFIG_PHSEG2_PRG_OFF  0xFE    // XXXXXXX0

#define CAN_CONFIG_LINE_FILTER_BIT  0x02
#define CAN_CONFIG_LINE_FILTER_ON   0xFF    // XXXXXX1X
#define CAN_CONFIG_LINE_FILTER_OFF  0xFD    // XXXXXX0X

#define CAN_CONFIG_SAMPLE_BIT       0x04
#define CAN_CONFIG_SAMPLE_ONCE     0xFF    // XXXXX1XX
#define CAN_CONFIG_SAMPLE_THRICE    0xFB    // XXXXX0XX

#define CAN_CONFIG_MSG_TYPE_BIT     0x08
#define CAN_CONFIG_STD_MSG          0xFF    // XXXX1XXX
#define CAN_CONFIG_XTD_MSG          0xF7    // XXXX0XXX

// continues..
```

```
// ..continued

#define CAN_CONFIG_DBL_BUFFER_BIT      0x10
#define CAN_CONFIG_DBL_BUFFER_ON      0xFF // XXX1XXXX
#define CAN_CONFIG_DBL_BUFFER_OFF     0xEF // XXX0XXXX

#define CAN_CONFIG_MSG_BITS           0x60
#define CAN_CONFIG_ALL_MSG            0xFF // X11XXXXX
#define CAN_CONFIG_VALID_XTD_MSG      0xDF // X10XXXXX
#define CAN_CONFIG_VALID_STD_MSG      0xBF // X01XXXXX
#define 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:

```
#define CAN_TX_PRIORITY_BITS      0x03
#define CAN_TX_PRIORITY_0        0xFC // XXXXXX00
#define CAN_TX_PRIORITY_1        0xFD // XXXXXX01
#define CAN_TX_PRIORITY_2        0xFE // XXXXXX10
#define CAN_TX_PRIORITY_3        0xFF // XXXXXX11

#define CAN_TX_FRAME_BIT         0x08
#define CAN_TX_STD_FRAME         0xFF // XXXXX1XX
#define CAN_TX_XTD_FRAME         0xF7 // XXXXX0XX

#define CAN_TX_RTR_BIT           0x40
#define CAN_TX_NO_RTR_FRAME      0xFF // X1XXXXXX
#define 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.

```
#define CAN_RX_FILTER_BITS 0x07 // Use it to access filter bits
#define CAN_RX_FILTER_1 0x00
#define CAN_RX_FILTER_2 0x01
#define CAN_RX_FILTER_3 0x02
#define CAN_RX_FILTER_4 0x03
#define CAN_RX_FILTER_5 0x04
#define CAN_RX_FILTER_6 0x05
#define CAN_RX_OVERFLOW 0x08 // Set if Overflowed; else clear
#define CAN_RX_INVALID_MSG 0x10 // Set if invalid; else clear
#define CAN_RX_XTD_FRAME 0x20 // Set if XTD msg; else clear
#define CAN_RX_RTR_FRAME 0x40 // Set if RTR msg; else clear
#define CAN_RX_DBL_BUFFERED 0x80 // Set if msg was
// hardware double-buffered
```

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

```
if (MsgFlag & CAN_RX_OVERFLOW != 0) {
    ... // Receiver overflow has occurred; previous message is lost.
}
```

CAN_MASK

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

```
#define CAN_MASK_B1 0
#define CAN_MASK_B2 1
```

CAN_FILTER

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

```
#define CAN_FILTER_B1_F1 0
#define CAN_FILTER_B1_F2 1
#define CAN_FILTER_B2_F1 2
#define CAN_FILTER_B2_F2 3
#define CAN_FILTER_B2_F3 4
#define CAN_FILTER_B2_F4 5
```

Library Example

```
unsigned short aa, aa1, len, aa2;
unsigned char data[ 8];
long id;
unsigned short zr, cont, oldstate;

//.....

void main() {
    PORTC = 0;
    TRISC = 0;
    PORTD = 0;
    TRISD = 0;
    aa = 0;
    aa1 = 0;
    aa2 = 0;

    // Form value to be used with CANSendMessage
    aa1 = CAN_TX_PRIORITY_0 &
          CAN_TX_XTD_FRAME &
          CAN_TX_NO_RTR_FRAME;

    // Form value to be used with CANInitialize
    aa = 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;

    data[ 0] = 0;

    // Initialize CAN
    CANInitialize(1,1,3,3,1,aa);

    // Set CAN to CONFIG mode
    CANSetOperationMode(CAN_MODE_CONFIG,0xFF);

    id = -1;

    // continues ..
}
```

```

// .. continued

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

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

// Set id of filter B1_F1 to 3
CANSetFilter(CAN_FILTER_B2_F3, 3, CAN_CONFIG_XTD_MSG);

// Set CAN to NORMAL mode
CANSetOperationMode(CAN_MODE_NORMAL, 0xFF);

PORTD = 0xFF;
id = 12111;
CANWrite(id, data, 1, aa1);           // Send message via CAN

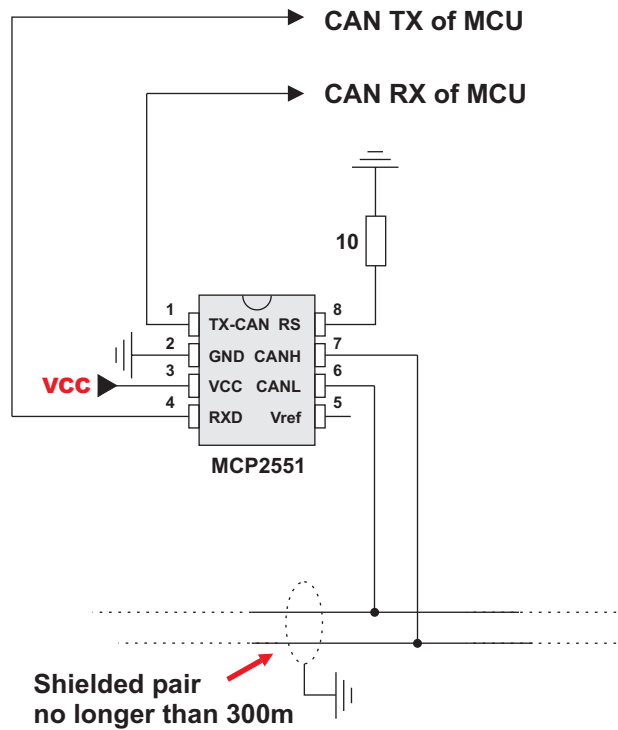
while (1) {
    oldstate = 0;
    zr = CANRead(&id, data, &len, &aa2);
    if ((id == 3) & zr) {
        PORTD = 0xAA;
        PORTC = data[0];           // Output data at PORTC
        data[0]++;

        // If message contains two data bytes, output second byte at PORTD
        if (len == 2) PORTD = data[1];

        data[1] = 0xFF;
        id = 12111;
        CANWrite(id, data, 2, aa1); // Send incremented data back
    }
}
} //~!

```

Hardware Connection



CANSPI Library

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

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

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

Note: Be sure to check CAN constants necessary for using some of the functions. See page 145.

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

Library Routines

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

Following routines are for the internal use by compiler only:

```
RegsToCANSPIID  
CANSPIIDToRegs
```


CANSPISetOperationMode

Prototype	<code>void CANSPISetOperationMode(char mode, char wait_flag);</code>
Description	<p>Sets CAN to requested mode, i.e. copies mode to CANSTAT. Parameter <code>mode</code> needs to be one of CAN_OP_MODE constants (see CAN constants, page 145).</p> <p>Parameter <code>wait_flag</code> needs to be either 0 or 0xFF: If set to 0xFF, this is a blocking call – the function won't "return" until the requested mode is set. If 0, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use function <code>CANSPIGetOperationMode</code> to verify correct operation mode before performing mode specific operation.</p>
Requires	CANSPI functions are supported by any PIC MCU that has SPI interface on PORTC. Also, CS pin of MCP2510 or MCP2515 must be connected to RC0.
Example	<code>CANSPISetOperationMode(CAN_MODE_CONFIG, 0xFF);</code>

CANSPIGetOperationMode

Prototype	<code>char CANSPIGetOperationMode(void);</code>
Returns	Current opmode.
Description	Function returns current operational mode of CAN module.
Example	<code>if (CANSPIGetOperationMode() == CAN_MODE_NORMAL) { ... };</code>

CANSPIInitialize

Prototype	<pre>void CANSPIInitialize(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS, char * RstPort, char RstPin, char * CSPort, char CSPin);</pre>
Description	<p>Initializes CANSPI. All pending transmissions are aborted. Sets all mask registers to 0 to allow all messages.</p> <p>Filter registers are set according to flag value:</p> <pre>if (CAN_CONFIG_FLAGS & CAN_CONFIG_VALID_XTD_MSG != 0) // Set all filters to XTD_MSG else if (config & CONFIG_VALID_STD_MSG != 0) // Set all filters to STD_MSG else // Set half the filters to STD, and the rest to XTD_MSG</pre> <p>Parameters:</p> <p>SJW as defined in 18XXX8 datasheet (1–4) BRP as defined in 18XXX8 datasheet (1–64) PHSEG1 as defined in 18XXX8 datasheet (1–8) PHSEG2 as defined in 18XXX8 datasheet (1–8) PROPSEG as defined in 18XXX8 datasheet (1–8) CAN_CONFIG_FLAGS is formed from predefined constants (see CAN constants, page 145).</p>
Requires	<p>SPI_Init() must be called before initializing CANSPI. CANSPI must be in Config mode; otherwise the function will be ignored.</p>
Example	<pre>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;</pre> <p>...</p> <pre>// initialize external CAN module CANSPIInitialize(1,1,3,3,1,init, &PORTC, 2, &PORTC, 0);</pre>

CANSPISetBaudRate

Prototype	<code>void CANSPISetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);</code>
Description	<p>Sets CANSPI baud rate. Due to complexity of CANSPI protocol, you cannot simply force a bps value. Instead, use this function when CANSPI is in Config mode. Refer to datasheet for details.</p> <p>Parameters:</p> <p>SJW as defined in 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)</p>
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	<pre>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; ... CANSPISetBaudRate(1, 1, 3, 3, 1, init);</pre>

CANSPISetMask

Prototype	void CANSPISetMask(char CAN_MASK, long value, char CAN_CONFIG_FLAGS);
Description	<p>Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers.</p> <p>Parameters: CAN_MASK is one of predefined constant values (see CAN constants); value is the mask register value; CAN_CONFIG_FLAGS selects type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG.</p>
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.
Example	<pre>// Set all mask bits to 1, i.e. all filtered bits are relevant: CANSPISetMask(CAN_MASK_B1, -1, CAN_CONFIG_XTD_MSG); /* Note that -1 is just a cheaper way to write 0xFFFFFFFF. Complement will do the trick and fill it up with ones. */</pre>

CANSPISetFilter

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

CANSPIRead

Prototype	<code>char CANSPIRead(long *id, char *data, char *datalen, char *CAN_RX_MSG_FLAGS);</code>
Returns	Message from receive buffer or zero if no message found.
Description	<p>Function reads message from receive buffer. If at least one full receive buffer is found, it is extracted and returned. If none found, function returns zero.</p> <p>Parameters: <code>id</code> is message identifier; <code>data</code> is an array of bytes up to 8 bytes in length; <code>datalen</code> is data length, from 1–8; <code>CAN_RX_MSG_FLAGS</code> is value formed from constants (see CAN constants).</p>
Requires	CANSPI must be in mode in which receiving is possible.
Example	<pre>char rcv, rx, len, data[8]; long id; rcv = CANSPIRead(id, data, len, 0);</pre>

CANSPIWrite

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

Library Example

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

```
char data[ 8],aa, aa1, len, aa2;
long id;
char zr;
const char _TRUE = 0xFF;
const char _FALSE = 0x00;

void main(){
    TRISB = 0;
    Spi_Init(); // Initialize SPI module
    TRISC.F2 = 0; // Clear (TRISC,2)
    PORTC.F2 = 0; // Clear (PORTC,2)
    PORTC.F0 = 1; // Set (PORTC,0)
    TRISC.F0 = 0; // Clear (TRISC,0)
    PORTD = 0;
    TRISD = 0;
    aa = 0;
    aa1 = 0;
    aa2 = 0;

    // Form value to be used with CANSPIInitialize
    aa = CAN_CONFIG_SAMPLE_THRICE &
        CAN_CONFIG_PHSEG2_PRG_ON &
        CAN_CONFIG_STD_MSG &
        CAN_CONFIG_DBL_BUFFER_ON &
        CAN_CONFIG_VALID_XTD_MSG;

    PORTC.F2 = 1; // Set (PORTC,2)

    // Form value to be used with CANSPISendMessage
    aa1 = CAN_TX_PRIORITY_0 &
        CAN_TX_XTD_FRAME &
        CAN_TX_NO_RTR_FRAME;

    PORTC.F0 = 1; // Set (PORTC,0)

    // continues ..
```

```
// .. continued

Spi_Init(); // Initialize SPI

// Initialize external CAN module
CANSPIInitialize( 1,1,3,3,1,aa, &PORTC, 2, &PORTC, 0);

// Set CANSPI to CONFIG mode
CANSPISetOperationMode(CAN_MODE_CONFIG,_TRUE);
ID = -1;

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

// Set all mask2 bits to ones
CANSPISetMask(CAN_MASK_B2,id,CAN_CONFIG_XTD_MSG);

// Set id of filter B1_F1 to 12111
CANSPISetFilter(CAN_FILTER_B2_F4,12111,CAN_CONFIG_XTD_MSG);

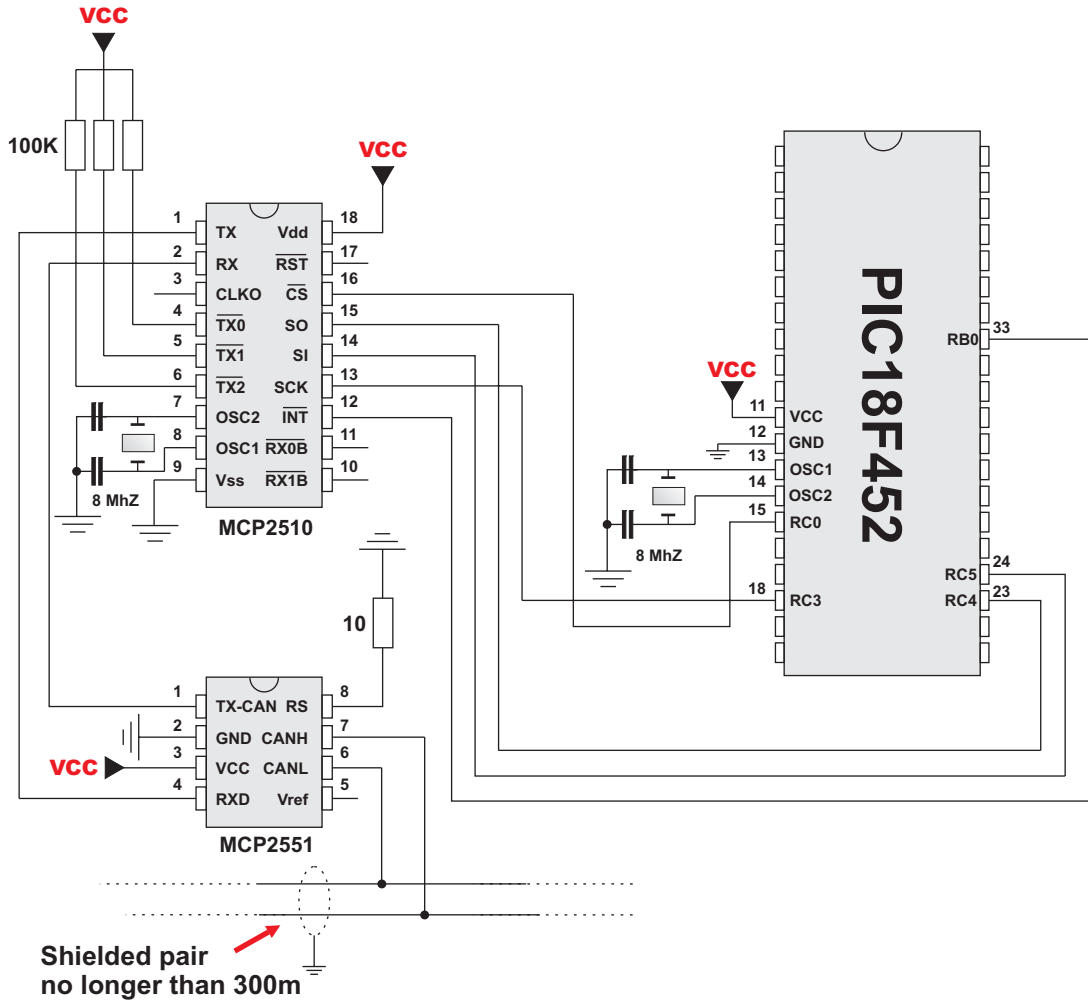
// Set CANSPI to NORMAL mode
CANSPISetOperationMode(CAN_MODE_NORMAL,_TRUE);

while (1) {
    zr = CANSPIRead(&id , &Data , &len, &aa2); // Receive data, if any
    if (id == 12111 & zr ) {
        PORTB = data[0]++ ; // Output data on PORTB
        id = 3;
        Delay_ms(500);

        // Send incremented data back
        CANSPIWrite(id,&data,1,aa1);

        // If message contains 2 data bytes, output second byte at PORTD
        if (len == 2) PORTD = data[1];
    }
}
} //~!
```

Hardware Connection



Compact Flash Library

Compact Flash Library provides routines for accessing data on Compact Flash card (abbrev. CF further in text). CF cards are widely used memory elements, commonly found in digital cameras. Great capacity (8MB ~ 2GB, and more) and excellent access time of typically few microseconds make them very attractive for microcontroller applications.

In CF card, data is divided into sectors, one sector usually comprising 512 bytes (few older models have sectors of 256B). Read and write operations are not performed directly, but successively through 512B buffer. Following routines can be used for CF with FAT16, and FAT32 file system. Note that routines for file handling can be used only with FAT16 file system.

Important! Before write operation, make sure you don't overwrite boot or FAT sector as it could make your card on PC or digital cam unreadable. Drive mapping tools, such as Winhex, can be of a great assistance.

Library Routines

```
Cf_Init  
Cf_Detect  
Cf_Total_Size  
Cf_Enable  
Cf_Disable  
Cf_Read_Init  
Cf_Read_Byte  
Cf_Write_Init  
Cf_Write_Byte  
  
Cf_Fat_Init  
Cf_Fat_Assign  
Cf_Fat_Reset  
Cf_Fat_Read  
Cf_Fat_Rewrite  
Cf_Fat_Append  
Cf_Fat_Delete  
Cf_Fat_Write  
Cf_Fat_Set_File_Date  
Cf_Fat_Get_File_Date  
Cf_Fat_Get_File_Size
```

Function Cf_Set_Reg_Adr is for compiler internal purpose only.

Cf_Init

Prototype	<code>void Cf_Init(char *ctrlport, char *dataport);</code>
Description	Initializes ports appropriately for communication with CF card. Specify two different ports: ctrlport and dataport.
Example	<code>Cf_Init(&PORTB, &PORTD);</code>

Cf_Detect

Prototype	<code>char Cf_Detect(void);</code>
Returns	Returns 1 if CF is present, otherwise returns 0.
Description	Checks for presence of CF card on ctrlport.
Example	<pre>// Wait until CF card is inserted: do nop; while (Cf_Detect() == 0);</pre>

Cf_Total_Size

Prototype	<code>unsigned long Cf_Total_Size(void);</code>
Returns	Card size in kilobytes.
Description	Returns size of Compact Flash card in kilobytes.
Requires	Ports must be initialized. See Cf_Init.
Example	<code>size = Cf_Total_Size();</code>

Cf_Enable

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

Cf_Disable

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

Cf_Read_Init

Prototype	<code>void Cf_Read_Init(long address, char sectcnt);</code>
Description	Initializes CF card for reading. Parameter <code>address</code> specifies sector address from where data will be read, and <code>sectcnt</code> is the number of sectors prepared for reading operation.
Requires	Ports must be initialized. See <code>Cf_Init</code> .
Example	<code>Cf_Read_Init(590, 1);</code>

Cf_Read_Byte

Prototype	<code>char Cf_Read_Byte(void);</code>
Returns	Returns byte from CF.
Description	Reads one byte from CF.
Requires	CF must be initialized for read operation. See <code>Cf_Read_Init</code> .
Example	<code>PORTC = Cf_Read_Byte(); // Read byte and display it on PORTC</code>

Cf_Write_Init

Prototype	<code>void Cf_Write_Init(long address, char sectcnt);</code>
Description	Initializes CF card for writing. Parameter <code>address</code> specifies sector address where data will be stored, and <code>sectcnt</code> is total number of sectors prepared for write operation.
Requires	Ports must be initialized. See <code>Cf_Init</code> .
Example	<code>Cf_Write_Init(590, 1);</code>

Cf_Write_Byte

Prototype	<code>void Cf_Write_Byte(char data);</code>
Description	Writes one byte (<code>data</code>) to CF. All 512 bytes are transferred to a buffer.
Requires	CF must be initialized for write operation. See <code>Cf_Write_Init</code> .
Example	<code>Cf_Write_Byte(100);</code>

Cf_Fat_Init

Prototype	<code>void Cf_Fat_Init(unsigned short *control_port, unsigned short wr, rd, a2, a1, a0, ry, ce, cd, unsigned short *data_port);</code>
Returns	Returns 0 if initialization is successful, 1 if boot sector was not found and 255 if card was not detected.
Description	Initializes ports appropriately for FAT operations with CF card. Specify two different ports: ctrlport and dataport. wr, rd, a2, a1, a0, ry, ce and cd are pin numbers on control port.
Requires	Nothing.
Example	<code>Cf_Fat_Init(PORTD, 6, 5, 2, 1, 0, 7, 3, 4, PORTC);</code>

Cf_Fat_Assign

Prototype	<code>unsigned short Cf_Fat_Assign(char *filename, char create_file);</code>
Returns	"1" if file is present (or file isn't present but new file is created), or "0" if file isn't present and no new file is created.
Description	Assigns file for FAT operations. If file isn't present, function creates new file with given filename. filename parameter is name of file (filename must be in format 8.3 UPPER-CASE). create_file is a parameter for creating new files. if create_file if different from 0 then new file is created (if there is no file with given filename).
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init.
Example	<code>Cf_Fat_Assign('MIKROELE.TXT', 1);</code>

Cf_Fat_Reset

Prototype	<code>void Cf_fat_Reset(unsigned long *size);</code>
Returns	Size of file in bytes. Size is stored on address of input variable.
Description	Opens assigned file for reading.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	<code>Cf_Fat_Reset(size);</code>

Cf_Fat_Read

Prototype	<code>void Cf_Fat_Read(unsigned short *bdata);</code>
Description	Reads data from file. bdata is data read from file.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign. File must be open for reading. See Cf_Fat_Reset.
Example	<code>Cf_Fat_Read(character);</code>

Cf_Fat_Rewrite

Prototype	<code>void Cf_Fat_Rewrite();</code>
Returns	Nothing.
Description	Rewrites assigned file.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	<code>Cf_Fat_Rewrite;</code>

Cf_Fat_Append

Prototype	<code>void Cf_fat_Append();</code>
Returns	Nothing.
Description	Opens file for writing. This procedure continues writing from the last byte in file.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	<code>Cf_Fat_Append;</code>

Cf_Fat_Delete

Prototype	<code>void Cf_Fat_Delete();</code>
Description	Deletes file from CF.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	<code>Cf_Fat_Delete;</code>

Cf_Fat_Write

Prototype	<code>void Cf_Fat_Write(char *fdata, unsigned data_len);</code>
Returns	Nothing.
Description	Writes data to CF. <i>fdata</i> parameter is data written to CF. <i>data_len</i> number of bytes that is written to CF.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.
Example	<code>Cf_Fat_Write(file_contents, 42); // write data to the assigned file</code>

Cf_Fat_Set_File_Date

Prototype	<code>void Cf_fat_Set_File_Date(unsigned int year, unsigned short month, unsigned short day, unsigned short hours, unsigned short mins, unsigned short seconds);</code>
Returns	Nothing.
Description	Sets time attributes of file. You can set file year, month, day. hours, mins, seconds.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign. File must be open for writing. See Cf_Fat_Rewrite or Cf_Fat_Append.
Example	<code>Cf_Fat_Set_File_Date(2005, 9, 30, 17, 41, 0);</code>

Cf_Fat_Get_File_Date

Prototype	<code>void Cf_Fat_Get_File_Date(unsigned int *year, unsigned short *month, unsigned short *day, unsigned short *hours, unsigned short *mins);</code>
Description	Reads time attributes of file. You can read file year, month, day, hours, mins.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	<code>Cf_Fat_Get_File_Date(year, month, day, hours, mins);</code>

Cf_Fat_Get_File_Size

Prototype	<code>unsigned long Cf_fat_Get_File_Size();</code>
Returns	Size of file in bytes.
Description	This function returns size of file in bytes.
Requires	Ports must be initialized for FAT operations with CF. See Cf_Fat_Init. File must be assigned. See Cf_Fat_Assign.
Example	<code>Cf_Fat_Get_File_Size;</code>

Library Example

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

```
unsigned i;

void main() {
    TRISC = 0;           // PORTC is output
    Cf_Init(&PORTB, &PORTD); // Initialize ports

    do nop;
    while (!Cf_Detect()); // Wait until CF card is inserted

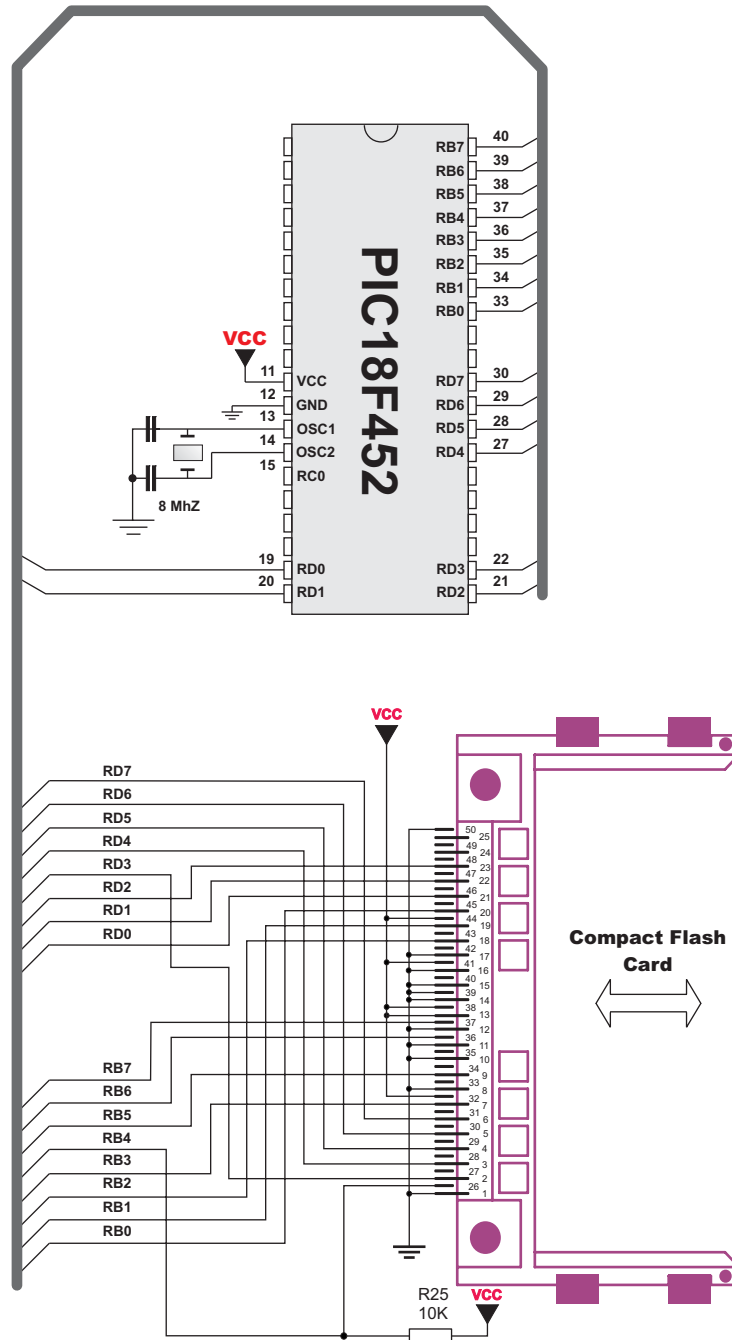
    Delay_ms(500);
    Cf_Write_Init(590, 1); // Initialize write at sector address 590

    // Write 512 bytes to sector (590)
    for (i = 0; i < 512; i++) Cf_Write_Byte(i + 11);

    PORTC = 0xFF;
    Delay_ms(1000);
    Cf_Read_Init(590, 1); // Initialize read at sector address 590

    // Read 512 bytes from sector (590)
    for (i = 0; i < 512; i++) {
        PORTC = Cf_Read_Byte(); // Read byte and display on PORTC
        Delay_ms(1000);
    }
}
```

HW Connection



Compact Flash FAT Library v2.xx

This is Compact Flash FAT library from previous version (v2.1). This library is included because of users that have developed projects with old CF library.

NOTE

This version of Compact Flash FAT library is deprecated. There will be no longer development for this version of library. Please use new version of Compact Flash library for your projects.

Important! File accessing routines can write file. File names must be exactly 8 characters long and written in uppercase. User must ensure different names for each file, as CF routines will not check for possible match.

Important! Before write operation, make sure you don't overwrite boot or FAT sector as it could make your card on PC or digital cam unreadable. Drive mapping tools, such as Winhex, can be of a great assistance.

Library Routines

```
Cf_Find_File
Cf_File_Write_Init
Cf_File_Write_Byte
Cf_Read_Sector
Cf_Write_Sector
Cf_Set_File_Date
Cf_File_Write_Complete
```

Cf_Find_File

Prototype	<code>void Cf_Find_File(unsigned short find_first, char *file_name);</code>
Description	Routine looks for files on CF card. Parameter <code>find_first</code> can be non-zero or zero; if non-zero, routine looks for the first file on card, in order of physical writing. Otherwise, routine "moves forward" to the next file from the current position, again in physical order. If file is found, routine writes its name and extension in the string <code>file_name</code> . If no file is found, the string will be filled with zeroes.
Requires	Ports must be initialized.
Example	<pre>Cf_Find_File(1, file); if (file[0]) { ...// if first file found, handle it}</pre>

Cf_File_Write_Init

Prototype	<code>void Cf_File_Write_Init(void);</code>
Description	Initializes CF card for file writing operation (FAT16 only).
Requires	Ports must be initialized. See Cf_Init.
Example	<code>Cf_File_Write_Init();</code>

Cf_File_Write_Byte

Prototype	<code>void Cf_File_Write_Byte(char data);</code>
Description	Adds one byte (data) to file. You can supply ASCII value as parameter, for example 48 for zero.
Requires	CF must be initialized for file write operation. See Cf_File_Write_Init.
Example	<pre>// Write 50,000 zeroes (bytes) to file: for (i = 0; i < 50000; i++) Cf_File_Write_Byte(48);</pre>

Cf_Read_Sector

Prototype	<code>void Cf_Read_Sector(int sector_number, unsigned short *buffer);</code>
Description	Reads one sector (sector_number) into buffer.
Requires	CF must be initialized for file write operation. See Cf_Init.
Example	<code>Cf_Read_Sector(22, data);</code>

Cf_Write_Sector

Prototype	<code>void Cf_Write_Sector(int sector_number, unsigned short *buffer);</code>
Description	Writes value from buffer to CF sector at sector_number.
Requires	CF must be initialized for file write operation. See Cf_Init.
Example	<code>Cf_Write_Sector(22, data);</code>

Cf_Set_File_Date

Prototype	<code>void Cf_Set_File_Date(int year, char month, day, hours, min, sec);</code>
Description	Writes system timestamp to a file. Use this routine before finalizing a file; otherwise, file will be appended a random timestamp.
Requires	CF must be initialized for file write operation. See Cf_File_Write_Init.
Example	<code>// April 1st 2005, 18:07:00 Cf_Set_File_Date(2005,4,1,18,7,0);</code>

Cf_File_Write_Complete

Prototype	<code>void Cf_File_Write_Complete(char filename[8], char *extension);</code>
Description	Finalizes writing to file. Upon all data has be written to file, use this function to close the file and make it readable. Parameter filename must be 8 chars long in uppercase.
Requires	CF must be initialized for file write operation. See Cf_File_Write_Init.
Example	<code>Cf_File_Write_Complete("MY_FILE1", "txt");</code>

EEPROM Library

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

Library Routines

```
Eeprom_Read
Eeprom_Write
```

Eeprom_Read

Prototype	<code>unsigned short Eeprom_Read(unsigned short address);</code>
Returns	Returns byte from the 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 <code>Eeprom_Write</code> and <code>Eeprom_Read</code> . Although PIC will write the correct value, <code>Eeprom_Read</code> might return an undefined result.
Example	<pre>char take; ... take = Eeprom_Read(0x3F);</pre>

Eeprom_Read

Prototype	<code>void Eeprom_Write(unsigned int address, unsigned short data);</code>
Description	<p>Writes data to specified address. PParameter address is of integer type, which means it supports MCUs with more than 256 bytes of EEPROM.</p> <p>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.</p>
Requires	<p>Requires EEPROM module.</p> <p>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.</p>
Example	<code>Eeprom_Write(0x32);</code>

Library Example

```

unsigned short i = 0, j = 0;

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

    j = 4;
    for (i = 0; i < 20u; i++)
        Eeprom_Write(i, j++);

    for (i = 0; i < 20u; i++) {
        PORTB = Eeprom_Read(i);
        Delay_ms(500);
    }
} //~!

```

Ethernet Library

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

Library Routines

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


Eth_Init

Prototype	<code>void Eth_Init(char *addrP, char *dataP, char *ctrlP, char pinReset, char pinIOW, char pinIOR);</code>
Description	<p>Performs initialization of Ethernet card and library. This includes:</p> <ul style="list-style-type: none"> - Setting of control and data ports; - Initialization of the Ethernet card (also called the Network Interface Card, or NIC); - Retrieval and local storage of the NIC's hardware (MAC) address; - Putting the NIC into the LISTEN mode. <p>Parameter <code>addrP</code> is a pointer to address port, which handles the addressing lines. Parameter <code>dataP</code> is pointer to data port. Parameter <code>ctrlP</code> is the control port. Parameter <code>pinReset</code> is the reset/enable pin for the ethernet card chip (on control port). Parameter <code>pinIOW</code> is the I/O Write request control pin. Parameter <code>pinIOR</code> is the I/O read request control pin.</p>
Requires	As specified for the entire library (please see top of this page).
Example	<code>Eth_Init(&PORTB, &PORTD, &PORTE, 2, 1, 0);</code>

Eth_Set_Ip_Address

Prototype	<code>void Eth_Set_Ip_Address(char ip1, char ip2, char ip3, char ip4);</code>
Description	Sets the IP address of the connected and initialized Ethernet network card. The arguments are the IP address numbers, in IPv4 format (e.g. 127.0.0.1).
Requires	This function should be called immediately after the NIC initialization (see <code>Eth_Init</code>). You can change your IP address at any time, anywhere in the code.
Example	<pre>// Set IP address 192.168.20.25 Eth_Set_Ip_Address(192u, 168u, 20u, 25u);</pre>

Eth_Set_Inport

Prototype	<code>unsigned short Eth_Inport(unsigned short address);</code>
Returns	One byte from the specified address.
Description	Retrieves a byte from the specified address of the Ethernet card chip.
Requires	The card (NIC) must be properly initialized. See <code>Eth_Init</code> .
Example	<code>udp_length = Eth_Inport(NIC_DATA);</code>

Eth_Scan_For_Event

Prototype	<code>unsigned Eth_Scan_For_Event(unsigned short *next_ptr);</code>
Returns	Type of the ethernet packet received. Two types are distinguished: ARP (MAC-IP address data request) and IP (Internet Protocol).
Description	Retrieves sender's MAC (hardware) address and type of the packet received. The function argument is an (internal) pointer to the next data packet in RTL8019's buffer, and is of no particular importance to the end user.
Requires	The card (NIC) must be properly initialized. See <code>Eth_Init</code> . Also, the function must be called in a proper sequence, i.e. right after the card init, and IP address/UDP port init.
Example	<pre> Eth_Init(&PORTB, &PORTD, &PORTE, 2, 1, 0); Eth_Set_Ip_Address(192u, 168u, 20u, 25u); Eth_Set_Udp_Port(10001); do { // Main block of every Ethernet example event_type = Eth_Scan_For_Event(&next_ptr); if (event_type) { switch (event_type) { case ARP: Arp_Event(); break; case IP : Ip_Event(); } Eth_Outport(CR, 0x22); Eth_Outport(BNDRY, next_ptr); } } while (1); </pre>

Eth_Get_Ip_Hdr_Len

Prototype	<code>unsigned short Eth_Get_Ip_Hdr_Len(void);</code>
Returns	Header length of the received IP packet.
Description	Returns header length of the received IP packet. Before other data based upon the IP protocol (TCP, UDP, ICMP) can be analyzed, the sub-protocol data must be properly loaded from the received IP packet.
Requires	The card (NIC) must be properly initialized. See <code>Eth_Init</code> . The function must be called in a proper sequence, i.e. immediately after determining that the packet received is the IP packet.
Example	<pre>// Receive IP Header opt_len = Eth_Get_Ip_Hdr_Len() - 20;</pre>

Eth_Load_Ip_Packet

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

Eth_Get_Hdr_Chksum

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

Eth_Get_Source_Ip_Address

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

Eth_Get_Dest_Ip_Address

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

Eth_Arp_Response

Prototype	<code>void Eth_Arp_Response(void);</code>
Description	An automated ARP response. User should simply call this function once he detects the ARP event on the NIC.
Requires	As specified for the entire library.
Example	<code>Eth_Arp_Response();</code>

Eth_Get_Icmp_Info

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

Eth_Ping_Response

Prototype	<code>void Eth_Ping_Response(void);</code>
Description	An automated ICMP (Ping) response. User should call this function when answering to an ICMP/IP event.
Requires	As specified for the entire library.
Example	<code>Eth_Ping_Response();</code>

Eth_Get_Udp_Source_Port

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

Eth_Get_Udp_Dest_Port

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

Eth_Get_Udp_Port

Prototype	<code>unsigned short Eth_Get_Udp_Port(void);</code>
Returns	Returns the UDP port (socket) number that is set for the PIC's Ethernet card.
Description	The function returns the UDP port (socket) number that is set for the PIC's Ethernet card. After the UDP port is set at the beginning of the session (<code>Eth_Set_Udp_Port</code>), its number is later used to test whether the received UDP packet is targeted at the port we are using.
Requires	The network card must be properly initialized (see <code>Eth_Init</code>), and the UDP port properly set (see <code>Eth_Set_Udp_Port</code>). This library currently supports working with only one UDP port (socket) at a time.
Example	<pre> if (udp_dest_port == Eth_Get_Udp_Port()) { ... // Respond to action } </pre>

Eth_Set_Udp_Port

Prototype	<code>void Eth_Set_Udp_Port(unsigned udp_port);</code>
Description	Sets up the default UDP port, which will handle user requests. The user can decide, upon receiving the UDP packet, which port was this packet sent to, and whether it will be handled or rejected.
Requires	As specified for the entire library.
Example	<code>Eth_Set_Udp_Port(10001);</code>

Eth_Send_Udp

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

Eth_Load_Tcp_Header

Prototype	<code>void Eth_Load_Tcp_Header(void);</code>
Description	Loads various TCP Header data into PIC's Ethernet variables.
Requires	This function must be called in a proper sequence, i.e. immediately after retrieving the source and destination port (socket) of the TCP message.
Example	<pre>// retrieve 'source port' tcp_source_port = Eth_Inport(NIC_DATA) << 8; tcp_source_port = Eth_Inport(NIC_DATA); // retrieve 'destination port' tcp_dest_port = Eth_Inport(NIC_DATA) << 8; tcp_dest_port = Eth_Inport(NIC_DATA); // We only respond to port 80 (HTML requests) if (tcp_dest_port == 80u) { Eth_Load_Tcp_Header(); // retrieve TCP Header data (most of it) //... }</pre>

Eth_Get_Tcp_Hdr_Offset

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

Eth_Get_Tcp_Flags

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

Eth_Set_Tcp_Data

Prototype	<code>void Eth_Set_Tcp_Data(const unsigned short *data);</code>
Description	Prepares data to be sent on HTTP request. This library can handle only HTTP requests, so sending other TCP-based protocols, such as FTP, will cause an error. Note that TCP/IP was not designed with 8-bit MCU's in mind, so be gentle with your HTTP requests.
Requires	As specified for the entire library.
Example	<pre>// Let's prepare a simple HTML page in our string: const char httpPage1[] = "HTTP/1.0 200 OK\nContent-type: text/html\n" "<html>\n" "<body>\n" "<h1>Hello world!</h1>\n" "</body>\n" "</html>"; //... Eth_Set_Tcp_Data(httpPage1); //...</pre>

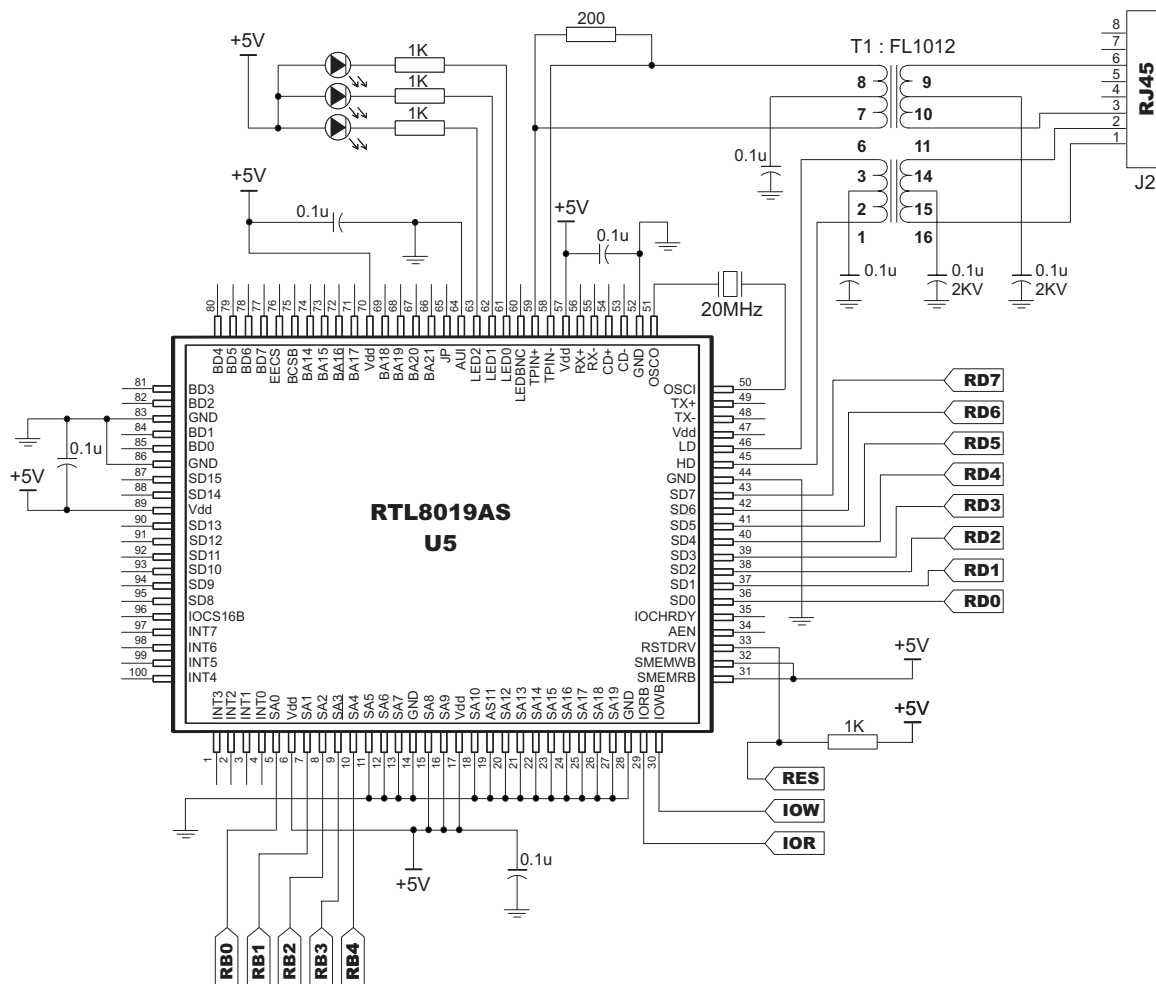
Eth_Tcp_Response

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

Library Example

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

HW Connection



SPI Ethernet Library

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

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

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

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

Note: SPI_Init() must be called before initializing SPI Ethernet.

Library Routines

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

SPI_Ethernet_Init

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

SPI_Ethernet_doPacket

Prototype	<code>void SPI_Ethernet_doPacket();</code>
Returns	Nothing.
Description	Process one incoming packet if available. This function is public.
Requires	SPI_Ethernet_init must have been called before this function must be called as often as possible by use.
Example	<code>SPI_Ethernet_doPacket();</code>

SPI_Ethernet_putByte

Prototype	<code>void SPI_Ethernet_putByte(unsigned char v);</code>
Returns	Nothing.
Description	<code>v</code> - value to store Store one byte to current EWRPT ENC location this function is public.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	<code>SPI_Ethernet_putByte(0xa0);</code>

SPI_Ethernet_getByte

Prototype	<code>unsigned char SPI_Ethernet_getByte();</code>
Returns	Value of byte @ addr.
Description	Get next byte from current ERDPT ENC location this function is public.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	<code>b = SPI_Ethernet_getByte();</code>

SPI_Ethernet_UserTCP

Prototype	<code>unsigned int SPI_Ethernet_UserTCP(unsigned char *remoteHost, unsigned int remotePort, unsigned int localPort, unsigned int reqLength);</code>
Returns	Returns the length in bytes of the HTTP reply, or 0 if nothing to transmit.
Description	This function is called by the library. The user accesses to the HTTP request by successive calls to SPI_Ethernet_getByte() the user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte() the function must return the length in bytes of the HTTP reply, or 0 if nothing to transmit. If you don't need to reply to HTTP requests, just define this function with a return(0) as single statement.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	

SPI_Ethernet_UserUDP

Prototype	<code>unsigned int SPI_Ethernet_UserUDP(unsigned char *remoteHost, unsigned int remotePort, unsigned int destPort, unsigned int reqLength);</code>
Returns	Returns the length in bytes of the UDP reply, or 0 if nothing to transmit.
Description	This function is called by the library. The user accesses to the UDP request by successive calls to SPI_Ethernet_getByte(). The user puts data in the transmit buffer by successive calls to SPI_Ethernet_putByte(). The function must return the length in bytes of the UDP reply, or 0 if nothing to transmit. If you don't need to reply to UDP requests, just define this function with a return(0) as single statement.
Requires	SPI_Ethernet_init must have been called before calling this function.
Example	

Library Example

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

```
#define SPI_Ethernet_HALFDUPLEX      0
#define SPI_Ethernet_FULLDUPLEX     1

/*****
 * ROM constant strings
 */
const unsigned char httpHeader[] = "HTTP/1.1 200 OK\nContent-type: " ; // HTTP header
const unsigned char httpMimeTypeHTML[] = "text/html\n\n" ;           // HTML MIME type
const unsigned char httpMimeTypeScript[] = "text/plain\n\n" ;       // TEXT MIME type
unsigned char httpMethod[] = "GET /";
/*
 * web page, splitted into 2 parts :
 * when coming short of ROM, fragmented data is handled more efficiently by linker
 *
 * this HTML page calls the boards to get its status, and builds itself with
 javascript
 */
const char *indexPage = "<HTML><HEAD></HEAD><BODY>\n
<h1>PIC + ENC28J60 Mini Web Server</h1>\n
<a href=/>Reload</a>\n
<script src=/s></script>\n
<table><tr><td valign=top><table border=1 style=\"font-size:20px ;font-family: termi-
nal ;\">\n
<tr><th colspan=2>ADC</th></tr>\n
<tr><td>AN2</td><td><script>document.write(AN2)</script></td></tr>\n
<tr><td>AN3</td><td><script>document.write(AN3)</script></td></tr>\n
</table></td><td><table border=1 style=\"font-size:20px ;font-family: terminal ;\">\n
<tr><th colspan=2>PORTB</th></tr>\n
<script>\n
var str,i;\n
str=\"\n\";\n
for(i=0;i<8;i++)\n
{ str+=\"<tr><td bgcolor=pink>BUTTON #\"+i+\"</td>\";\n
if(PORTB&(1<<i)){ str+=\"<td bgcolor=red>ON\";}\n
else { str+=\"<td bgcolor=#cccccc>OFF\";}\n
str+=\"</td></tr>\";\n
document.write(str) ;\n
```



```

</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>\
" ;
//                                str+=\"</td><td><a
href=/t\"+i+\">Toggle</a></td></tr>\";\
/*****
 * 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[ 31] ;                                       // 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) ;
    }

/*
 * 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) ;
    }

/*
 * 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 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 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 readable 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
    intToStr(remotePort, dyna + 16) ;
    dyna[ 22] = '[' ;
    intToStr(destPort, dyna + 23) ;
    dyna[ 29] = ']' ;
    dyna[ 30] = 0 ;
}

```

```

// the total length of the request is the length of the dynamic string plus the text
// of the request
    len = 30 + reqLength ;
    // puts the dynamic string into the transmit buffer
    ptr = dyna ;
    while(*ptr)
    {
        SPI_Ethernet_putByte(*ptr++) ;
    }

// then puts the request string converted into upper char into the transmit buffer
    while(reqLength--)
    {
        SPI_Ethernet_putByte(toupper(ENC28J60_getByte())) ;
    }

    return(len) ;    // back to the library with the length of the UDP reply
}

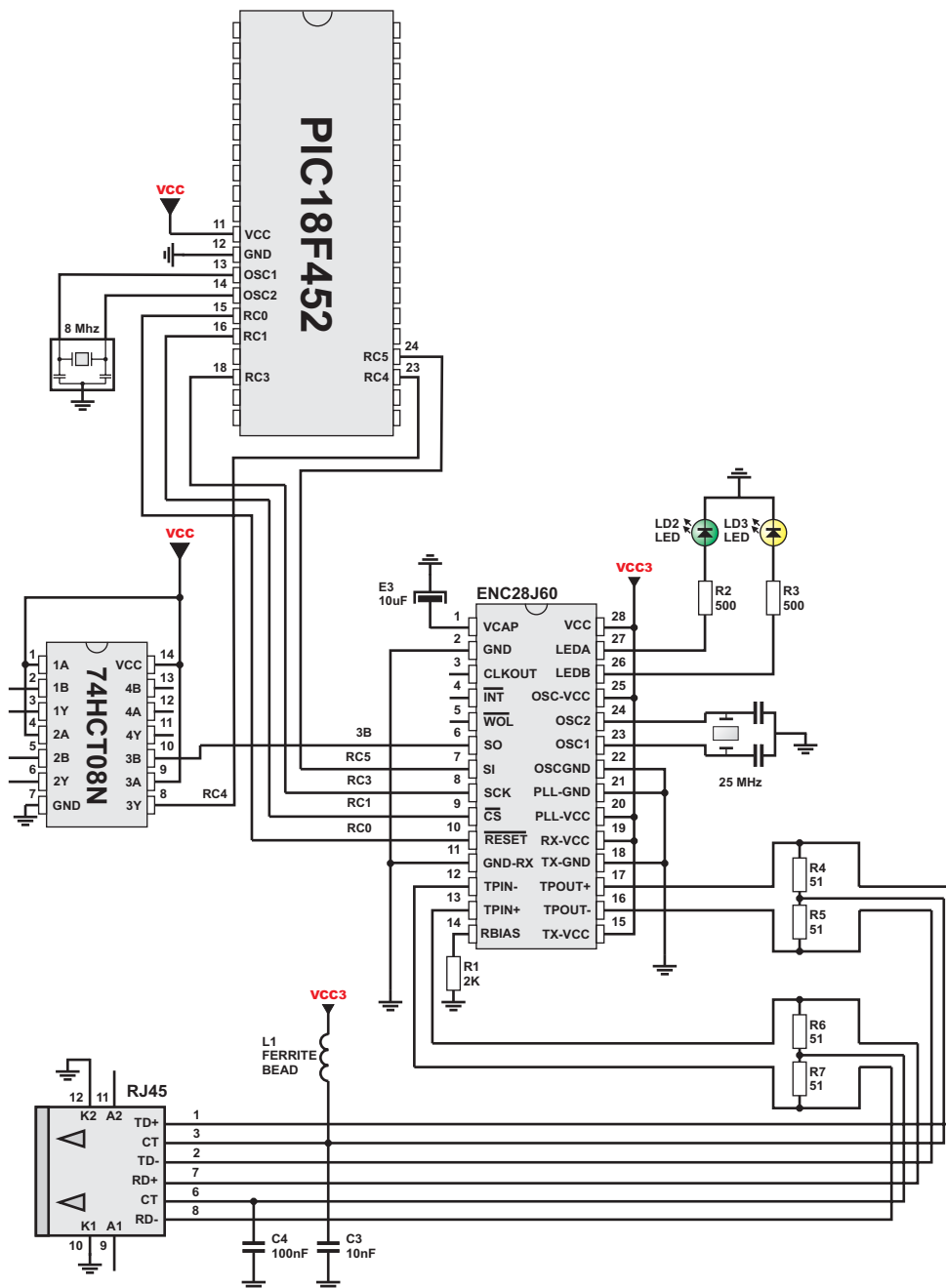
/*
 * main entry
 */
void main()
{
    ADCON1 = 0x00 ;           // ADC convertors will be used
    PORTA = 0 ;
    TRISA = 0xff ;          // set PORTA as input for ADC
    PORTB = 0 ;
    TRISB = 0xff ;          // set PORTB as input for buttons
    PORTD = 0 ;
    TRISD = 0 ;             // set PORTD as output
    /* starts ENC28J60 with :
     * reset bit on RC0
     * CS bit on RC1
     * my MAC & IP address
     * full duplex
     */
    Spi_Init();           // Initialize SPI module

    SPI_Ethernet_Init(&PORTC, 0, &PORTC, 1, myMacAddr, myIpAddr, SPI_Ethernet_FULLDUPLEX);
    while(1)               // do forever
    {
        SPI_Ethernet_doPacket() ;    // process incoming Ethernet packets

        /*
         * add your stuff here if needed
         * SPI_Ethernet_doPacket() must be called as often as possible
         * otherwise packets could be lost
         */
    }
}

```

HW Connection



Flash Memory Library

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

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

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

Please refer to datasheet before using flash library.

Library Routines

```
Flash_Read
Flash_Write
Flash_Erase
```

Flash_Read

Prototype	<code>unsigned Flash_Read(unsigned address); // for PIC16</code> <code>unsigned short Flash_Read(long address); // for PIC18</code>
Returns	Returns data byte from Flash memory.
Description	Reads data from the specified address in Flash memory.
Example	<code>Flash_Read(0x0D00);</code>

Flash_Write

Prototype	<pre>// for PIC16 void Flash_Write(unsigned address, unsigned int * data); // for PIC18 void Flash_Write(long address, unsigned short * fdata);</pre>
Description	Writes chunk of data to Flash memory. With PIC18, data needs to be exactly 64 bytes in size. Keep in mind that this function erases target memory before writing Data to it. This means that if write was unsuccessful, previous data will be lost.
Example	<pre>// Write consecutive values in 64 consecutive locations char toWrite[64]; // initialize array: for (i = 0; i < 63; i++) toWrite[i] = i; Flash_Write(0x0D00, toWrite);</pre>

Flash_Erase

Prototype	<pre>void Flash_Erase(unsigned address);</pre>
Description	Erases 32 bytes memory block starting from a given address. Implemented only for those MCU's whose flash memory does not support erase-and-write operations (refer to datasheet for details).
Example	<pre>// Erase 32 byte memory memory block, starting from address // \$0D00: Flash_Erase(\$0D00);</pre>

Library Example

The example demonstrates simple write to the flash memory for PIC18, then reads the data and displays it on PORTB.

```
unsigned short i = 0, j = 0;
unsigned long addr;
unsigned short dataRd;
unsigned short dataWr[ 64] =
    { 1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,
      1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,
      1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,
      1,2,3,4 };

void main() {
    PORTB = 0;
    TRISB = 0;
    PORTC = 0;
    TRISC = 0;

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

    addr = 0x00000A30;
    for (i = 0; i < 64; i++) {
        dataRd = Flash_Read(addr++);
        PORTB = dataRd;
        Delay_ms(500);
    }
} //~!
```

I2C Library

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

Note: Certain PICmicros with two I²C modules, such as P18F8722, require you to specify the module you want to use. Simply append the number 1 or 2 to a I2C. For example, `I2C2_Wr()`; Also, for the sake of backward compability with previous compiler versions and easier code management, MCU's with multiple I2C modules have I²C library which is identical to I2C1 (i.e. you can use `I2C_Init()` instead of `I2C1_Init()` for I²C operations).

Library Routines

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

I2C_Init

Prototype	<code>void I2C_Init(long clock);</code>
Description	Initializes I ² C with desired <code>clock</code> (refer to device data sheet for correct values in respect with <code>Fosc</code>). Needs to be called before using other functions of I2C Library.
Requires	Library requires MSSP module on PORTB or PORTC.
Example	<code>I2C_Init(100000);</code>

I2C_Start

Prototype	<code>char I2C_Start(void);</code>
Returns	If there is no error, function returns 0.
Description	Determines if I ² C bus is free and issues START signal.
Requires	I ² C must be configured before using this function. See I2C_Init.
Example	<code>I2C_Start();</code>

I2C_Repeated_Start

Prototype	<code>void I2C_Repeated_Start(void);</code>
Description	Issues repeated START signal.
Requires	I ² C must be configured before using this function. See I2C_Init.
Example	<code>I2C_Repeated_Start();</code>

I2C_Is_Idle

Prototype	<code>char I2C_Is_Idle(void);</code>
Returns	Returns 1 if I ² C bus is free, otherwise returns 0.
Description	Tests if I ² C bus is free.
Requires	I ² C must be configured before using this function. See I2C_Init.
Example	<code>if (I2C_Is_Idle()) { ... }</code>

I2C_Rd

Prototype	<code>char I2C_Rd(char ack);</code>
Returns	Returns one byte from the slave.
Description	Reads one byte from the slave, and sends not acknowledge signal if parameter <code>ack</code> is 0, otherwise it sends acknowledge.
Requires	START signal needs to be issued in order to use this function. See <code>I2C_Start</code> .
Example	<code>temp = I2C_Rd(0); // Read data and send not acknowledge signal</code>

I2C_Wr

Prototype	<code>char I2C_Wr(char data);</code>
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter <code>data</code>) via I ² C bus.
Requires	START signal needs to be issued in order to use this function. See <code>I2C_Start</code> .
Example	<code>I2C_Write(0xA3);</code>

I2C_Stop

Prototype	<code>void I2C_Stop(void);</code>
Description	Issues STOP signal.
Requires	I ² C must be configured before using this function. See <code>I2C_Init</code> .

Library Example

This code demonstrates use of I²C Library functions. PIC MCU is connected (SCL, SDA pins) to 24c02 EEPROM. Program sends data to EEPROM (data is written at address 2). Then, we read data via I²C from EEPROM and send its value to PORTD, to check if the cycle was successful (see the figure below how to interface 24c02 to PIC).

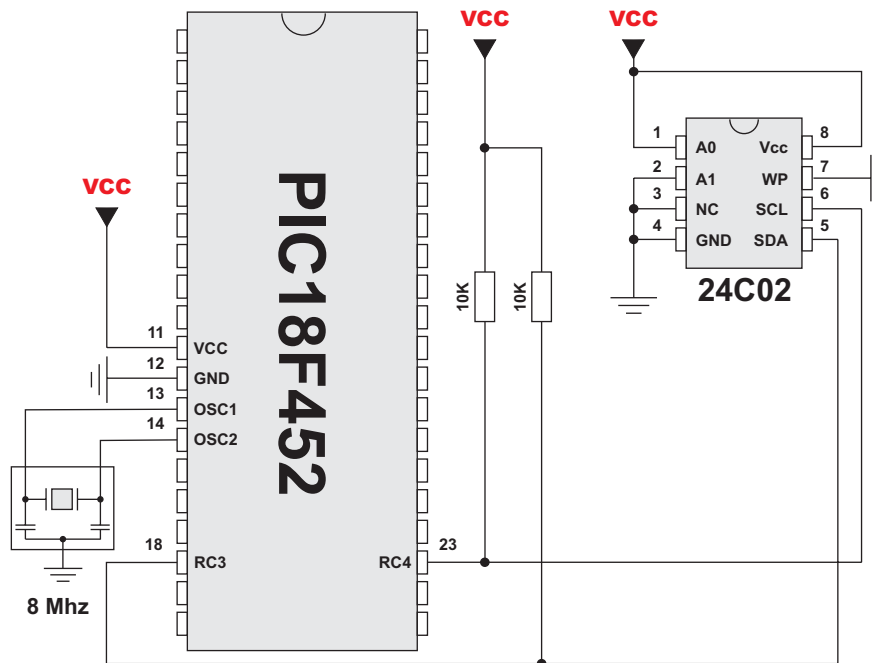
```
void main(){
    PORTB = 0;
    TRISB = 0;

    I2C_Init(100000);
    I2C_Start();           // Issue I2C start signal
    I2C_Wr(0xA2);         // Send byte via I2C (command to 24c02)
    I2C_Wr(2);           // Send byte (address of EEPROM location)
    I2C_Wr(0xF0);        // Send data (data to be written)
    I2C_Stop();

    Delay_ms(100);

    I2C_Start();         // Issue I2C start signal
    I2C_Wr(0xA2);        // Send byte via I2C (device address + W)
    I2C_Wr(2);          // Send byte (data address)
    I2C_Repeated_Start(); // Issue I2C signal repeated start
    I2C_Wr(0xA3);        // Send byte (device address + R)
    PORTB = I2C_Rd(0u);  // Read the data (NO acknowledge)
    I2C_Stop();
}
```

HW Connection



Keypad Library

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

Library Routines

```
Keypad_Init
Keypad_Read
Keypad_Released
```

Keypad_Init

Prototype	<code>void Keypad_Init(char *port);</code>
Description	Initializes <code>port</code> to work with keypad. The function needs to be called before using other routines of the Keypad library.
Example	<code>Keypad_Init(&PORTB);</code>

Keypad_Read

Prototype	<code>unsigned Keypad_Read(void);</code>
Returns	1..16, depending on the key pressed, or 0 if no key is pressed.
Description	Checks if any key is pressed. Function returns 1 to 16, depending on the key pressed, or 0 if no key is pressed.
Requires	Port needs to be appropriately initialized; see <code>Keypad_Init</code> .
Example	<code>kp = Keypad_Read();</code>

Keypad_Released

Prototype	<code>unsigned Keypad_Released(void);</code>
Returns	1..16, depending on the key.
Description	Call to <code>Keypad_Released</code> is a blocking call: function waits until any key is pressed and released. When released, function returns 1 to 16, depending on the key.
Requires	Port needs to be appropriately initialized; see <code>Keypad_Init</code> .
Example	<code>kp = Keypad_Released();</code>

Library Example

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

```

unsigned short kp, cnt;
char txt[ 5 ];

void main() {
    cnt = 0;
    Keypad_Init(&PORTC);
    Lcd_Init(&PORTB);           // Initialize LCD on PORTC
    Lcd_Cmd(LCD_CLEAR);       // Clear display
    Lcd_Cmd(LCD_CURSOR_OFF);  // Cursor off

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

    do {
        kp = 0;

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

        cnt++;

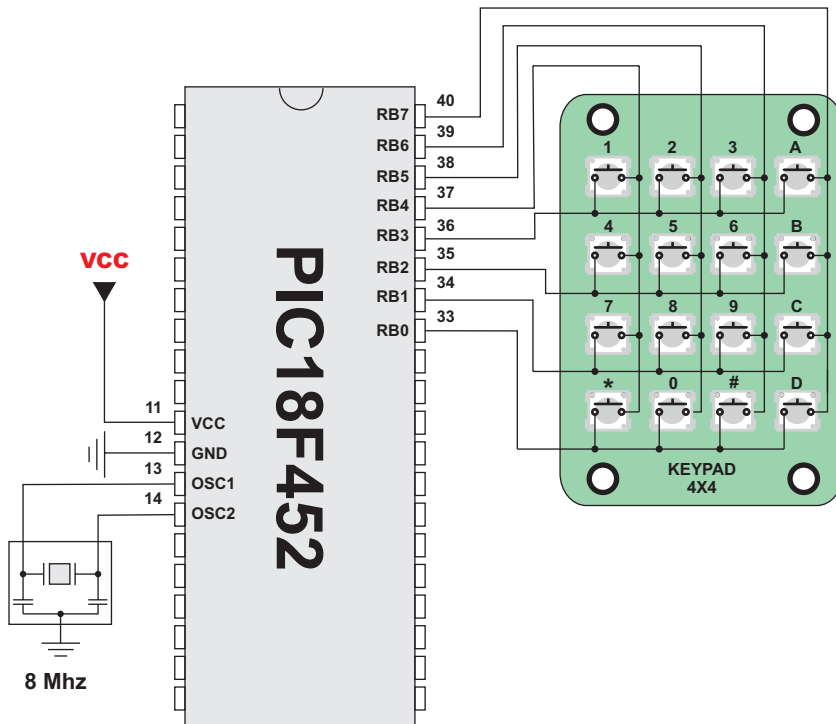
        //--- prepare value for output
        if (kp > 10)
            kp += 54;
        else
            kp += 47;

        //--- print it on LCD
        Lcd_Chr(1, 10, kp);
        WordToStr(cnt, txt);
        Lcd_Out(2, 10, txt);

    } while (1);
} //~!

```

HW Connection



LCD Library (4-bit interface)

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

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

Library Routines

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

Lcd_Config

Prototype	<code>void Lcd_Config(char *port, char RS, char EN, char WR, char D7, char D6, char D5, char D4);</code>
Description	Initializes LCD at port with pin settings you specify: parameters RS, EN, WR, D7 .. D4 need to be a combination of values 0–7 (e.g. 3,6,0,7,2,1,4).
Example	<code>Lcd_Config(&PORTD, 1, 2, 0, 3, 5, 4, 6);</code>

Lcd_Init

Prototype	<code>void Lcd_Init(char *port);</code>
Description	Initializes LCD at <code>port</code> with default pin settings (see the connection scheme at the end of the chapter): D7 -> PORT.7, D6 -> PORT.6, D5 -> PORT.5, D4 -> PORT.4, E -> PORT.3, RS -> PORT.2.
Example	<code>Lcd_Init(&PORTB);</code>

Lcd_Out

Prototype	<code>void Lcd_Out(char row, char col, char *text);</code>
Description	Prints <code>text</code> on LCD at specified row and column (parameter <code>row</code> and <code>col</code>). Both string variables and literals can be passed as <code>text</code> .
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> or <code>Lcd_Init</code> .
Example	<code>Lcd_Out(1, 3, "Hello!"); // Print "Hello!" at line 1, char 3</code>

Lcd_Out_Cp

Prototype	<code>void Lcd_Out_Cp(char *text);</code>
Description	Prints <code>text</code> on LCD at current cursor position. Both string variables and literals can be passed as <code>text</code> .
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> or <code>Lcd_Init</code> .
Example	<code>Lcd_Out_Cp("Here!"); // Print "Here!" at current cursor position</code>

Lcd_Chr

Prototype	<code>void Lcd_Chr(char row, char col, char character);</code>
Description	Prints <code>character</code> on LCD at specified row and column (parameters <code>row</code> and <code>col</code>). Both variables and literals can be passed as <code>character</code> .
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> or <code>Lcd_Init</code> .
Example	<code>Lcd_Out(2, 3, 'i');</code> // Print 'i' at line 2, char 3

Lcd_Chr_Cp

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

Lcd_Cmd

Prototype	<code>void Lcd_Cmd(char command);</code>
Description	Sends <code>command</code> to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown on the following page.
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> or <code>Lcd_Init</code> .
Example	<code>Lcd_Cmd(Lcd_Clear);</code> // Clear LCD display

LCD Commands

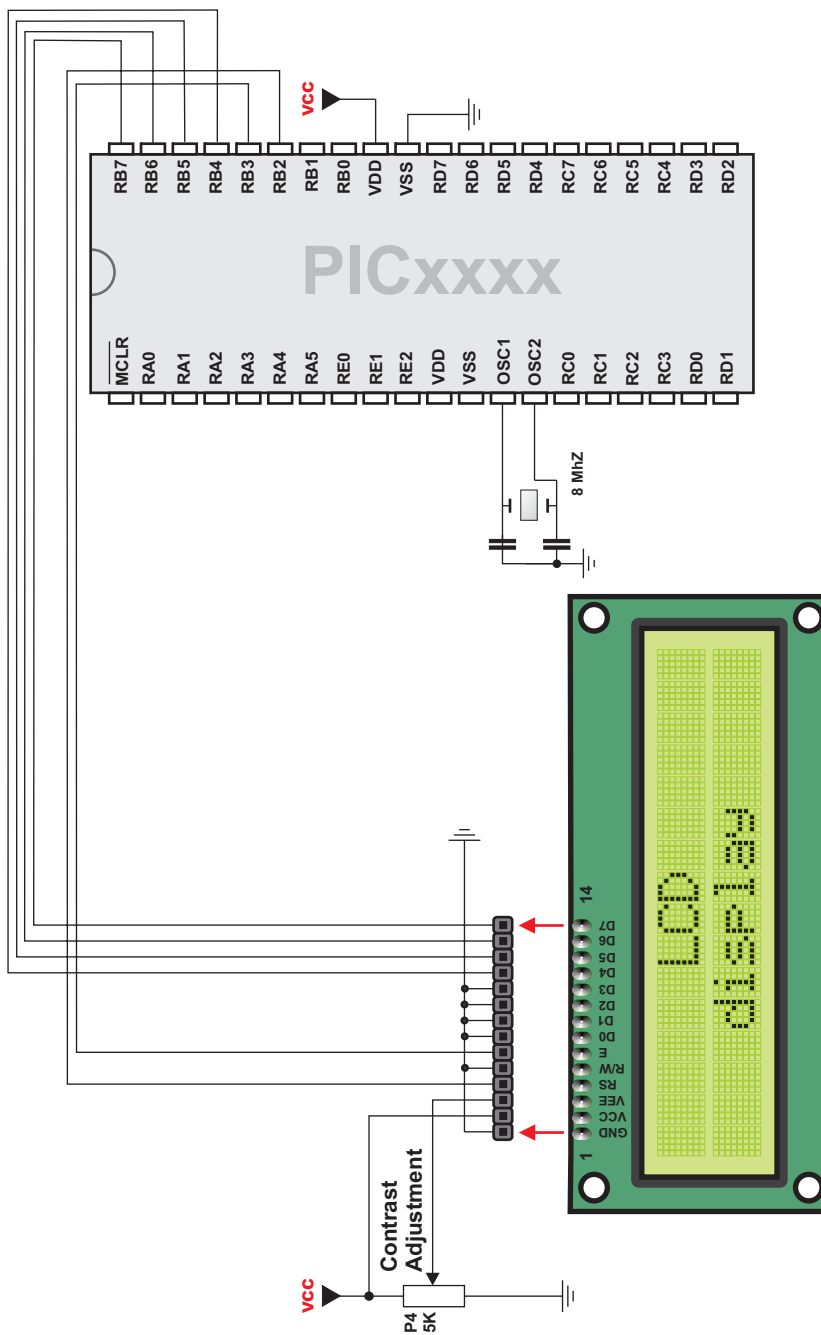
LCD Command	Purpose
LCD_FIRST_ROW	Move cursor to 1st row
LCD_SECOND_ROW	Move cursor to 2nd row
LCD_THIRD_ROW	Move cursor to 3rd row
LCD_FOURTH_ROW	Move cursor to 4th row
LCD_CLEAR	Clear display
LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to original position. Display data RAM is unaffected.
LCD_CURSOR_OFF	Turn off cursor
LCD_UNDERLINE_ON	Underline cursor on
LCD_BLINK_CURSOR_ON	Blink cursor on
LCD_MOVE_CURSOR_LEFT	Move cursor left without changing display data RAM
LCD_MOVE_CURSOR_RIGHT	Move cursor right without changing display data RAM
LCD_TURN_ON	Turn LCD display on
LCD_TURN_OFF	Turn LCD display off
LCD_SHIFT_LEFT	Shift display left without changing display data RAM
LCD_SHIFT_RIGHT	Shift display right without changing display data RAM

Library Example (default pin settings)

```
char *text = "mikroElektronika";

void main() {
    TRISB = 0;                // PORTB is output
    Lcd_Init(&PORTB);         // Initialize LCD connected to PORTB
    Lcd_Cmd(Lcd_CLEAR);      // Clear display
    Lcd_Cmd(Lcd_CURSOR_OFF); // Turn cursor off
    Lcd_Out(1, 1, text);     // Print text to LCD, 2nd row, 1st column
} //~!
```


Hardware Connection



LCD Custom Library (4-bit interface)

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

Library Routines

```
Lcd_Custom_Config
Lcd_Custom_Out
Lcd_Custom_Out_Cp
Lcd_Custom_Chr
Lcd_Custom_Chr_Cp
Lcd_Custom_Cmd
```

Lcd_Custom_Config

Prototype	<code>void Lcd_Custom_Config(char * data_port, char D7, char D6, char D5, char D4, char * ctrl_port, char RS, char WR, char EN);</code>
Description	Initializes LCD data port and control port with pin settings you specify.
Example	<code>Lcd_Custom_Config(&PORTD, 3, 2, 1, 0, &PORTB, 2, 3, 4);</code>

Lcd_Custom_Out

Prototype	<code>void Lcd_Custom_Out(char row, char col, char *text);</code>
Description	Prints <code>text</code> on LCD at specified row and column (parameter <code>row</code> and <code>col</code>). Both string variables and literals can be passed as <code>text</code> .
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> .
Example	<code>Lcd_Custom_Out(1, 3, "Hello!");</code> //Print "Hello!" at line 1, char 3

Lcd_Custom_Out_Cp

Prototype	<code>void Lcd_Custom_Out_Cp(char *text);</code>
Description	Prints <code>text</code> on LCD at current cursor position. Both string variables and literals can be passed as <code>text</code> .
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> .
Example	<code>Lcd_Custom_Out_Cp("Here!");</code> // Print "Here!" at current cursor position

Lcd_Custom_Chr

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

Lcd_Custom_Chr_Cp

Prototype	<code>void Lcd_Custom_Chr_Cp(char character);</code>
Description	Prints <code>character</code> on LCD at current cursor position. Both variables and literals can be passed as <code>character</code> .
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> .
Example	<code>Lcd_Custom_Out_Cp('e');</code> // Print 'e' at current cursor position

Lcd_Custom_Cmd

Prototype	<code>void Lcd_Custom_Cmd(char command);</code>
Description	Sends <code>command</code> to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown on the following page.
Requires	Port with LCD must be initialized. See <code>Lcd_Config</code> .
Example	<code>Lcd_Custom_Cmd(Lcd_Clear);</code> // Clear LCD display

LCD Commands

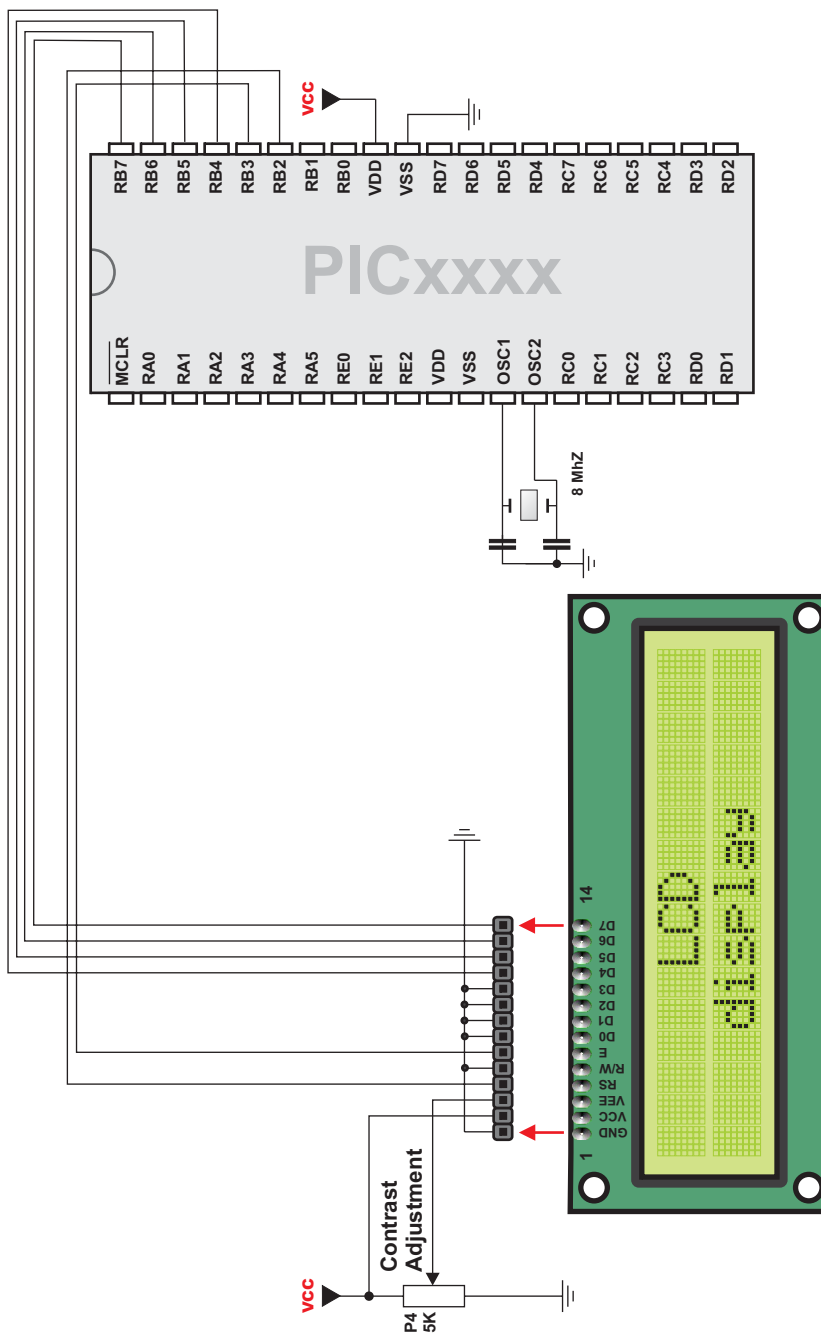
LCD Command	Purpose
LCD_FIRST_ROW	Move cursor to 1st row
LCD_SECOND_ROW	Move cursor to 2nd row
LCD_THIRD_ROW	Move cursor to 3rd row
LCD_FOURTH_ROW	Move cursor to 4th row
LCD_CLEAR	Clear display
LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to original position. Display data RAM is unaffected.
LCD_CURSOR_OFF	Turn off cursor
LCD_UNDERLINE_ON	Underline cursor on
LCD_BLINK_CURSOR_ON	Blink cursor on
LCD_MOVE_CURSOR_LEFT	Move cursor left without changing display data RAM
LCD_MOVE_CURSOR_RIGHT	Move cursor right without changing display data RAM
LCD_TURN_ON	Turn LCD display on
LCD_TURN_OFF	Turn LCD display off
LCD_SHIFT_LEFT	Shift display left without changing display data RAM
LCD_SHIFT_RIGHT	Shift display right without changing display data RAM

Library Example (default pin settings)

```
char *text = "mikroElektronika";

void main() {
    TRISB = 0;                // PORTB is output
    Lcd_Custom_Config(&PORTB,7,6,5,4,&PORTB,3,0,2); // Initialize LCD connected to
PORTB
    Lcd_Custom_Cmd(Lcd_CLEAR); // Clear display
    Lcd_Custom_Cmd(Lcd_CURSOR_OFF); // Turn cursor off
    Lcd_Custom_Out(1, 1, text); // Print text to LCD, 2nd row, 1st col-
umn
} //~!
```

Hardware Connection



LCD8 Library (8-bit interface)

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

Library Routines

```
Lcd8_Config
Lcd8_Init
Lcd8_Out
Lcd8_Out_Cp
Lcd8_Chr
Lcd8_Chr_Cp
Lcd8_Cmd
```

Lcd8_Config

Prototype	<code>void Lcd8_Config(char *ctrlport, char *dataport, char RS, char EN, char WR, char D7, char D6, char D5, char D4, char D3, char D2, char D1, char D0);</code>
Description	Initializes LCD at Control port (ctrlport) and Data port (dataport) with pin settings you specify: Parameters RS, EN, and WR need to be in range 0–7; Parameters D7 .. D0 need to be a combination of values 0–7 (e.g. 3,6,5,0,7,2,1,4).
Example	<code>Lcd8_Config(&PORTC, &PORTD, 0, 1, 2, 6, 5, 4, 3, 7, 1, 2, 0);</code>

Lcd8_Init

Prototype	<code>void Lcd8_Init(char *ctrlport, char *dataport);</code>
Description	<p>Initializes LCD at Control port (<code>ctrlport</code>) and Data port (<code>dataport</code>) with default pin settings (see the connection scheme at the end of the chapter):</p> <p>E -> <code>ctrlport.3</code>, RS -> <code>ctrlport.2</code>, R/W -> <code>ctrlport.0</code>, D7 -> <code>dataport.7</code>, D6 -> <code>dataport.6</code>, D5 -> <code>dataport.5</code>, D4 -> <code>dataport.4</code>, D3 -> <code>dataport.3</code>, D2 -> <code>dataport.2</code>, D1 -> <code>dataport.1</code>, D0 -> <code>dataport.0</code></p>
Example	<code>Lcd8_Init(&PORTB, &PORTC);</code>

Lcd8_Out

Prototype	<code>void Lcd8_Out(char row, char col, char *text);</code>
Description	Prints <code>text</code> on LCD at specified row and column (parameter <code>row</code> and <code>col</code>). Both string variables and literals can be passed as <code>text</code> .
Requires	Ports with LCD must be initialized. See <code>Lcd8_Config</code> or <code>Lcd8_Init</code> .
Example	<code>Lcd8_Out(1, 3, "Hello!"); // Print "Hello!" at line 1, char 3</code>

Lcd8_Out_Cp

Prototype	<code>void Lcd8_Out_Cp(char *text);</code>
Description	Prints <code>text</code> on LCD at current cursor position. Both string variables and literals can be passed as <code>text</code> .
Requires	Ports with LCD must be initialized. See <code>Lcd8_Config</code> or <code>Lcd8_Init</code> .
Example	<code>Lcd8_Out_Cp("Here!"); // Print "Here!" at current cursor position</code>

Lcd8_Chr

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

Lcd8_Chr_Cp

Prototype	<code>void Lcd8_Chr_Cp(char character);</code>
Description	Prints <code>character</code> on LCD at current cursor position. Both variables and literals can be passed as <code>character</code> .
Requires	Ports with LCD must be initialized. See <code>Lcd8_Config</code> or <code>Lcd8_Init</code> .
Example	<code>Lcd8_Out_Cp('e'); // Print 'e' at current cursor position</code>

Lcd8_Cmd

Prototype	<code>void Lcd8_Cmd(char command);</code>
Description	Sends <code>command</code> to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is on the page 186.
Requires	Ports with LCD must be initialized. See <code>Lcd8_Config</code> or <code>Lcd8_Init</code> .
Example	<code>Lcd8_Cmd(Lcd_Clear); // Clear LCD display</code>

Library Example (default pin settings)

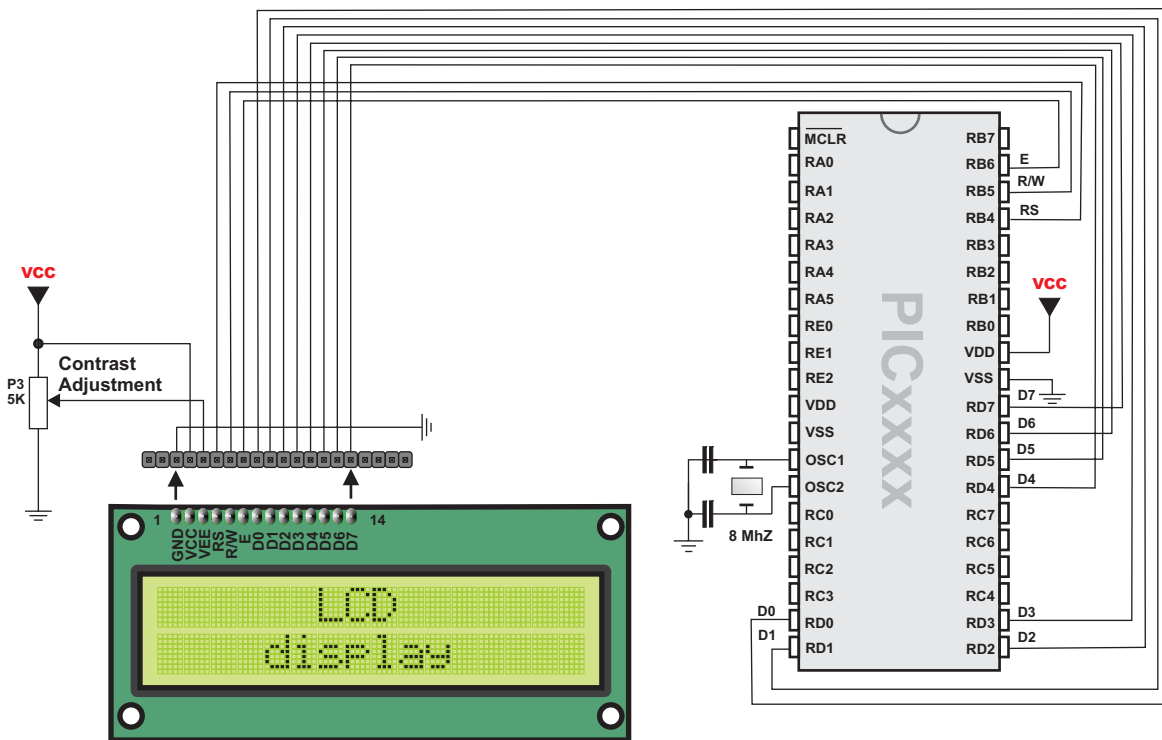
```

char *text = "mikroElektronika";

void main() {
    TRISB = 0;           // PORTB is output
    TRISC = 0;          // PORTC is output
    Lcd8_Init(&PORTB, &PORTC); // Initialize LCD at PORTB and PORTC
    Lcd8_Cmd(Lcd_CURSOR_OFF); // Turn off cursor
    Lcd8_Out(1, 1, text); // Print text on LCD
}

```

Hardware Connection



GLCD Library

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

Library Routines

Basic routines:

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

Advanced routines:

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

Glcd_Init

Prototype	<code>void Glcd_Init(unsigned char *ctrl_port, char cs1, char cs2, char rs, char rw, char rst, char en, unsigned char *data_port);</code>
Description	Initializes GLCD at lower byte of <code>data_port</code> with pin settings you specify. Parameters <code>cs1</code> , <code>cs2</code> , <code>rs</code> , <code>rw</code> , <code>rst</code> , and <code>en</code> can be pins of any available port. This function needs to be called before using other routines of GLCD library.
Example	<code>Glcd_Init(PORTB, PORTC, 3, 5, 7, 1, 2);</code>

Glcd_Set_Side

Prototype	<code>void Glcd_Set_Side(unsigned short x);</code>
Description	Selects side of GLCD, left or right. Parameter <code>x</code> specifies the side: values from 0 to 63 specify the left side, and values higher than 64 specify the right side. Use the functions <code>Glcd_Set_Side</code> , <code>Glcd_Set_X</code> , and <code>Glcd_Set_Page</code> to specify an exact position on GLCD. Then, you can use <code>Glcd_Write_Data</code> or <code>Glcd_Read_Data</code> on that location.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Select_Side(0);</code>

Glcd_Set_Page

Prototype	<code>void Glcd_Set_Page(unsigned short page);</code>
Description	Selects page of GLCD, technically a line on display; parameter <code>page</code> can be 0..7.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Set_Page(5);</code>

Glcd_Set_X

Prototype	<code>void Glcd_Set_X(unsigned short x_pos);</code>
Description	Positions to <code>x</code> dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Set_X(25);</code>

Glcd_Read_Data

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

Glcd_Write_Data

Prototype	<code>void Glcd_Write_Data(unsigned short data);</code>
Description	Writes data to the current location in GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Write_Data(data);</code>

Glcd_Fill

Prototype	<code>void Glcd_Fill(unsigned short pattern);</code>
Description	Fills the GLCD memory with byte pattern. To clear the GLCD screen, use <code>Glcd_Fill(0)</code> ; to fill the screen completely, use <code>Glcd_Fill(\$FF)</code> .
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Fill(0); // Clear screen</code>

Glcd_Dot

Prototype	<code>void Glcd_Dot(unsigned short x, unsigned short y, char color);</code>
Description	Draws a dot on the GLCD at coordinates (x, y). Parameter color determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Dot(0, 0, 2); // Invert the dot in the upper left corner</code>

Glcd_Line

Prototype	<code>void Glcd_Line(int x1, int y1, int x2, int y2, char color);</code>
Description	Draws a line on the GLCD from (x1, y1) to (x2, y2). Parameter <code>color</code> determines the dot state: 0 draws an empty line (clear dots), 1 draws a full line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Line(0, 63, 50, 0, 2);</code>

Glcd_V_Line

Prototype	<code>void Glcd_V_Line(unsigned short y1, unsigned short y2, unsigned short x, char color);</code>
Description	Similar to <code>GLcd_Line</code> , draws a vertical line on the GLCD from (x, y1) to (x, y2).
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_V_Line(0, 63, 0, 1);</code>

Glcd_H_Line

Prototype	<code>void Glcd_H_Line(unsigned short x1, unsigned short x2, unsigned short y, char color);</code>
Description	Similar to <code>GLcd_Line</code> , draws a horizontal line on the GLCD from (x1, y) to (x2, y).
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_H_Line(0, 127, 0, 1);</code>

Glcd_Rectangle

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

Glcd_Box

Prototype	<code>void Glcd_Box(unsigned short x1, unsigned short y1, unsigned short x2, unsigned short y2, char color);</code>
Description	Draws a box on the GLCD. Parameters (x1, y1) set the upper left corner, (x2, y2) set the bottom right corner. Parameter color defines the fill: 0 draws a white box (clear dots), 1 draws a full box (put dots), and 2 draws an inverted box (invert each dot).
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	<code>Glcd_Box(10, 0, 30, 35, 1);</code>

Glcd_Circle

Prototype	<code>void Glcd_Circle(int x, int y, int radius, char color);</code>
Description	Draws a circle on the GLCD, centered at (x, y) with radius. Parameter color defines the circle line: 0 draws an empty line (clear dots), 1 draws a solid line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Circle(63, 31, 25, 2);</code>

Glcd_Set_Font

Prototype	<code>void Glcd_Set_Font(const char *font, unsigned short font_width, unsigned short font_height, unsigned font_offset);</code>
Description	<p>Sets font for routines <code>Glcd_Write_Char</code> and <code>Glcd_Write_Text</code>. Parameter <code>font</code> needs to be formatted in an array of byte.</p> <p>Parameters <code>font_width</code> and <code>font_height</code> specify the width and height of characters in dots. Font width should not exceed 128 dots, and font height shouldn't exceed 8 dots.</p> <p>Parameter <code>font_offset</code> determines the ASCII character from which the supplied font starts. Demo fonts supplied with the library have an offset of 32, which means that they start with space.</p> <p>You can create your own fonts by following the guidelines given in file “<code>GLcd_Fonts.c</code>”. This file contains the default fonts for GLCD, and is located in your installation folder, “Extra Examples” > “GLCD”.</p>
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>// Use the custom 5x8 font "myfont" which starts with space (32): Glcd_Set_Font(myfont_5x8, 5, 8, 32);</code>

Glcd_Write_Char

Prototype	<code>void Glcd_Write_Char(unsigned short character, unsigned short x, unsigned short page, char color);</code>
Description	Prints <code>character</code> at <code>page</code> (one of 8 GLCD lines, 0..7), <code>x</code> dots away from the left border of display. Parameter <code>color</code> defines the “fill”: 0 prints a “white” letter (clear dots), 1 prints a solid letter (put dots), and 2 prints a “smart” letter (invert each dot).
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Write_Char('C', 0, 0, 1);</code>

Glcd_Write_Text

Prototype	<code>void Glcd_Write_Text(char *text, unsigned short x, unsigned short page, unsigned short color);</code>
Description	Prints <code>text</code> at <code>page</code> (one of 8 GLCD lines, 0..7), <code>x</code> dots away from the left border of display. Parameter <code>color</code> defines the “fill”: 0 prints a “white” letters (clear dots), 1 prints solid letters (put dots), and 2 prints “smart” letters (invert each dot).
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Write_Text("Hello world!", 0, 0, 1);</code>

Glcd_Image

Prototype	<code>void Glcd_Image(const char *image);</code>
Description	Displays bitmap image on the GLCD. Parameter <code>image</code> should be formatted as an array of integers. Use the mikroC's integrated Bitmap-to-LCD editor (menu option Tools > BMP2LCD) to convert image to a constant array suitable for display on GLCD.
Requires	GLCD needs to be initialized. See <code>Glcd_Init</code> .
Example	<code>Glcd_Image(my_image);</code>

Library Example

The following drawing demo tests advanced routines of GLCD library.

```
unsigned short j, k;

void main() {

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

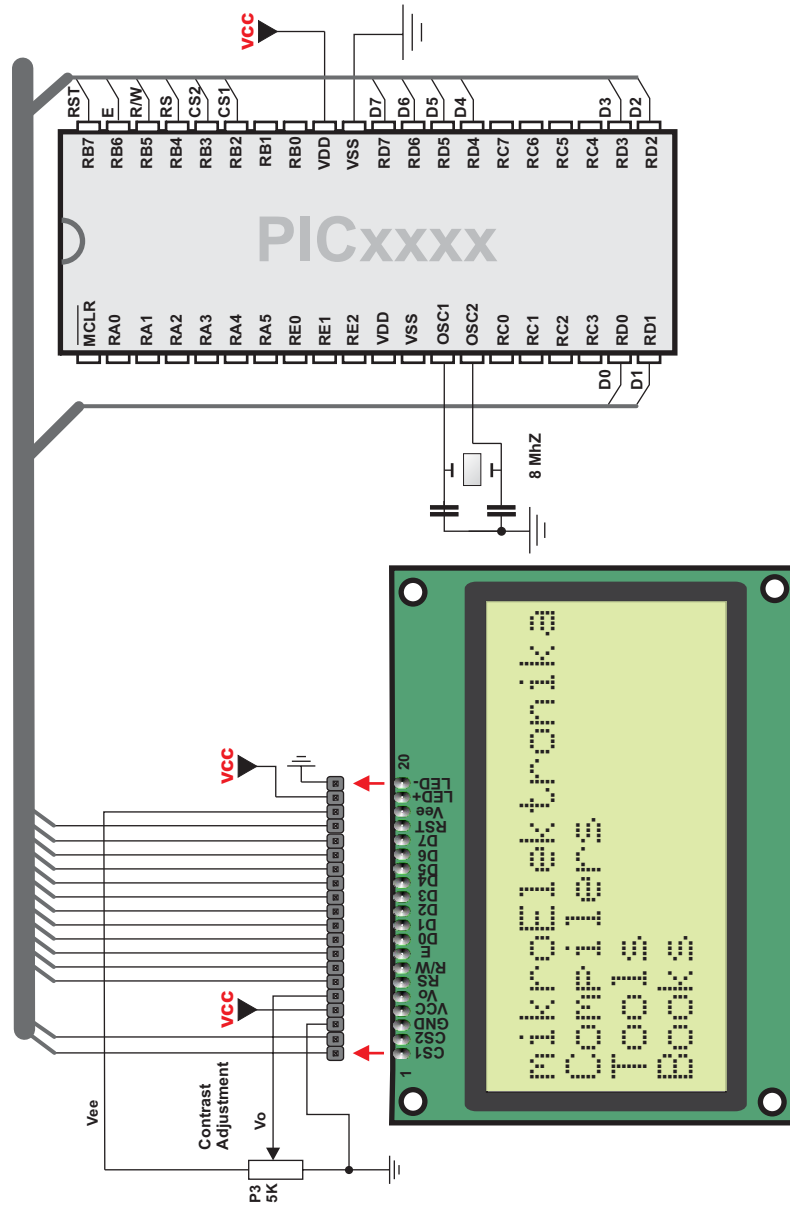
    // Set font for displaying text
    Glcd_Set_Font(@FontSystem5x8, 5, 8, 32);

    do {
        // Draw circles
        Glcd_Fill(0); // Clear screen
        Glcd_Write_Text("Circles", 0, 0, 1);
        j = 4;
        while (j < 31) {
            Glcd_Circle(63, 31, j, 2);
            j += 4;
        }
        Delay_ms(4000);

        // Draw boxes
        Glcd_Fill(0); // Clear screen
        Glcd_Write_Text("Rectangles", 0, 0, 1);
        j = 0;
        while (j < 31) {
            Glcd_Box(j, 0, j + 20, j + 25, 2);
            j += 4;
        }
        Delay_ms(4000);

        // Draw Lines
        Glcd_Fill(0); // Clear screen
        Glcd_Write_Text("Lines", 0, 0, 1);
        for (j = 0; j < 16; j++) {
            k = j*4 + 3;
            Glcd_Line(0, 0, 127, k, 2);
        }
        for (j = 0; j < 31; j++) {
            k = j*4 + 3;
            Glcd_Line(0, 63, k, 0, 2);
        }
        Delay_ms(4000);
    } while (1);
} //~!
```

Hardware Connection



T6963C Graphic LCD Library

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

Library Routines

```
T6963C_Init  
T6963C_writeData  
T6963C_writeCommand  
T6963C_setPtr  
T6963C_waitReady  
T6963C_fill  
T6963C_dot  
T6963C_write_char  
T6963C_write_text  
T6963C_line  
T6963C_rectangle  
T6963C_box  
T6963C_circle  
T6963C_image  
T6963C_sprite  
T6963C_set_cursor  
T6963C_clearBit  
T6963C_setBit  
T6963C_negBit  
T6963C_displayGrPanel  
T6963C_displayTxtPanel  
T6963C_setGrPanel  
T6963C_setTxtPanel  
T6963C_panelFill  
T6963C_grFill  
T6963C_txtFill  
T6963C_cursor_height  
T6963C_graphics  
T6963C_text  
T6963C_cursor  
T6963C_cursor_blink  
T6963C_Init_240x128  
T6963C_Init_240x64
```

T6963C_init

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

T6963C_writeData

Prototype	<code>void T6963C_writeData(unsigned char data);</code>
Description	Routine that writes data to T6963C controller.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_writeData(AddrL);</code>

T6963C_writeCommand

Prototype	<code>void T6963C_writeCommand(unsigned char data);</code>
Description	Routine that writes command to T6963C controller.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_writeCommand(T6963C_CURSOR_POINTER_SET);</code>

T6963C_setPtr

Prototype	<code>void T6963C_setPtr(unsigned int addr, unsigned char t);</code>
Description	This routine sets the memory pointer p for command c.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_writeCommand(T6963C_CURSOR_POINTER_SET);</code>

T6963C_waitReady

Prototype	<code>void T6963C_waitReady(void);</code>
Description	This routine pools the status byte, and loops until ready.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_waitReady();</code>

T6963C_fill

Prototype	<code>void T6963C_fill(unsigned char data, unsigned int start, unsigned int len);</code>
Description	This routine fills length with bytes to controller memory from start address.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_fill(0x33,0x00FF,0x000F);</code>

T6963C_dot

Prototype	<code>void T6963C_dot(int x, int y, unsigned char color);</code>
Description	This sets current text work panel. It writes string str row x line y. mode = T6963C_ROM_MODE_[OR EXOR AND].
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_dot(x0, y0, pcolor);</code>

T6963C_write_char

Prototype	<code>void T6963C_dot(int x, int y, unsigned char color);</code>
Description	This routine sets current text work panel. It writes char c row x line y. mode = T6963C_ROM_MODE_[OR EXOR AND]
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_write_char('A',22,23,AND);</code>

T6963C_write_text

Prototype	<code>void T6963C_write_text(unsigned char *str, unsigned char x, unsigned char y, unsigned char mode);</code>
Description	This sets current text work panel. It writes string str row x line y. mode = T6963C_ROM_MODE_[OR EXOR AND]
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_write_text(" GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR);</code>

T6963C_line

Prototype	<code>void T6963C_line(int px0, int py0, int px1, int py1, unsigned char pcolor);</code>
Description	This routine current graphic work panel. It's draw a line from (x0, y0) to (x1, y1). pcolor = T6963C_[WHITE[BLACK]
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_line(0, 0, 239, 127, T6963C_WHITE);</code>

T6963C_rectangle

Prototype	<code>void T6963C_rectangle(int x0, int y0, int x1, int y1, unsigned char pcolor);</code>
Description	It sets current graphic work panel. It draws the border of the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE);</code>

T6963C_box

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

T6963C_circle

Prototype	<code>void T6963C_circle(int x, int y, long r, unsigned char pcolor);</code>
Description	This routine sets current graphic work panel. It draws a circle, center is (x, y), diameter is r. pcolor = T6963C_[WHITE[BLACK]
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_circle(120, 64, 110, T6963C_WHITE);</code>

T6963C_image

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

T6963C_sprite

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

T6963C_set_cursor

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

T6963C_clearBit

Prototype	<code>void T6963C_clearBit(char b);</code>
Description	Clear control bit.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_clearBit(b);</code>

T6963C_setBit

Prototype	<code>void T6963C_setBit(char b);</code>
Description	Set control bit.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_setBit(b);</code>

T6963C_negBit

Prototype	<code>void T6963C_negBit(char b);</code>
Description	Neg control bit.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_negBit(b);</code>

T6963C_displayGrPanel

Prototype	<code>void T6963C_displayGrPanel(unsigned int n);</code>
Description	Display graphic panel number n.
Requires	GLCD needs to be initialized, see T6963C_init.
Example	<code>T6963C_displayGrPanel(n);</code>

T6963C_displayTxtPanel

Prototype	<code>void T6963C_displayTxtPanel(unsigned int n);</code>
Description	Display text panel number n.
Requires	GLCD needs to be initialized, see T6963C_init.
Example	<code>T6963C_displayTxtPanel(n);</code>

T6963C_setGrPanel

Prototype	<code>void T6963C_setGrPanel(unsigned int n);</code>
Description	Compute graphic start address for panel number n.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_setGrPanel(n);</code>

T6963C_setTxtPanel

Prototype	<code>void T6963C_setTxtPanel(unsigned int n);</code>
Description	Compute text start address for panel number n.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_setTxtPanel(n);</code>

T6963C_panelFill

Prototype	<code>void T6963C_panelFill(unsigned int v);</code>
Description	Fill full #n panel with v bitmap (0 to clear).
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_panelFill(v);</code>

T6963C_grFill

Prototype	<code>void T6963C_grFill(unsigned int v);</code>
Description	Fill graphic #n panel with v bitmap (0 to clear).
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_grFill(v);</code>

T6963C_txtFill

Prototype	<code>void T6963C_txtFill(unsigned int v);</code>
Description	Fill text #n panel with char v + 32 (0 to clear).
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_txtFill(v);</code>

T6963C_cursor_height

Prototype	<code>void T6963C_cursor_height(unsigned int n);</code>
Description	Set cursor size.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_cursor_height(n);</code>

T6963C_graphics

Prototype	<code>void T6963C_graphics(unsigned int n);</code>
Description	Set graphics on/off.
Requires	GLCD needs to be initialized, see T6963C_init.
Example	<code>T6963C_graphics(1);</code>

T6963C_text

Prototype	<code>void T6963C_text(unsigned int n);</code>
Description	Set text on/off.
Requires	GLCD needs to be initialized, see T6963C_init.
Example	<code>T6963C_text(1);</code>

T6963C_cursor

Prototype	<code>void T6963C_cursor(unsigned int n);</code>
Description	Set cursor on/off.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_cursor(1);</code>

T6963C_cursor_blink

Prototype	<code>void T6963C_cursor_blink(unsigned int n);</code>
Description	Set cursor blink on/off.
Requires	Ports must be initialized. See T6963C_init.
Example	<code>T6963C_cursor_blink(0);</code>

T6963C_Init_240x128

Prototype	procedure T6963C_Init_240x128;
Description	Initialize T6963C based GLCD (240x128 pixels) with default settings for mE GLCD's.
Example	T6963C_Init_240x128;

T6963C_Init_240x64

Prototype	procedure T6963C_Init_240x64;
Description	Initialize T6963C based GLCD (240x64 pixels) with default settings for mE GLCD's.
Example	T6963C_Init_240x64;

Library Example

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

```
#include      "T6963C.h"
/*
 * bitmap pictures stored in ROM
 */
extern const char mc[] ;
extern const char einstein[] ;
/*
 * initial PWM duty cycle for contrast power supply
 */
unsigned char   PWM_duty = 200 ;

void main(void)
{
    unsigned char   panel ;           // current panel
    unsigned int    i ;              // general purpose register
    unsigned char   curs ;           // cursor visibility
    unsigned int    cposx, cposy ;   // cursor x-y position

    TRISC = 0 ;                      // port C is output only
    PORTC = 0b00000000 ; // chip enable, reverse on, 8x8 font

    //continues...
```

```
//continued...
```

```

/*
 * init display for 240 pixel width and 128 pixel height
 * 8 bits character width
 * data bus on PORTD
 * control bus on PORTC
 * bit 3 is !WR
 * bit 1 is !RD
 * bit 1 is C!D
 * bit 5 is RST
 */
T6963C_Init_240x128();
//T6963C_init(240, 128, 8, &PORTD, &PORTC, 3, 2, 1, 5) ;
/*
 * 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(" GLCD LIBRARY DEMO, WELCOME !", 0, 0,
T6963C_ROM_MODE_XOR) ;
    T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mC", 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) ;

```

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

```

//continued...
    /*
     * 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) ;
// fill the graphic screen with a picture

    for(;;)
    {

        /*
         * if RB1 is pressed, toggle the display between
        graphic panel 0 and graphic 1
         */
        if(PORTB & 0b00000010)
        {
            panel++ ;
            panel &= 1 ;
            T6963C_displayGrPanel(panel) ;
            Delay_ms(300) ;
        }

    }

//continues...

```

```
//continued...
/*
 * if RB2 is pressed, display only graphic panel
 */
else if(PORTB & 0b00000100)
{
    T6963C_graphics(1) ;
    T6963C_text(0) ;
    Delay_ms(300) ;
}

/*
 * if RB3 is pressed, display only text panel
 */
else if(PORTB & 0b00001000)
{
    T6963C_graphics(0) ;
    T6963C_text(1) ;
    Delay_ms(300) ;
}

/*
 * if RB4 is pressed, display text and graphic
panels
 */
else if(PORTB & 0b00010000)
{
    T6963C_graphics(1) ;
    T6963C_text(1) ;
    Delay_ms(300) ;
}

/*
 * if RB5 is pressed, change cursor
 */

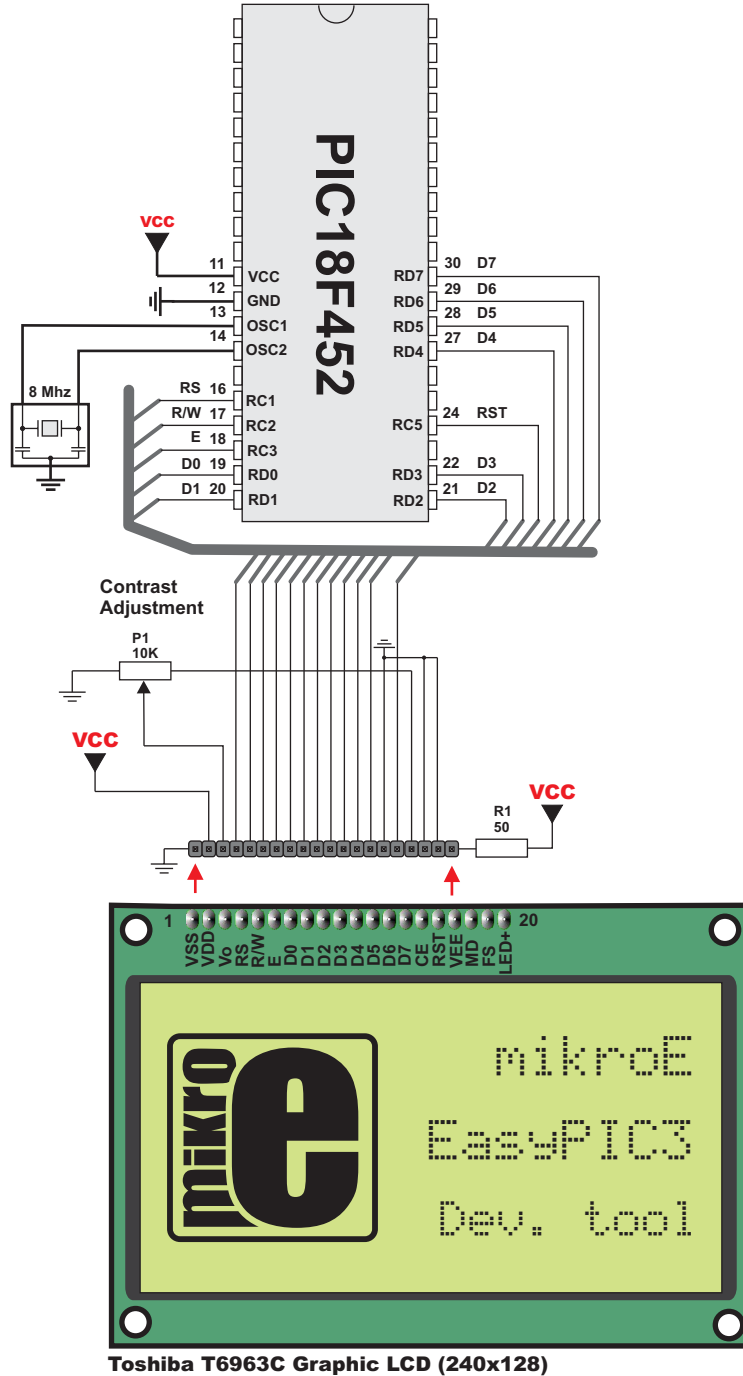
else if(PORTB & 0b00100000)
{
    curs++ ;
    if(curs == 3) curs = 0 ;
    switch(curs)

//continues...
```

```
//continued...
```

```
    /*  
     * 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) ;  
    }  
}
```

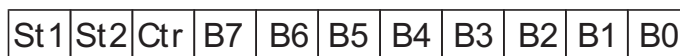
Hardware Connection



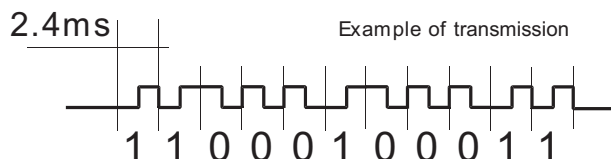
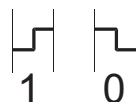
Manchester Code Library

mikroC provides a library for handling Manchester coded signals. Manchester code is a code in which data and clock signals are combined to form a single self-synchronizing data stream; each encoded bit contains a transition at the midpoint of a bit period, the direction of transition determines whether the bit is a 0 or a 1; second half is the true bit value and the first half is the complement of the true bit value (as shown in the figure below).

Manchester RF_Send_Byte format



Bi-phase coding



Notes: Manchester receive routines are blocking calls (`Man_Receive_Config`, `Man_Receive_Init`, `Man_Receive`). This means that PIC will wait until the task is performed (e.g. byte is received, synchronization achieved, etc). Routines for receiving are limited to a baud rate scope from 340 ~ 560 bps.

Library Routines

```

Man_Receive_Config
Man_Receive_Init
Man_Receive
Man_Send_Config
Man_Send_Init
Man_Send

```

Man_Receive_Config

Prototype	<code>void Man_Receive_Config(char *port, char rxpin);</code>
Description	The function prepares PIC for receiving signal. You need to specify the <code>port</code> and <code>rxpin</code> (0–7) of input signal. In case of multiple errors on reception, you should call <code>Man_Receive_Init</code> once again to enable synchronization.
Example	<code>Man_Receive_Config(&PORTD, 6);</code>

Man_Receive_Init

Prototype	<code>void Man_Receive_Init(char *port);</code>
Description	The function prepares PIC for receiving signal. You need to specify the <code>port</code> ; <code>rxpin</code> is pin 6 by default. In case of multiple errors on reception, you should call <code>Man_Receive_Init</code> once again to enable synchronization.
Example	<code>Man_Receive_Init(&PORTD);</code>

Man_Receive

Prototype	<code>void Man_Receive(char *error);</code>
Returns	Returns one byte from signal.
Description	Function extracts one byte from signal. If signal format does not match the expected, <code>error</code> flag will be set to 255.
Requires	To use this function, you must first prepare the PIC for receiving. See <code>Man_Receive_Config</code> or <code>Man_Receive_Init</code> .
Example	<pre>temp = Man_Receive(error); if (error) { ... /* error handling */ }</pre>

Man_Send_Config

Prototype	<code>void Man_Send_Config(char *port, char txpin);</code>
Description	The function prepares PIC for sending signal. You need to specify <code>port</code> and <code>txpin</code> (0–7) for outgoing signal. Baud rate is const 500 bps.
Example	<code>Man_Send_Config(&PORTD, 0);</code>

Man_Send_Init

Prototype	<code>void Man_Receive_Init(char *port);</code>
Description	The function prepares PIC for sending signal. You need to specify <code>port</code> for outgoing signal; <code>txpin</code> is pin 0 by default. Baud rate is const 500 bps.
Example	<code>Man_Send_Init(&PORTD);</code>

Man_Send

Prototype	<code>void Man_Send(unsigned short data);</code>
Description	Sends one byte (<code>data</code>).
Requires	To use this function, you must first prepare the PIC for sending. See <code>Man_Send_Config</code> or <code>Man_Send_Init</code> .
Example	<code>unsigned short msg;</code> <code>...</code> <code>Man_Send(msg);</code>

Library Example

```

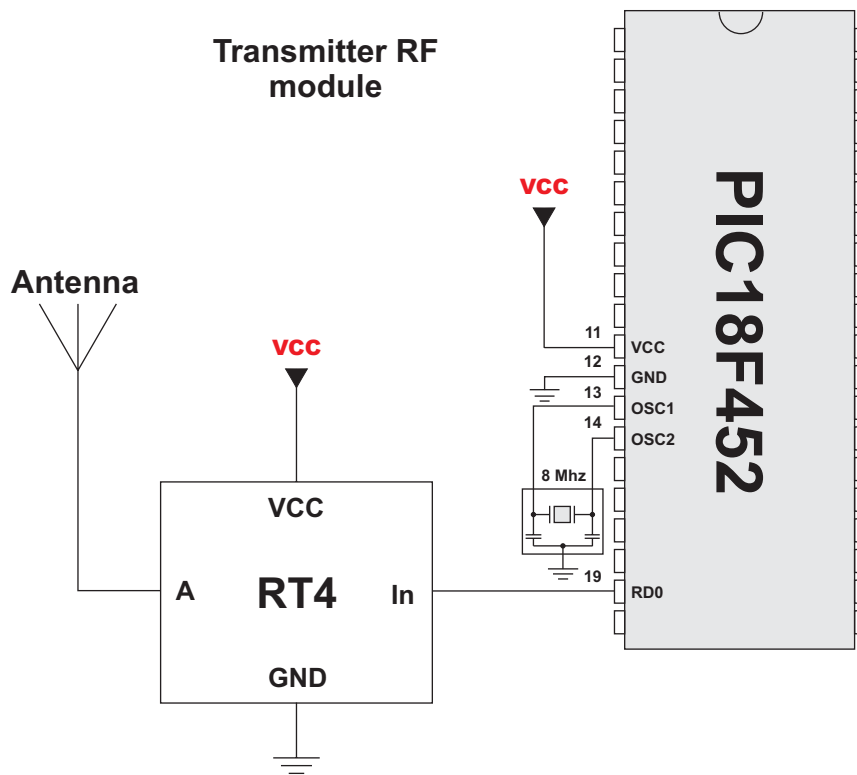
unsigned short error, ErrorCount, IdleCount, temp, LetterCount;

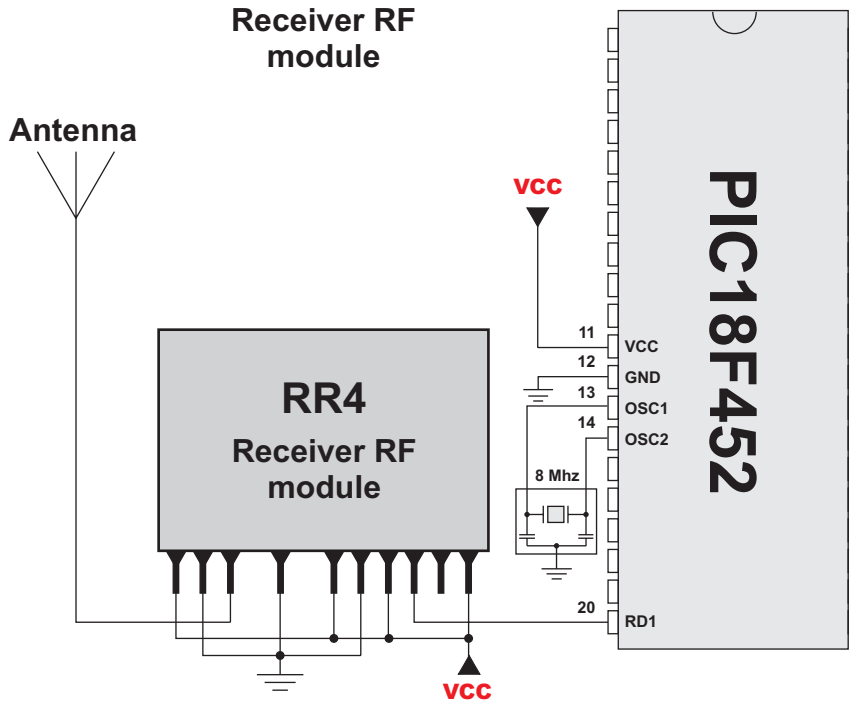
void main() {
    ErrorCount = 0;
    TRISC      = 0;           // Error indicator
    PORTC      = 0;
    Man_Receive_Config(&PORTD, 6); // Synchronize receiver
    Lcd_Init(&PORTB);        // Initialize LCD on PORTB

    while (1) {              // Endless loop
        IdleCount = 0;       // Reset idle counter
        do {
            temp = Man_Receive(error); // Attempt byte receive
            if (error)
                ErrorCount++;
            else
                PORTC = 0;
            if (ErrorCount > 20) { // If there are too many errors
                ErrorCount = 0; // synchronize the receiver again
                PORTC = 0xAA; // Indicate error
                Man_Receive_Init(&PORTD); // Synchronize receiver
            }
            IdleCount++;
            if (IdleCount > 18) { // If nothing received after some time
                IdleCount = 0; // try to synchronize again
                Man_Receive_Init(&PORTD); // Synchronize receiver
            }
        } while (temp != 0x0B); // End of message marker
        if (error != 255) { // If no error then write the message
            Lcd_Cmd(LCD_CLEAR);
            LetterCount = 0;
            while (LetterCount < 17) { // Message is 16 chars long
                LetterCount++;
                temp = Man_Receive(error);
                if (error != 255)
                    Lcd_Chr_Cp(temp)
                else {
                    ErrorCount++; break;
                }
            }
            temp = Man_Receive(error);
            if (temp != 0x0E)
                ErrorCount++;
        } // end if
    } // end while
} //~!

```

Hardware Connection





Multi Media Card Library

mikroC provides a library for accessing data on Multi Media Card via SPI communication. This library supports Secure Digital (SD) flash memory card standard also.

Notes:

- Library works with PIC18 family only;
- Library functions create and read files from the root directory only;
- Library functions populate both FAT1 and FAT2 tables when writing to files, but the file data is being read from the FAT1 table only; i.e. there is no recovery if FAT1 table is corrupted.
- Since version 5.0.0.3, library can cope with media that have the Master Boot Record (MBR) in sector 0. It reads the necessary information from it, and jumps to the first available primary logical 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: `Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH);` must be called before initializing `Mmc_Init`.

Library Routines

```
Mmc_Init  
Mmc_Read_Sector  
Mmc_Write_Sector  
Mmc_Read_Cid  
Mmc_Read_Csd  
  
Mmc_Fat_Init  
Mmc_Fat_Assign  
Mmc_Fat_Reset  
Mmc_Fat_Rewrite  
Mmc_Fat_Append  
Mmc_Fat_Read  
Mmc_Fat_Write  
Mmc_Set_File_Date  
Mmc_Fat_Delete  
Mmc_Fat_Get_File_Date  
Mmc_Fat_Get_File_Size  
Mmc_Fat_Get_Swap_File
```

Mmc_Init

Prototype	<code>unsigned short Mmc_Init(char *port, char pin);</code>
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Initializes MMC with chip select pin being given by the parameters port and pin; communication port and pins are designated by the hardware SPI settings for the respective MCU. Function returns 0 if MMC card is present and successfully initialized, otherwise returns 1. Mmc_Init needs to be called before using other functions of this library.
Requires	<code>Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH);</code> must be called before calling Mmc_Init.
Example	<code>Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH); while (Mmc_Init(&PORTC,2)) ; // Loop until MMC is initialized</code>

Mmc_Read_Sector

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

Mmc_Write_Sector

Prototype	<code>unsigned short Mmc_Write_Sector(unsigned long sector, char *data);</code>
Returns	Returns 0 if write was successful; returns 1 if there was an error in sending write command; returns 2 if there was an error in writing.
Description	Function writes 512 bytes of data to 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	<code>error = Mmc_Write_Sector(sector, data);</code>

Mmc_Read_Cid

Prototype	<code>unsigned short Mmc_Read_Cid(unsigned short *data_for_registers);</code>
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 <code>data_for_registers</code> .
Requires	Library needs to be initialized, see <code>Mmc_Init</code> .
Example	<code>error = Mmc_Read_Cid(data);</code>

Mmc_Read_Csd

Prototype	<code>unsigned short Mmc_Read_Csd(unsigned short *data_for_registers);</code>
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 <code>data_for_registers</code> .
Requires	Library needs to be initialized, see <code>Mmc_Init</code> .
Example	<code>error = Mmc_Read_Csd(data);</code>

Mmc_Fat_Init

Prototype	<code>unsigned short Mmc_Fat_Init(unsigned short *port, unsigned short pin);</code>
Returns	Returns 0 if initialization is successful, 1 if boot sector was not found and 255 if card was not detected.
Description	<p>Initializes MMC/SD cards for FAT routines; CS line for communication is given through the port and pin parameters.</p> <p>This function needs to be called before using other functions of MMC FAT library.</p>
Requires	<code>Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH);</code> must be called before calling <code>Mmc_Fat_Init</code> .
Example	<pre>Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH); // Loop until MMC FAT is initialized at RC2 while (Mmc_Fat_Init(&PORTC, 2)) ;</pre>

Mmc_Fat_Assign

Prototype	<code>void Mmc_Fat_Assign(char *filename);</code>
Description	This routine designates ("assigns") the file we'll be working with. Function looks for the file specified by the <code>filename</code> in the root directory. If the file is found, routine will initialize it by getting its start sector, size, etc. If the file is not found, an empty file will be created with the given name. The <code>filename</code> must be 8 + 3 characters in uppercase.
Requires	Library needs to be initialized; see <code>Mmc_Fat_Init</code> .
Example	<pre>// Assign the file "EXAMPLE1.TXT" in the root directory of MMC. // If the file is not found, routine will create one. Mmc_Fat_Assign("EXAMPLE1TXT");</pre>

Mmc_Fat_Reset

Prototype	<code>void Mmc_Fat_Reset(unsigned long *size);</code>
Description	Function resets the file pointer (moves it to the start of the file) of the assigned file, so that the file can be read. Parameter <code>size</code> stores the size of the assigned file, in bytes.
Requires	Library needs to be initialized; see <code>Mmc_Fat_Init</code> .
Example	<code>Mmc_Fat_Reset(&filesize);</code>

Mmc_Fat_Rewrite

Prototype	<code>void Mmc_Fat_Rewrite(void);</code>
Description	Function resets the file pointer and clears the assigned file, so that new data can be written into the file.
Requires	Library needs to be initialized; see <code>Mmc_Fat_Init</code> .
Example	<code>Mmc_Fat_Rewrite();</code>

Mmc_Fat_Append

Prototype	<code>void Mmc_Fat_Append(void);</code>
Description	The function moves the file pointer to the end of the assigned file, so that data can be appended to the file.
Requires	Library needs to be initialized; see <code>Mmc_Fat_Init</code> .
Example	<code>Mmc_Fat_Append();</code>

Mmc_Fat_Read

Prototype	<code>void Mmc_Fat_Read(unsigned short *data);</code>
Description	Function reads the byte at which the file pointer points to and stores data into parameter <code>data</code> . The file pointer automatically increments with each call of <code>Mmc_Fat_Read</code> .
Requires	File pointer must be initialized; see <code>Mmc_Fat_Reset</code> .
Example	<code>Mmc_Fat_Read(&mydata);</code>

Mmc_Fat_Write

Prototype	<code>void Mmc_Fat_Write(char *fdata, unsigned data_len);</code>
Description	Function writes a chunk of <code>data_len</code> bytes (<code>fdata</code>) to the currently assigned file, at the position of the file pointer.
Requires	File pointer must be initialized; see <code>Mmc_Fat_Append</code> or <code>Mmc_Fat_Rewrite</code> .
Example	<code>Mmc_Fat_Write(txt, 21);</code> <code>Mmc_Fat_Write("Hello\nworld", 1);</code>

Mmc_Set_File_Date

Prototype	<code>void Mmc_Set_File_Date(unsigned year, char month, char day, char hours, char min, char sec);</code>
Description	Writes system timestamp to a file. Use this routine before each writing to the file; otherwise, file will be appended a random timestamp.
Requires	File pointer must be initialized; see <code>Mmc_Fat_Append</code> or <code>Mmc_Fat_Rewrite</code> .
Example	<code>// April 1st 2005, 18:07:00</code> <code>Mmc_Set_File_Date(2005, 4, 1, 18, 7, 0);</code>

Mmc_Fat_Delete

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

Mmc_Fat_Get_File_Date

Prototype	<code>void Mmc_fat_Get_File_Date(unsigned int *year, unsigned short *month, unsigned short *day, unsigned short *hours, unsigned short *mins);</code>
Description	Reads time attributes of file. You can read file year, month, day, hours, mins, seconds.
Requires	Ports must be initialized for FAT operations with MMC. See Mmc_Fat_Init. File must be assigned. See Mmc_Fat_Assign.
Example	<code>Mmc_Fat_Get_File_Date(year, month, day, hours, mins);</code>

Mmc_Fat_Get_File_Size

Prototype	<code>unsigned long Mmc_fat_Get_File_Size();</code>
Description	This function returns size of file in bytes.
Requires	Ports must be initialized for FAT operations with MMC. See Mmc_Fat_Init. File must be assigned. See Mmc_Fat_Assign.
Example	<code>Mmc_Fat_Get_File_Size;</code>

Mmc_Fat_Get_Swap_File

Prototype	<code>unsigned long Mmc_Fat_Get_Swap_File(unsigned long sectors_cnt);</code>
Returns	No. of start sector for the newly created swap file, if swap file was created; otherwise, the function returns zero.
Description	<p>This function is used to create a swap file on the MMC/SD media. It accepts as <code>sectors_cnt</code> argument the number of consecutive sectors that user wants the swap file to have. During its execution, the function searches for the available consecutive sectors, their number being specified by the <code>sectors_cnt</code> argument. If there is such space on the media, the swap file named MIKROSWP.SYS is created, and that space is designated (in FAT tables) to it. The attributes of this file are: system, archive and hidden, in order to distinct it from other files. If a file named MIKROSWP.SYS already exists on the media, this function deletes it upon creating the new one.</p> <p>The purpose of the swap file is to make reading and writing to MMC/SD media as fast as possible, by using the <code>Mmc_Read_Sector()</code> and <code>Mmc_Write_Sector()</code> functions directly, without potentially damaging the FAT system. Swap file can be considered as a "window" on the media where user can freely write/read the data, in any way (s)he wants to. Its main purpose in 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.</p>
Requires	Ports must be initialized for FAT operations with MMC. See <code>Mmc_Fat_Init</code> .
Example	<pre>//Tries to create a swap file, whose size will be at least 1000 //sectors. //If it succeeds, it sends the No. of start sector over USART void M_Create_Swap_File() { size = Mmc_Fat_Get_Swap_File(1000); if (size) { Usart_Write(0xAA); Usart_Write(Lo(size)); Usart_Write(Hi(size)); Usart_Write(Higher(size)); Usart_Write(Highest(size)); Usart_Write(0xAA); } } //~</pre>

Library Example

The following code tests MMC library routines. First, we fill the buffer with 512 “M” characters and write it to sector 56; then we repeat the sequence with character “E” at sector 56. Finally, we read the sectors 55 and 56 to check if the write was successful.

```

unsigned i;
unsigned short tmp;
unsigned short data[ 512] ;

void main() {

    Usart_Init(9600);

    Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW,LOW_2_HIGH);
    // Initialize SPI

    // Wait until MMC is initialized
    while (Mmc_Init(&PORTC, 2)) ;

    // Fill the buffer with the 'M' character
    for (i = 0; i <= 511; i++) data[i] = "M";

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

    // Fill the buffer with the 'E' character
    for (i = 0; i <= 511; i++) data[i] = "E";

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

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

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

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

    // Send 512 bytes from buffer to USART

    if (tmp == 0)
        for (i = 0; i < 512; i++) Usart_Write(data[i]);

} //~!

```

Library Example

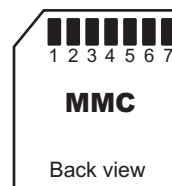
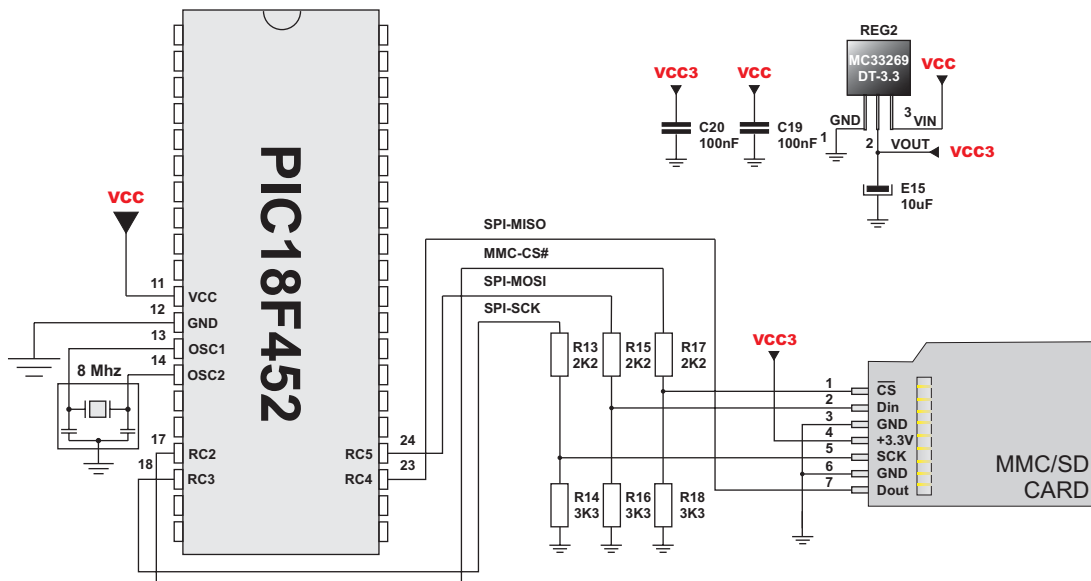
The following program tests MMC FAT routines. It creates 5 different files in the root of MMC card, and fills them with some data. You can check the file dates which should be different.

```

char FAT_ERROR[ 20] = "FAT16 not found";
char file_contents[ 50] = "XX MMC/SD FAT16 library by Anton Rieckert";
char filename[ 14] = "MIKRO00xTXT"; // File names
unsigned short tmp, character, loop;
long i, size;
void main() {
    PORTB = 0;
    TRISB = 0;
    Usart_Init(19200); // Set up USART for reading the files
    Spi_Init_Advanced(MASTER_OSC_DIV16, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH);
    // Initialize SPI
    if (!Mmc_Fat_Init(&PORTC, 2)) { // Try to find the FAT
        tmp = 0;
        while (FAT_ERROR[ tmp] )
            Usart_Write(FAT_ERROR[ tmp++] );
    }
    for (loop = 1; loop <= 5; loop++) { // We want 5 files on our MMC card
        filename[ 7] = loop + 64; // Set number 1, 2, 3, 4 or 5
        Mmc_Fat_Assign(&filename, 1); // If file not found, create new file
        Mmc_Fat_Rewrite(); // Clear the file, start with new data
        file_contents[ 0] = loop / 10 + 48;
        file_contents[ 1] = loop % 10 + 48;
        Mmc_Fat_Write(file_contents, 41); // Write data to the assigned file
        Mmc_Fat_Append(); // Add more data to file
        Mmc_Fat_Write(file_contents, 41); // Write data to file
        Delay_ms(200);
    }
    // Now if we want to add more data to those same files
    for (loop = 1; loop <= 5; loop++) {
        filename[ 7] = loop + 64;
        Mmc_Fat_Assign(&filename, 1); // Assign a file
        Mmc_Fat_Append();
        Mmc_Fat_Set_File_Date(2005, 6, 21, 10, loop, 0);
        Mmc_Fat_Write(" for mikroElektronika 2005\r\n", 30);
        Mmc_Fat_Append();
        Mmc_Fat_Write(file_contents, 41);
        Mmc_Fat_Reset(&size); // To read file, returns file size
        for (i = 1; i <= size; i++) { // Write whole file to USART
            Mmc_Fat_Read(&character);
            Usart_Write(character);
        }
        Delay_ms(200);
    }
} //~!

```

Hardware Connection



OneWire Library

OneWire library provides routines for communication via OneWire bus, for example with DS1820 digital thermometer. This is a Master/Slave protocol, and all the cabling required is a single wire. Because of the hardware configuration it uses (single pullup and open collector drivers), it allows for the slaves even to get their power supply from that line.

Some basic characteristics of this protocol are:

- single master system,
- low cost,
- low transfer rates (up to 16 kbps),
- fairly long distances (up to 300 meters),
- small data transfer packages.

Each OneWire device also has a unique 64-bit registration number (8-bit device type, 48-bit serial number and 8-bit CRC), so multiple slaves can co-exist on the same bus.

Note that oscillator frequency F_{osc} needs to be at least 4MHz in order to use the routines with Dallas digital thermometers.

Library Routines

```
Ow_Reset  
Ow_Read  
Ow_Write
```


Ow_Reset

Prototype	<code>char Ow_Reset(char *port, char pin);</code>
Returns	Returns 0 if DS1820 is present, 1 if not present.
Description	Issues OneWire reset signal for DS1820. Parameters <code>port</code> and <code>pin</code> specify the location of DS1820.
Requires	Works with Dallas DS1820 temperature sensor only.
Example	<code>Ow_Reset(&PORTA, 5); // reset DS1820 connected to the RA5 pin</code>

Ow_Read

Prototype	<code>char Ow_Read(char *port, char pin);</code>
Returns	Data read from an external device over the OneWire bus.
Description	Reads one byte of data via the OneWire bus.
Example	<code>tmp = Ow_Read(&PORTA, 5);</code>

Ow_Write

Prototype	<code>void Ow_Write(char *port, char pin, char par);</code>
Description	Writes one byte of data (argument <code>par</code>) via OneWire bus.
Example	<code>Ow_Write(&PORTA, 5, 0xCC);</code>

Library Example

```

unsigned temp;
unsigned short j;

void Display_Temperature(unsigned int temp) {
    //...
}

void main() {
    ADCON1 = 0xFF;           // Configure RA5 pin as digital I/O
    PORTA  = 0xFF;
    TRISA  = 0x0F;           // PORTA is input
    PORTB  = 0;
    TRISB  = 0;             // PORTB is output

    // Initialize LCD on PORTB and prepare for output

    do {

        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
        Delay_ms(400);

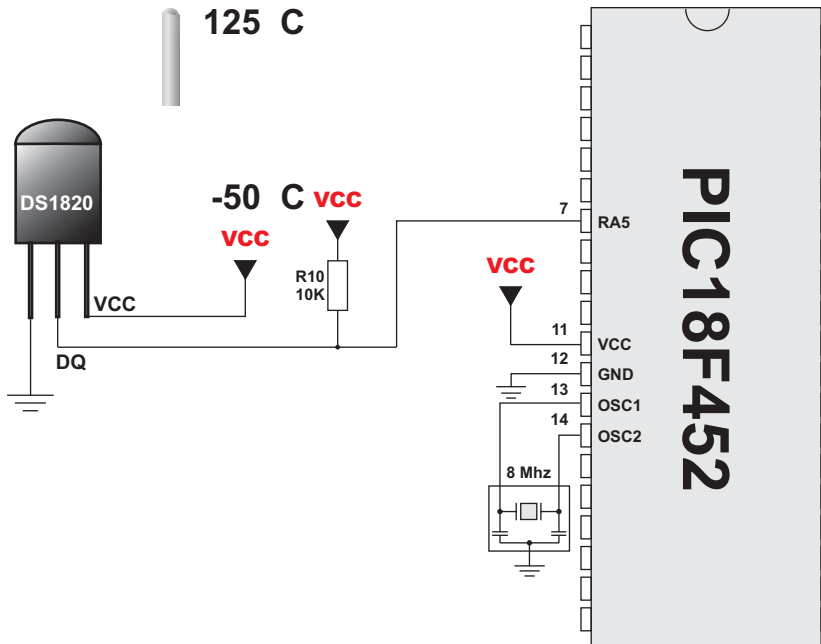
        j = OW_Read(&PORTA,5);        // Get temperature LSB
        temp = OW_Read(&PORTA,5);     // Get temperature MSB
        temp <<= 8; temp += j;        // Form the result
        Display_Temperature(temp);    // Format and display result on LCD
        Delay_ms(500);

    } while (1);

} //~!

```

Hardware Connection



PS/2 Library

mikroC provides a library for communicating with common PS/2 keyboard. The library does not utilize interrupts for data retrieval, and requires oscillator clock to be 6MHz and above.

Library Routines

Ps2_Init
Ps2_Config
Ps2_Key_Read

Ps2_Init

Prototype	<code>void Ps2_Init(unsigned short *port);</code>
Description	<p>Initializes <code>port</code> for work with PS/2 keyboard, with default pin settings. Port pin 0 is Data line, and port pin 1 is Clock line.</p> <p>You need to call either <code>Ps2_Init</code> or <code>Ps2_Config</code> before using other routines of PS/2 library.</p>
Requires	Both Data and Clock lines need to be in pull-up mode.

Ps2_Config

Prototype	<code>void Ps2_Config(char *port, char clock, char data);</code>
Description	<p>Initializes <code>port</code> for work with PS/2 keyboard, with custom pin settings. Parameters <code>data</code> and <code>clock</code> specify pins of <code>port</code> for Data line and Clock line, respectively. <code>Data</code> and <code>clock</code> need to be in range 0..7 and cannot point to the same pin.</p> <p>You need to call either <code>Ps2_Init</code> or <code>Ps2_Config</code> before using other routines of PS/2 library.</p>
Requires	Both Data and Clock lines need to be in pull-up mode.
Example	<code>Ps2_Config(&PORTB, 2, 3);</code>

Ps2_Key_Read

Prototype	<code>char Ps2_Key_Read(char *value, char *special, char *pressed);</code>
Returns	Returns 1 if reading of a key from the keyboard was successful, otherwise 0.
Description	<p>The function retrieves information about key pressed.</p> <p>Parameter <code>value</code> holds the value of the key pressed. For characters, numerals, punctuation marks, and space, <code>value</code> will store the appropriate ASCII value. Routine “recognizes” the function of Shift and Caps Lock, and behaves appropriately.</p> <p>Parameter <code>special</code> is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of these, <code>special</code> will be set to 1, otherwise 0.</p> <p>Parameter <code>pressed</code> is set to 1 if the key is pressed, and 0 if released.</p>
Requires	PS/2 keyboard needs to be initialized; see <code>Ps2_Init</code> or <code>Ps2_Config</code> .
Example	<pre>// Press Enter to continue: do { if (Ps2_Key_Read(&value, &special, &pressed)) { if ((value == 13) && (special == 1)) break; } } while (1);</pre>

Library Example

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

```
unsigned short keydata, special, down;

void main() {
    CMCON = 0x07;    // Disable analog comparators (comment this for PIC18)
    INTCON = 0;      // Disable all interrupts
    Ps2_Init(&PORTA); // Init PS/2 Keyboard on PORTA
    Delay_ms(100);   // Wait for keyboard to finish

    do {
        if (Ps2_Key_Read(&keydata, &special, &down)) {
            if (down && (keydata == 16)) { // Backspace
                // ...do something with a backspace...
            }
            else if (down && (keydata == 13)) { // Enter
                Usart_Write(13);
            }
            else if (down && !special && keydata) {
                Usart_Write(keydata);
            }
        }
        Delay_ms(10); // debounce
    } while (1);
} //~!
```

PWM Library

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

Note: Certain PICmicros with two or more CCP modules, such as P18F8520, require you to specify the module you want to use. Simply append the number 1 or 2 to a Pwm. For example, Pwm2_Start(); Also, for the sake of backward compatibility with previous compiler versions and easier code management, MCU's with multiple PWM modules have PWM library which is identical to PWM1 (i.e. you can use Pwm_Init() instead of PWM1_Init() to initialize CCP1).

Library Routines

```
Pwm_Init
Pwm_Change_Duty
Pwm_Start
Pwm_Stop
```

Pwm_Init

Prototype	<code>void Pwm_Init(long freq);</code>
Description	Initializes the PWM module with duty ratio 0. Parameter <code>freq</code> is a desired PWM frequency in Hz (refer to device data sheet for correct values in respect with <code>Fosc</code>). <code>Pwm_Init</code> needs to be called before using other functions from PWM Library.
Requires	You need a CCP module in order to use this library. Check mikroC installation folder, subfolder "Examples", for alternate solutions.
Example	<code>Pwm_Init(5000); // Initialize PWM module at 5KHz</code>

Pwm_Change_Duty

Prototype	<code>void Pwm_Change_Duty(char duty_ratio);</code>
Description	Changes PWM duty ratio. Parameter <code>duty_ratio</code> takes values from 0 to 255, where 0 is 0%, 127 is 50%, and 255 is 100% duty ratio. Other specific values for duty ratio can be calculated as $(\text{Percent} * 255) / 100$.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initialized – see <code>Pwm_Init</code> .
Example	<code>Pwm_Change_Duty(192); // Set duty ratio to 75%</code>

Pwm_Start

Prototype	<code>void Pwm_Start(void);</code>
Description	Starts PWM.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initialized – see <code>Pwm_Init</code> .
Example	<code>Pwm_Start();</code>

Pwm_Stop

Prototype	<code>void Pwm_Stop(void);</code>
Description	Stops PWM.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initialized – see <code>Pwm_Init</code> .
Example	<code>Pwm_Stop();</code>

Library Example

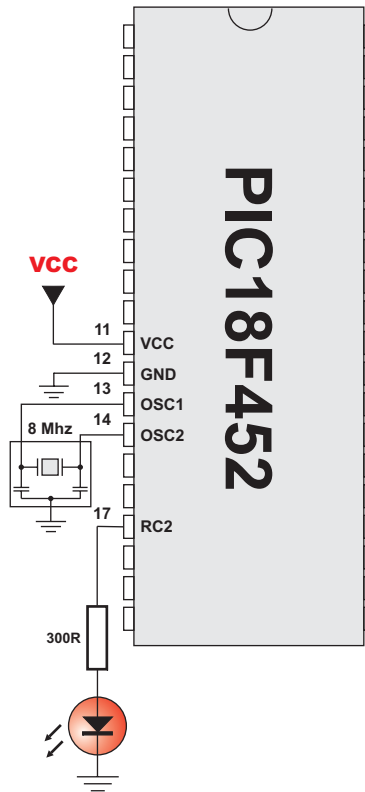
*/*The example changes PWM duty ratio on pin RC2 continually. If LED is connected to RC2, you can observe the gradual change of emitted light. */*

```
char i = 0, j = 0;

void main() {
    PORTC = 0xFF;           // PORTC is output
    Pwm_Init(5000);        // Initialize PWM module at 5KHz
    Pwm_Start();           // Start PWM

    while (1) {
        // Slow down, allow us to see the change on LED:
        for (i = 0; i < 20; i++) Delay_us(500);
        j++;
        Pwm_Change_Duty(j); // Change duty ratio
    }
}
```

Hardware Connection



RS-485 Library

RS-485 is a multipoint communication which allows multiple devices to be connected to a single signal cable. mikroC provides a set of library routines to provide you comfortable work with RS-485 system using Master/Slave architecture. Master and Slave devices interchange packets of information, each of these packets containing synchronization bytes, CRC byte, address byte, and the data. Each Slave has its unique address and receives only the packets addressed to it. Slave can never initiate communication. It is programmer's responsibility to ensure that only one device transmits via 485 bus at a time.

RS-485 routines require USART module on PORTC. Pins of USART need to be attached to RS-485 interface transceiver, such as LTC485 or similar. Pins of transceiver (Receiver Output Enable and Driver Outputs Enable) should be connected to PORTC, pin 2 (check the figure at end of the chapter).

Note: Address 50 is the common address for all Slaves (packets containing address 50 will be received by all Slaves). The only exceptions are Slaves with addresses 150 and 169, which require their particular address to be specified in the packet.

Note: `Usart_Init()` must be called before initializing RS485.

Library Routines

```
RS485Master_Init  
RS485Master_Receive  
RS485Master_Send  
RS485Slave_Init  
RS485Slave_Receive  
RS485Slave_Send
```

RS485Master_Init

Prototype	<code>void Rs485master_Init(unsigned short * port, unsigned short pin);</code>
Description	Initializes PIC MCU as Master in RS-485 communication.
Requires	USART HW module needs to be initialized. See USART_Init.
Example	<code>RS485Master_Init(PORTC, 2);</code>

RS485Master_Receive

Prototype	<code>void RS485Master_Receive(char *data);</code>
Description	<p>Receives any message sent by Slaves. Messages are multi-byte, so this function must be called for each byte received (see the example at the end of the chapter). Upon receiving a message, buffer is filled with the following values:</p> <p>data[0..2] is the message, data[3] is number of message bytes received, 1-3, data[4] is set to 255 when message is received, data[5] is set to 255 if error has occurred, data[6] is the address of the Slave which sent the message.</p> <p>Function automatically adjusts data[4] and data[5] upon every received message. These flags need to be cleared from the program.</p>
Requires	MCU must be initialized as Master in RS-485 communication in order to be assigned an address. See RS485Master_Init.
Example	<pre> unsigned short msg[8] ; ... RS485Master_Receive(msg); </pre>

RS485Master_Send

Prototype	<code>void RS485Master_Send(char *data, char datalen, char address);</code>
Description	Sends data from buffer to Slave(s) specified by address via RS-485; datalen is a number of bytes in message (1 <= datalen <= 3).
Requires	MCU must be initialized as Master in RS-485 communication in order to be assigned an address. See <code>RS485Master_Init</code> . It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.
Example	<pre>unsigned short msg[8] ; ... RS485Master_Send(msg, 3, 0x12);</pre>

RS485Slave_Init

Prototype	<code>void Rs485slave_Init(unsigned short * port, unsigned short pin, char address);</code>
Description	Initializes MCU as Slave with a specified address in RS-485 communication. Slave address can take any value between 0 and 255, except 50, which is common address for all slaves.
Requires	USART HW module needs to be initialized. See <code>USART_Init</code> .
Example	<pre>RS485Slave_Init(PORTC, 2 ,160); // Initialize MCU as Slave with address 160</pre>

RS485Slave_Receive

Prototype	<code>void RS485Slave_Receive(char *data);</code>
Description	<p>Receives message addressed to it. Messages are multi-byte, so this function must be called for each byte received (see the example at the end of the chapter). Upon receiving a message, buffer is filled with the following values:</p> <p>data[0..2] is the message, data[3] is number of message bytes received, 1–3, data[4] is set to 255 when message is received, data[5] is set to 255 if error has occurred, data[6] is the address of the Slave which sent the message.</p> <p>Function automatically adjusts data[4] and data[5] upon every received message. These flags need to be cleared from the program.</p>
Requires	MCU must be initialized as Slave in RS-485 communication in order to be assigned an address. See RS485Slave_Init.
Example	<pre>unsigned short msg[8]; ... RS485Slave_Read(msg);</pre>

RS485Slave_Send

Prototype	<code>void RS485Slave_Send(char *data, char datalen);</code>
Description	Sends data from buffer to Master via RS-485; datalen is a number of bytes in message (1 <= datalen <= 3).
Requires	<p>MCU must be initialized as Slave in RS-485 communication in order to be assigned an address. See RS485Slave_Init.</p> <p>It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.</p>
Example	<pre>unsigned short msg[8]; ... RS485Slave_Send(msg, 2);</pre>

Library Example

The example demonstrates working with PIC as Slave node in RS-485 communication. PIC receives only packets addressed to it (address 160 in our example), and general messages with target address 50. The received data is forwarded to PORTB, and sent back to Master.

```

unsigned short dat[ 8]; // buffer for receiving/sending messages
char i = 0, j = 0;

void interrupt() {
/* Every byte is received by RS485Slave_Read(dat);
   If message is received without errors,
   data[4] is set to 255 */

   if (RCSTA.OERR) PORTD = 0x81;
   RS485Slave_Read(dat);
} //~

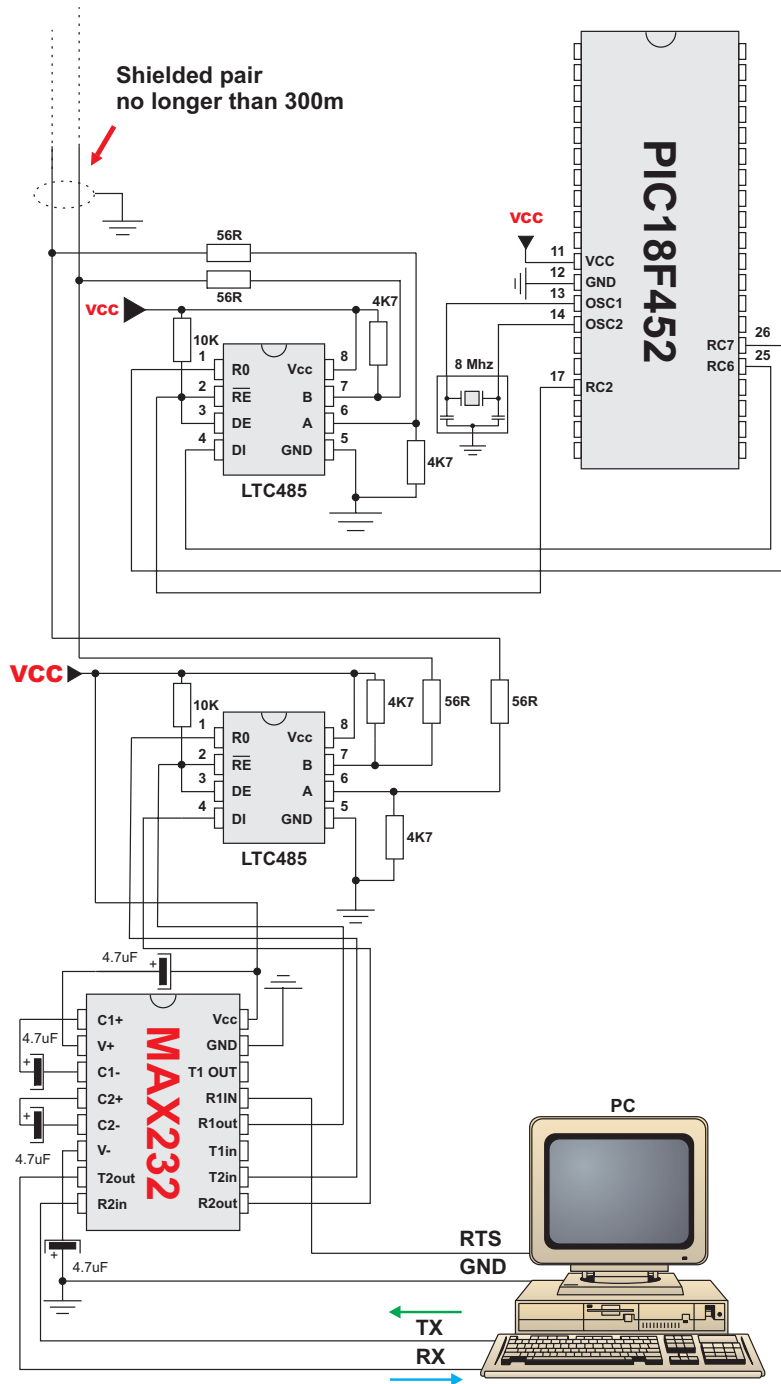
void main() {

   TRISB = 0;
   TRISD = 0;
   Usart_Init(9600); // Initialize usart module
   RS485Slave_Init(PORTC ,2 ,160); // Initialize MCU as Slave with address 160
   PIE1.RCIE = 1; // Enable interrupt
   INTCON.PEIE = 1; // on byte received
   PIE2.TXIE = 0; // via USART (RS485)
   INTCON.GIE = 1;
   PORTB = 0;
   PORTD = 0;
   dat[ 4] = 0; // Ensure that msg received flag is 0
   dat[ 5] = 0; // Ensure that error flag is 0

   do {
     if (dat[ 5] ) PORTD = 0xAA; // If there is error, set PORTD to $AA
     if (dat[ 4] ) { // If message received:
       dat[ 4] = 0; // Clear message received flag
       j = dat[ 3]; // Number of data bytes received
       for (i = 1; i < j; i++)
         PORTB = dat[ --i]; // Output received data bytes
       dat[ 0] ++; // Increment received dat[0]
       RS485Slave_Write(dat, 1); // Send it back to Master
     }
   } while (1);
} //~!

```

Hardware Connection



Software I2C Library

mikroC provides routines which implement software I²C. These routines are hardware independent and can be used with any MCU. Software I2C enables you to use MCU as Master in I2C communication. Multi-master mode is not supported.

Note: This library implements time-based activities, so interrupts need to be disabled when using Soft I²C.

Library Routines

```
Soft_I2C_Config
Soft_I2C_Start
Soft_I2C_Read
Soft_I2C_Write
Soft_I2C_Stop
```

Soft_I2C_Config

Prototype	<code>void Soft_I2C_Config(char *port, const char SCL, const char SDA, const char SCK);</code>
Description	Configures software I ² C. Parameter <code>port</code> specifies port of MCU on which SDA and SCL pins are located. Parameters SCL and SDA need to be in range 0–7 and cannot point at the same pin. Soft_I2C_Config needs to be called before using other functions from Soft I2C Library.
Example	<code>Soft_I2C_Config(PORTB, 1, 2);</code>

Soft_I2C_Start

Prototype	<code>void Soft_I2C_Start(void);</code>
Description	Issues START signal. Needs to be called prior to sending and receiving data.
Requires	Soft I ² C must be configured before using this function. See <code>Soft_I2C_Config</code> .
Example	<code>Soft_I2C_Start();</code>

Soft_I2C_Read

Prototype	<code>char Soft_I2C_Read(char ack);</code>
Returns	Returns one byte from the slave.
Description	Reads one byte from the slave, and sends not acknowledge signal if parameter <code>ack</code> is 0, otherwise it sends acknowledge.
Requires	START signal needs to be issued in order to use this function. See <code>Soft_I2C_Start</code> .
Example	<code>tmp = Soft_I2C_Read(0); //Read data, send not-acknowledge signal</code>

Soft_I2C_Write

Prototype	<code>char Soft_I2C_Write(char data);</code>
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter <code>data</code>) via I ² C bus.
Requires	START signal needs to be issued in order to use this function. See <code>Soft_I2C_Start</code> .
Example	<code>Soft_I2C_Write(0xA3);</code>

Soft_I2C_Stop

Prototype	<code>void Soft_I2C_Stop(void);</code>
Description	Issues STOP signal.
Requires	START signal needs to be issued in order to use this function. See <code>Soft_I2C_Start</code> .
Example	<code>Soft_I2C_Stop();</code>

Library Example

```
/* The example demonstrates use of Software I2C Library.  
PIC MCU is connected (SCL, SDA pins) to PCF8583 RTC (real-time clock).  
Program sends date data to RTC. */
```

```
void main() {  
  
    Soft_I2C_Config(&PORTD, 4,3); // Initialize full master mode  
  
    Soft_I2C_Start(); // Issue start signal  
    Soft_I2C_Write(0xA0); // Address PCF8583  
    Soft_I2C_Write(0); // Start from word at address 0 (config word)  
    Soft_I2C_Write(0x80); // Write 0x80 to config. (pause counter...)  
    Soft_I2C_Write(0); // Write 0 to cents word  
    Soft_I2C_Write(0); // Write 0 to seconds word  
    Soft_I2C_Write(0x30); // Write 0x30 to minutes word  
    Soft_I2C_Write(0x11); // Write 0x11 to hours word  
    Soft_I2C_Write(0x30); // Write 0x24 to year/date word  
    Soft_I2C_Write(0x08); // Write 0x08 to weekday/month  
    Soft_I2C_Stop(); // Issue stop signal  
  
    Soft_I2C_Start(); // Issue start signal  
    Soft_I2C_Write(0xA0); // Address PCF8530  
    Soft_I2C_Write(0); // Start from word at address 0  
    Soft_I2C_Write(0); // Write 0 to config word (enable counting)  
    Soft_I2C_Stop(); // Issue stop signal  
  
} //~!
```

Software SPI Library

mikroC provides library which implement software SPI. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

The library configures SPI to master mode, clock = 50kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge.

Note: These functions implement time-based activities, so interrupts need to be disabled when using the library.

Library Routines

```
Soft_Spi_Config
Soft_Spi_Read
Soft_Spi_Write
```

Soft_Spi_Config

Prototype	<code>void Soft_Spi_Config(char *port, const char SDI, const char SDO, const char SCK);</code>
Description	Configures and initializes software SPI. Parameter port specifies port of MCU on which SDI, SDO, and SCK pins will be located. Parameters SDI, SDO, and SCK need to be in range 0–7 and cannot point at the same pin. Soft_Spi_Config needs to be called before using other functions from Soft SPI Library.
Example	This will set SPI to master mode, clock = 50kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge. SDI pin is RB1, SDO pin is RB2 and SCK pin is RB3: <code>Soft_Spi_Config(PORTB, 1, 2, 3);</code>

Soft_Spi_Read

Prototype	<code>char Soft_Spi_Read(char buffer);</code>
Returns	Returns the received data.
Description	Provides clock by sending <code>buffer</code> and receives data.
Requires	Soft SPI must be initialized and communication established before using this function. See <code>Soft_Spi_Config</code> .
Example	<code>tmp = Soft_Spi_Read(buffer);</code>

Soft_Spi_Write

Prototype	<code>void Soft_Spi_Write(char data);</code>
Description	Immediately transmits data.
Requires	Soft SPI must be initialized and communication established before using this function. See <code>Soft_Spi_Config</code> .
Example	<code>Soft_Spi_Write(1);</code>

Library Example

This is a sample program which demonstrates the use of the Microchip's MCP4921 12-bit D/A converter with PIC mcu's. This device accepts digital input (number from 0..4095) and transforms it to the output voltage, ranging from 0..Vref. In this example the D/A is connected to PORTC and communicates with PIC through the SPI. The reference voltage on the mikroElektronika's DAC module is 5 V. In this example, the entire DAC's resolution range (12bit ? 4096 increments) is covered, meaning that you'll need to hold a button for about 7 minutes to get from mid-range to the end-of-range.

```

const char _CHIP_SELECT = 1, _TRUE = 0xFF;
unsigned value;

void InitMain() {
    Soft_SPI_Config(&PORTB, 4,5,3);
    TRISB &= ~(_CHIP_SELECT);           // ClearBit(TRISC,CHIP_SELECT);
    TRISC = 0x03;
} //~
// DAC increments (0..4095) --> output voltage (0..Vref)
void DAC_Output(unsigned valueDAC) {
    char temp;
    PORTB &= ~(_CHIP_SELECT);           // ClearBit(PORTC,CHIP_SELECT);
    temp = (valueDAC >> 8) & 0x0F;      // Prepare hi-byte for transfer
    temp |= 0x30;                        // It's a 12-bit number, so only
    Soft_Spi_Write(temp);                // lower nibble of high byte is used
    temp = valueDAC;                     // Prepare lo-byte for transfer
    Soft_Spi_Write(temp);
    PORTB |= _CHIP_SELECT;              // SetBit(PORTC,CHIP_SELECT);
} //~

void main() {
    InitMain();
    DAC_Output(2048);                    // When program starts, DAC gives
    value = 2048;                         // the output in the mid-range
    while (1) {                            // Main loop
        if ((Button(&PORTC,0,1,1)==_TRUE) // Test button on B0 (increment)
            && (value < 4095)) {
            value++ ;
        } else {
            if ((Button(&PORTC,1,1,1)==_TRUE) // If RB0 is not active then test
                && (value > 0)) {           // RB1 (decrement)
                value-- ;
            }
        }
    }
    DAC_Output(value);                    // Perform output
    Delay_ms(100);                         // Slow down key repeat pace
}
} //~!

```

Software UART Library

mikroC provides library which implements software UART. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via RS232 protocol – simply use the functions listed below.

Note: This library implements time-based activities, so interrupts need to be disabled when using Soft UART.

Library Routines

```
Soft_Uart_Init
Soft_Uart_Read
Soft_Uart_Write
```

Soft_Uart_Init

Prototype	<code>void Soft_Uart_Init(unsigned short *port, unsigned short rx, unsigned short tx, unsigned short baud_rate, char inverted);</code>
Description	<p>Initializes software UART. Parameter <code>port</code> specifies port of MCU on which RX and TX pins are located; parameters <code>rx</code> and <code>tx</code> need to be in range 0–7 and cannot point at the same pin; <code>baud_rate</code> is the desired baud rate. Maximum baud rate depends on PIC's clock and working conditions. Parameter <code>inverted</code>, if set to non-zero value, indicates inverted logic on output.</p> <p><code>Soft_Uart_Init</code> needs to be called before using other functions from Soft UART Library.</p>
Example	<code>Soft_Uart_Init(PORTB, 1, 2, 9600, 0);</code>

Soft_Uart_Read

Prototype	<code>unsigned short Soft_Uart_Read(unsigned short *error);</code>
Returns	Returns a received byte.
Description	Function receives a byte via software UART. Parameter <code>error</code> will be zero if the transfer was successful. This is a non-blocking function call, so you should test the <code>error</code> manually (check the example below).
Requires	Soft UART must be initialized and communication established before using this function. See <code>Soft_Uart_Init</code> .
Example	<pre>// Here's a loop which holds until data is received: do data = Soft_Uart_Read(&error); while (error); // Now we can work with it: if (data) { ...}</pre>

Soft_Uart_Write

Prototype	<code>void Soft_Uart_Write(char data);</code>
Description	Function transmits a byte (<code>data</code>) via UART.
Requires	<p>Soft UART must be initialized and communication established before using this function. See <code>Soft_Uart_Init</code>.</p> <p>Be aware that during transmission, software UART is incapable of receiving data – data transfer protocol must be set in such a way to prevent loss of information.</p>
Example	<pre>char some_byte = 0x0A; ... Soft_Uart_Write(some_byte);</pre>

Library Example

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

```
unsigned short data = 0, ro = 0;
unsigned short *er;

void main() {
    er = &ro;

    // Init (8 bit, 2400 baud rate, no parity bit, non-inverted logic)
    Soft_Uart_Init(PORTB, 1, 2, 2400, 0);

    do {
        do {
            data = Soft_Uart_Read(er);           // Receive data
        } while (*er);
        Soft_Uart_Write(data);                 // Send data via UART
    } while (1);
} //~!
```

Sound Library

mikroC provides a Sound Library which allows you to use sound signalization in your applications. You need a simple piezo speaker (or other hardware) on designated port.

Library Routines

Sound_Init
Sound_Play

Sound_Init

Prototype	<code>void Sound_Init(char *port, char pin);</code>
Description	Prepares hardware for output at specified port and pin. Parameter pin needs to be within range 0–7.
Example	<code>Sound_Init(PORTB, 2); // Initialize sound on RB2</code>

Sound_Play

Prototype	<code>void Sound_Play(char period_div_10, unsigned num_of_periods);</code>
Description	Plays the sound at the specified port and pin (see Sound_Init). Parameter period_div_10 is a sound period given in MCU cycles divided by ten, and generated sound lasts for a specified number of periods (num_of_periods).
Requires	To hear the sound, you need a piezo speaker (or other hardware) on designated port. Also, you must call Sound_Init to prepare hardware for output.
Example	To play sound of 1KHz: $T = 1/f = 1\text{ms} = 1000 \text{ cycles @ } 4\text{MHz}$. This gives us our first parameter: $1000/10 = 100$. Play 150 periods like this: <code>Sound_Play(100, 150);</code>

Library Example

The example is a simple demonstration of how to use sound library for playing tones on a piezo speaker. The code can be used with any MCU that has PORTB and ADC on PORTA. Sound frequencies in this example are generated by reading the value from ADC and using the lower byte of the result as base for T ($f = 1/T$).

```
int adcValue;

void main() {

    PORTB = 0;           // Clear PORTB
    TRISB = 0;          // PORTB is output
    INTCON = 0;         // Disable all interrupts
    ADCON1 = 0x82;      // Configure VDD as Vref, and analog channels
    TRISA = 0xFF;       // PORTA is input
    Sound_Init(PORTB, 2); // Initialize sound on RB2

    while (1) {         // Play in loop:
        adcValue = ADC_Read(2); // Get lower byte from ADC
        Sound_Play(adcValue, 200); // Play the sound
    }
}
```

SPI Library

SPI module is available with a number of PIC MCU models. mikroC 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: Certain PICmicros with two SPI modules, such as P18F8722, require you to specify the module you want to use. Simply append the number 1 or 2 to a Spi. For example, `Spi2_Write()`; Also, for the sake of backward compability with previous compiler versions and easier code management, MCU's with multiple SPI modules have SPI library which is identical to SPI1 (i.e. you can use `SPI_Init()` instead of `SPI1_Init()` for SPI operations).

Library Routines

```
Spi_Init
Spi_Init_Advanced
Spi_Read
Spi_Write
```

Spi_Init

Prototype	<code>void Spi_Init(void);</code>
Description	<p>Configures and initializes SPI with default settings. <code>SPI_Init_Advanced</code> or <code>SPI_Init</code> needs to be called before using other functions from SPI Library.</p> <p>Default settings are: Master mode, clock $F_{osc}/4$, clock idle state low, data transmitted on low to high edge, and input data sampled at the middle of interval.</p> <p>For custom configuration, use <code>Spi_Init_Advanced</code>.</p>
Requires	You need PIC MCU with hardware integrated SPI.
Example	<code>Spi_Init();</code>

Spi_Init_Advanced

Prototype	void Spi_Init_Advanced(char master, char data_sample, char clock_idle, char transmit_edge);
Description	<p>Configures and initializes SPI. Spi_Init_Advanced or SPI_Init needs to be called before using other functions of SPI Library.</p> <p>Parameter <code>mast_slav</code> determines the work mode for SPI; can have the values:</p> <pre> MASTER_OSC_DIV4 // Master clock=Fosc/4 MASTER_OSC_DIV16 // Master clock=Fosc/16 MASTER_OSC_DIV64 // Master clock=Fosc/64 MASTER_TMR2 // Master clock source TMR2 SLAVE_SS_ENABLE // Master Slave select enabled SLAVE_SS_DIS // Master Slave select disabled </pre> <p>The <code>data_sample</code> determines when data is sampled; can have the values:</p> <pre> DATA_SAMPLE_MIDDLE // Input data sampled in middle of interval DATA_SAMPLE_END // Input data sampled at the end of interval </pre> <p>Parameter <code>clock_idle</code> determines idle state for clock; can have the following values:</p> <pre> CLK_IDLE_HIGH // Clock idle HIGH CLK_IDLE_LOW // Clock idle LOW </pre> <p>Parameter <code>transmit_edge</code> can have the following values:</p> <pre> LOW_2_HIGH // Data transmit on low to high edge HIGH_2_LOW // Data transmit on high to low edge </pre>
Requires	You need PIC MCU with hardware integrated SPI.
Example	<p>This will set SPI to master mode, clock = Fosc/4, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge:</p> <pre> Spi_Init_Advanced(MASTER_OSC_DIV4, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH) </pre>

Spi_Read

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

Spi_Write

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

Library Example

The code demonstrates how to use SPI library functions. Assumed HW configuration is: max7219 (chip select pin) connected to RC1, and SDO, SDI, SCK pins are connected to corresponding pins of max7219.

```
//----- Function Declarations
void max7219_init1();
//----- F.D. end

char i;

void main() {
    Spi_Init();                // Standard configuration
    TRISC &= 0xFD;
    max7219_init1();          // Initialize max7219
    for (i = 1; i <= 8u; i++) {
        PORTC &= 0xFD;        // Select max7219
        Spi_Write(i);         // Send i to max7219 (digit place)
        Spi_Write(8 - i);     // Send i to max7219 (digit)
        PORTC |= 2;           // Deselect max7219
    }
    TRISB = 0;
    PORTB = i;
} //~!
```


USART Library

USART hardware module is available with a number of PICmicros. mikroC USART Library provides comfortable work with the Asynchronous (full duplex) mode. You can easily communicate with other devices via RS232 protocol (for example with PC, see the figure at the end of the topic – RS232 HW connection). You need a PIC MCU with hardware integrated USART, for example PIC16F877. Then, simply use the functions listed below.

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

Library Routines

```
Usart_Init
Usart_Data_Ready
Usart_Read
Usart_Write
```

Note: Certain PICmicros with two USART modules, such as P18F8520, require you to specify the module you want to use. Simply append the number 1 or 2 to a function name. For example, `Usart_Write2()`; Also, for the sake of backward compatibility with previous compiler versions and easier code management, MCU's with multiple USART modules have USART library which is identical to USART1 (i.e. you can use `Usart_Init()` instead of `Usart_Init1()` for Usart operations).

Usart_Init

Prototype	<code>void Usart_Init(const long baud_rate);</code>
Description	Initializes hardware USART module with the desired baud rate. Refer to the device data sheet for baud rates allowed for specific Fosc. If you specify the unsupported baud rate, compiler will report an error. Usart_Init needs to be called before using other functions from USART Library.
Requires	You need PIC MCU with hardware USART.
Example	<code>Usart_Init(2400); // Establish communication at 2400 bps</code>

Usart_Data_Ready

Prototype	<code>char Usart_Data_Ready(void);</code>
Returns	Function returns 1 if data is ready or 0 if there is no data.
Description	Use the function to test if data is ready for transmission.
Requires	USART HW module must be initialized and communication established before using this function. See <code>Usart_Init</code> .
Example	<pre>int receive; ... // If data is ready, read it: if (Usart_Data_Ready()) receive = Usart_Read;</pre>

Usart_Read

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

Usart_Write

Prototype	<code>char Usart_Write(char data);</code>
Description	Function transmits a byte (data) via USART.
Requires	USART HW module must be initialized and communication established before using this function. See <code>Usart_Init</code> .
Example	<pre>int chunk; ... Usart_Write(chunk); /* send data chunk via USART */</pre>

Library Example

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

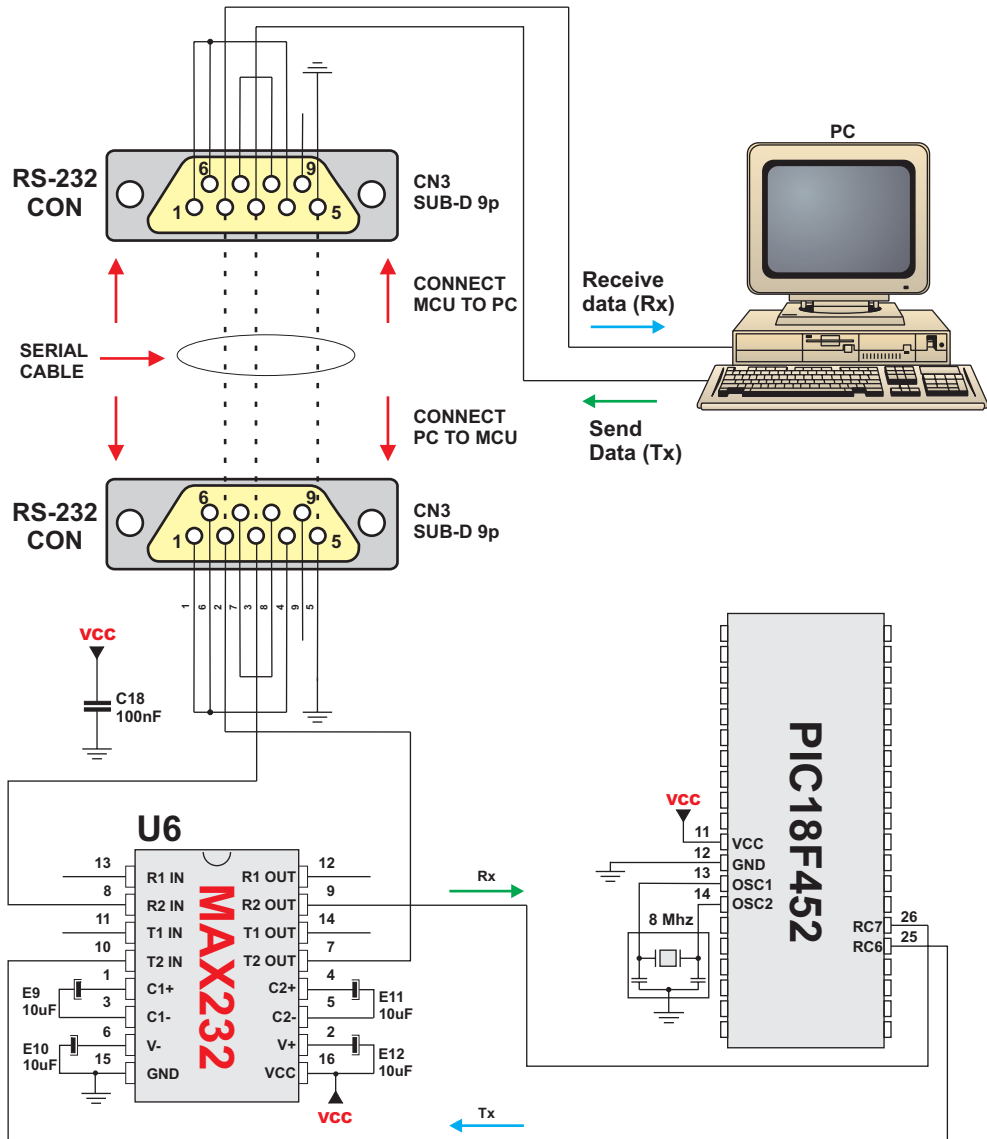
```
unsigned short i;

void main() {

    // Initialize USART module (8 bit, 2400 baud rate, no parity bit..)
    Usart_Init(2400);

    do {
        if (Usart_Data_Ready()) { // If data is received
            i = Usart_Read();      // Read the received data
            Usart_Write(i);       // Send data via USART
        }
    } while (1);
} //~!
```

Hardware Connection



USB HID Library

Universal Serial Bus (USB) provides a serial bus standard for connecting a wide variety of devices, including computers, cell phones, game consoles, PDAs, etc.

mikroC 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.

Library Routines

Hid_Enable
Hid_Read
Hid_Write
Hid_Disable

Hid_Enable

Prototype	<code>void Hid_Enable(unsigned *readbuff, unsigned *writebuff);</code>
Description	Enables USB HID communication. Parameters <code>readbuff</code> and <code>writebuff</code> 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.
Example	<code>Hid_Enable(&rd, &wr);</code>

Hid_Read

Prototype	<code>unsigned short Hid_Read(void);</code>
Returns	Number of characters in Read Buffer received from Host.
Description	Receives message from host and stores it in the Read Buffer. Function returns the number of characters received in Read Buffer.
Requires	USB HID needs to be enabled before using this function. See <code>Hid_Enable</code> .
Example	<code>get = Hid_Read();</code>

Hid_Write

Prototype	<code>void Hid_Write(unsigned *writebuff, unsigned short len);</code>
Description	Function sends data from <code>wrbuff</code> to host. Write Buffer is the same parameter as used in initialization. Parameter <code>len</code> should specify a length of the data to be transmitted.
Requires	USB HID needs to be enabled before using this function. See <code>Hid_Enable</code> .
Example	<code>Hid_Write(&wr, len);</code>

Hid_Disable

Prototype	<code>void Hid_Disable(void);</code>
Description	Disables USB HID communication.
Requires	USB HID needs to be enabled before using this function. See <code>Hid_Enable</code> .
Example	<code>Hid_Disable();</code>

Library Example

The following example continually sends sequence of numbers 0..255 to the PC via Universal Serial Bus.

```
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, INTOIE, 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
TOCON = 0x07;
TMR0H = (65536-156) >> 8;
TMR0L = (65536-156) & 0xFF;
INTCON.T0IE = 1;           // Enable TOIE
TOCON.TMR0ON = 1;
} //~

/** Main Program Routine */

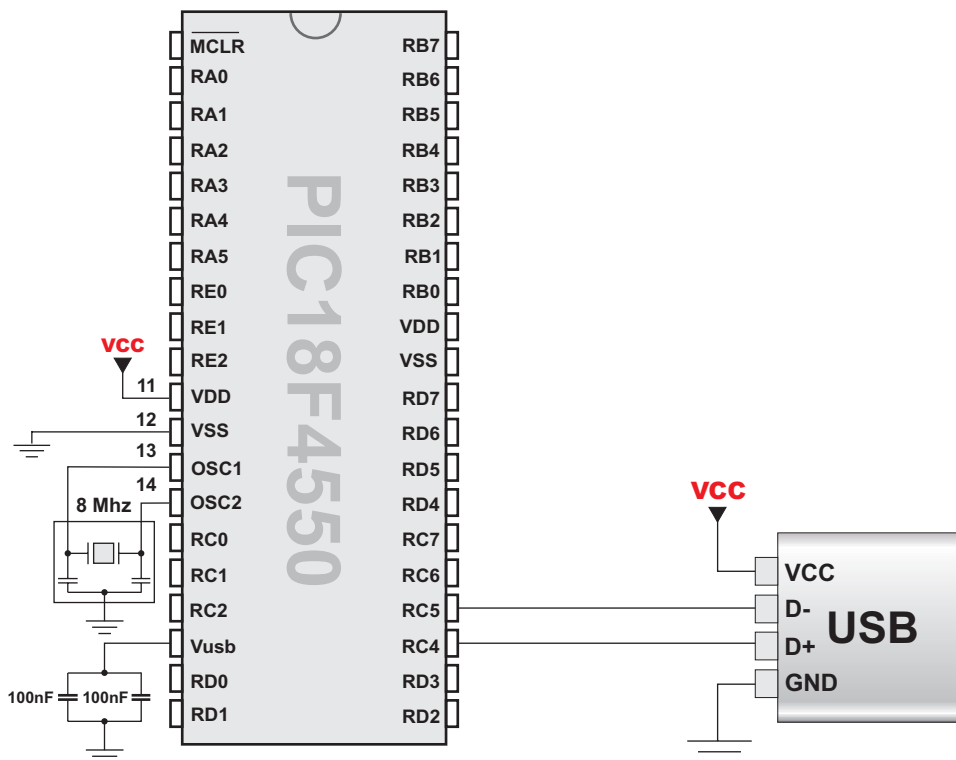
void main() {
    Init_Main();
    Hid_Enable(&userRD_buffer, &userWR_buffer);

    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



Util Library

Util library contains miscellaneous routines useful for project development.

Button

Prototype	<code>char Button(char *port, char pin, char time, char active_state);</code>
Returns	Returns 0 or 255.
Description	<p>Function eliminates the influence of contact flickering upon pressing a button (debouncing).</p> <p>Parameter <code>port</code> specifies the location of the button; parameter <code>pin</code> is the pin number on designated <code>port</code> and goes from 0..7; parameter <code>time</code> is a debounce period in milliseconds; parameter <code>active_state</code> can be either 0 or 1, and it determines if the button is active upon logical zero or logical one.</p>
Example	<p>Example reads RB0, to which the button is connected; on transition from 1 to 0 (release of button), PORTD is inverted:</p> <pre>do { if (Button(&PORTB, 0, 1, 1)) oldstate = 1; if (oldstate && Button(&PORTB, 0, 1, 0)) { PORTD = ~PORTD; oldstate = 0; } } while(1);</pre>

ANSI C Ctype Library

mikroC provides a set of standard ANSI C library functions for testing and mapping characters.

Note: Not all of the standard functions have been included. Functions have been implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming.

Library Routines

```
isalnum  
isalpha  
iscntrl  
isdigit  
isgraph  
islower  
isprint  
ispunct  
isspace  
isupper  
isxdigit  
toupper  
tolower
```

isalnum

Prototype	<code>char isalnum(char character);</code>
Description	Function returns 1 if the <code>character</code> is alphanumeric (A-Z, a-z, 0-9), otherwise returns zero.

isalpha

Prototype	<code>char isalpha(char character);</code>
Description	Function returns 1 if the <code>character</code> is alphabetic (A-Z, a-z), otherwise returns zero.

isctrl

Prototype	<code>char isctrl(char character);</code>
Description	Function returns 1 if the <code>character</code> is a control character or delete (decimal 0-31 and 127), otherwise returns zero.

isdigit

Prototype	<code>char isdigit(char character);</code>
Description	Function returns 1 if the <code>character</code> is a digit (0-9), otherwise returns zero.

isgraph

Prototype	<code>char isgraph(char character);</code>
Description	Function returns 1 if the <code>character</code> is a printable character, excluding the space (decimal 32), otherwise returns zero.

islower

Prototype	<code>char islower(char character);</code>
Description	Function returns 1 if the <code>character</code> is a lowercase letter (a-z), otherwise returns zero.

isprint

Prototype	<code>char isprint(char character);</code>
Description	Function returns 1 if the <code>character</code> is printable (decimal 32-126), otherwise returns zero.

ispunct

Prototype	<code>char ispunct(char character);</code>
Description	Function returns 1 if the <code>character</code> is punctuation (decimal 32-47, 58-63, 91-96, 123-126), otherwise returns zero.

isspace

Prototype	<code>char isspace(char character);</code>
Description	Function returns 1 if the <code>character</code> is white space (space, CR, HT, VT, NL, FF), otherwise returns zero.

isupper

Prototype	<code>char isupper(char character);</code>
Description	Function returns 1 if the <code>character</code> is an uppercase letter (A-Z), otherwise returns 0.

isxdigit

Prototype	<code>char isxdigit(char character);</code>
Description	Function returns 1 if the <code>character</code> is a hex digit (0-9, A-F, a-f), otherwise returns zero.

toupper

Prototype	<code>char toupper(int character);</code>
Description	If the <code>character</code> is a lowercase letter (a-z), function returns an uppercase letter. Otherwise, function returns an unchanged input parameter.

tolower

Prototype	<code>char tolower(int character);</code>
Description	If the <code>character</code> is an uppercase letter (A-Z), function returns a lowercase letter. Otherwise, function returns an unchanged input parameter.

ANSI C Math Library

mikroC provides a set of standard ANSI C library functions for floating point math handling.

Note: Functions have been implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming.

Library Routines

acos
asin
atan
atan2
ceil
cos
cosh
exp
fabs
floor
frexp
ldexp
log
log10
modf
pow
sin
sinh
sqrt
tan
tanh

acos

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

asin

Prototype	<code>double asin(double x);</code>
Description	Function returns the arc sine of parameter x ; that is, the value whose sine is x . Input parameter x must be between -1 and 1 (inclusive). The return value is in radians, between $-\pi/2$ and $\pi/2$ (inclusive).

atan

Prototype	<code>double atan(double x);</code>
Description	Function computes the arc tangent of parameter x ; that is, the value whose tangent is x . The return value is in radians, between $-\pi/2$ and $\pi/2$ (inclusive).

atan2

Prototype	<code>double atan2(double y, double x);</code>
Description	This is the two argument arc tangent function. It is similar to computing the arc tangent of y/x , except that the signs of both arguments are used to determine the quadrant of the result, and x is permitted to be zero. The return value is in radians, between $-\pi$ and π (inclusive).

ceil

Prototype	<code>double ceil(double num);</code>
Description	Function returns value of parameter <code>num</code> rounded up to the next whole number.

cos

Prototype	<code>double cos(double x);</code>
Description	Function returns the cosine of x in radians. The return value is from -1 to 1.

cosh

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

exp

Prototype	<code>double exp(double x);</code>
Description	Function returns the value of e — the base of natural logarithms — raised to the power of x (i.e. e^x).

fabs

Prototype	<code>double fabs(double num);</code>
Description	Function returns the absolute (i.e. positive) value of num .

floor

Prototype	<code>double floor(double num);</code>
Description	Function returns value of parameter <code>num</code> rounded down to the nearest integer.

frexp

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

ldexp

Prototype	<code>double ldexp(double num, int n);</code>
Description	Function returns the result of multiplying the floating-point number <code>num</code> by 2 raised to the power <code>exp</code> (i.e. returns $x \cdot 2^n$).

log

Prototype	<code>double log(double x);</code>
Description	Function returns the natural logarithm of <code>x</code> (i.e. $\log_e(x)$).

log10

Prototype	<code>double log10(double x);</code>
Description	Function returns the base-10 logarithm of x (i.e. $\log_{10}(x)$).

modf

Prototype	<code>double modf(double num, double *whole);</code>
Description	Function returns the signed fractional component of <code>num</code> , placing its whole number component into the variable pointed to by <code>whole</code> .

pow

Prototype	<code>double pow(double x, double y);</code>
Description	Function returns the value of x raised to the power of y (i.e. x^y). If the x is negative, function will automatically cast the y into <code>unsigned long</code> .

sin

Prototype	<code>double sin(double x);</code>
Description	Function returns the sine of x in radians. The return value is from -1 to 1.

sinh

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

sqrt

Prototype	<code>double sqrt(double num);</code>
Description	Function returns the non negative square root of num .

tan

Prototype	<code>double tan(double x);</code>
Description	Function returns the tangent of x in radians. The return value spans the allowed range of floating point in mikroC.

tanh

Prototype	<code>double tanh(double x);</code>
Description	Function returns the hyperbolic tangent of x , defined mathematically as $\sinh(x) / \cosh(x)$.

ANSI C Stdlib Library

mikroC provides a set of standard ANSI C library functions of general utility.

Note: Not all of the standard functions have been included. Functions have been implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming.

Library Routines

```
abs
atof
atoi
atol
div
ldiv
labs
max
min
rand
srand
xtoi
```

abs

Prototype	<code>int abs(int num);</code>
Description	Function returns the absolute (i.e. positive) value of <code>num</code> .

atof

Prototype	<code>double atof(char *s)</code>
Description	Function converts the input string <code>s</code> into a double precision value, and returns the value. Input string <code>s</code> should conform to the floating point literal format, with an optional white-space at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (this includes a null character).

atoi

Prototype	<code>int atoi(char *s);</code>
Description	Function converts the input string <code>s</code> into an integer value, and returns the value. Input string <code>s</code> 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 (this includes a null character).

atol

Prototype	<code>long atol(char *s)</code>
Description	Function converts the input string <code>s</code> into a long integer value, and returns the value. Input string <code>s</code> 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 (this includes a null character).

div

Prototype	<code>div_t div(int numer, int denom);</code>
Description	Function computes the result of the division of the numerator <code>numer</code> by the denominator <code>denom</code> ; function returns a structure of type <code>div_t</code> comprising quotient (<code>quot</code>) and remainder (<code>rem</code>).

ldiv

Prototype	<code>ldiv_t ldiv(long numer, long denom);</code>
Description	<p>Function is similar to the <code>div</code> function, except that the arguments and the result structure members all have type <code>long</code>.</p> <p>Function computes the result of the division of the numerator <code>numer</code> by the denominator <code>denom</code>; function returns a structure of type <code>ldiv_t</code> comprising quotient (<code>quot</code>) and remainder (<code>rem</code>).</p>

labs

Prototype	<code>long labs(long num);</code>
Description	Function returns the absolute (i.e. positive) value of a long integer <code>num</code> .

max

Prototype	<code>int max(int a, int b);</code>
Description	Function returns greater of the two integers, <code>a</code> and <code>b</code> .

min

Prototype	<code>int min(int a, int b);</code>
Description	Function returns lower of the two integers, <code>a</code> and <code>b</code> .

rand

Prototype	<code>int rand(void);</code>
Description	Function returns a sequence of pseudo-random numbers between 0 and 32767. Function will always produce the same sequence of numbers unless <code>srand()</code> is called to seed the starting point.

srand

Prototype	<code>void srand(unsigned seed);</code>
Description	Function uses the seed as a starting point for a new sequence of pseudo-random numbers to be returned by subsequent calls to <code>rand()</code> . No values are returned by this function.

xtoi

Prototype	<code>int xtoi(char *s);</code>
Description	Function converts the input string <code>s</code> consisting of hexadecimal digits into an integer value. Input parameter <code>s</code> 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 (this includes a null character).

ANSI C String Library

mikroC provides a set of standard ANSI C library functions useful for manipulating strings and arrays of `char`.

Note: Not all of the standard functions have been included. Functions have been implemented according to the ANSI C standard, but certain functions have been modified in order to facilitate PIC programming.

Library Routines

`memcmp`
`memcpy`
`memmove`
`memset`
`memchr`
`strcat`
`strchr`
`strcmp`
`strcpy`
`strlen`
`strncat`
`strncpy`
`strspn`
`strcspn`
`strncmp`
`strpbrk`
`strrchr`
`strstr`
`strtok`

memcmp

Prototype	<code>int *memcmp(void *s1, void *s2, int n);</code>
Description	Function compares the first <code>n</code> characters of objects pointed to by <code>s1</code> and <code>s2</code> , 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 <code>s1</code> is greater than the object pointed to by <code>s2</code> , and vice versa.

memcpy

Prototype	<code>void *memcpy(void *s1, void *s2, int n);</code>
Description	Function copies <i>n</i> characters from the object pointed to by <i>s2</i> into the object pointed to by <i>s1</i> . Objects may not overlap. Function returns the value of <i>s1</i> .

memmove

Prototype	<code>void *memmove(void *s1, void *s2, int n);</code>
Description	Function copies <i>n</i> characters from the object pointed to by <i>s2</i> into the object pointed to by <i>s1</i> . Unlike with <code>memcpy()</code> , memory areas <i>s1</i> and <i>s2</i> may overlap. Function returns the value of <i>s1</i> .

memset

Prototype	<code>void *memset(void *s, int c, int n);</code>
Description	Function copies the value of character <i>c</i> (converted to <code>char</code>) into each of the first <i>n</i> characters of the object pointed by <i>s</i> . Function returns the value of <i>s</i> .

memchr

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

strcat

Prototype	<code>char *strcat(char *s1, char *s2);</code>
Description	Function appends the string <code>s2</code> to the string <code>s1</code> , overwriting the null character at the end of <code>s1</code> . Then, a terminating null character is added to the result. Strings may not overlap, and <code>s1</code> must have enough space to store the result. Function returns a resulting string <code>s1</code> .

strchr

Prototype	<code>char *strchr(char *s, char c);</code>
Description	Function locates the first occurrence of character <code>c</code> in the string <code>s</code> . Function returns a pointer to the <code>c</code> , or a null pointer if <code>c</code> does not occur in <code>s</code> . The terminating null character is considered to be a part of the string.

strcmp

Prototype	<code>char strcmp(char *s1, char *s2);</code>
Description	Function compares strings <code>s1</code> and <code>s2</code> , 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 <code>s1</code> is greater than <code>s2</code> , and vice versa.

strcpy

Prototype	<code>char *strcpy(char *s1, char *s2);</code>
Description	Function copies the string <code>s2</code> into the string <code>s1</code> . If successful, function returns <code>s1</code> . The strings may not overlap.

strlen

Prototype	<code>unsigned strlen(char *s);</code>
Description	Function returns the length of the string <code>s</code> (the terminating null character does not count against string's length).

strncat

Prototype	<code>char *strncat(char *s1, char *s2, int n);</code>
Description	Function appends not more than <code>n</code> characters from the string <code>s2</code> to <code>s1</code> . The initial character of <code>s2</code> overwrites the null character at the end of <code>s1</code> . A terminating null character is always appended to the result. Function returns <code>s1</code> .

strncpy

Prototype	<code>char *strncpy(char *s1, char *s2, int n);</code>
Description	Function copies not more than <code>n</code> characters from string <code>s2</code> to <code>s1</code> . The strings may not overlap. If <code>s2</code> is shorter than <code>n</code> characters, then <code>s1</code> will be padded out with null characters to make up the difference. Function returns the resulting string <code>s1</code> .

strspn

Prototype	<code>int strspn(char *s1, char *s2);</code>
Description	Function returns the length of the maximum initial segment of <code>s1</code> which consists entirely of characters from <code>s2</code> . The terminating null character character at the end of the string is not compared.

strcspn

Prototype	<code>char strcspn(char * s1, char * s2);</code>
Description	The <code>strcspn</code> function computes the length of the maximum initial segment of the string pointed to by <code>s1</code> which consists entirely of characters not from the string pointed to by <code>s2</code> . Function returns the length of the segment.

strncmp

Prototype	<code>int strncmp(char * s1, char * s2, char len);</code>								
Description	<p>Function lexicographically compares the first n bytes of the strings s1 and s2 and returns a value indicating their relationship:</p> <table border="0"> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>< 0</td> <td>s1 "less than" s2</td> </tr> <tr> <td>= 0</td> <td>s1 "equal to" s2</td> </tr> <tr> <td>> 0</td> <td>s1 "greater than" s2</td> </tr> </tbody> </table> <p>The value returned by function is determined by the difference between the values of the first pair of bytes that differ in the strings being compared (within first n bytes).</p>	Value	Meaning	< 0	s1 "less than" s2	= 0	s1 "equal to" s2	> 0	s1 "greater than" s2
Value	Meaning								
< 0	s1 "less than" s2								
= 0	s1 "equal to" s2								
> 0	s1 "greater than" s2								

strpbrk

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

strpbrk

Prototype	<code>char * strrchr(char * ptr, unsigned int chr);</code>
Description	<p>Function searches the string ptr for the last occurrence of character ch. The null character terminating ptr is not included in the search. Function returns an index of the last ch found in ptr; if no matching character was found, function returns \$FF.</p>

strstr

Prototype	<code>char * strstr(char * s1, char * s2);</code>
Description	<p>Function locates the first occurrence of the string s2 in the string s1 (excluding the terminating null character).</p> <p>Function returns a number indicating the position of the first occurrence of s2 in s1; if no string was found, function returns \$FF. If s2 is a null string, the function returns 0.</p>

strtok

Prototype	<code>char * strtok(char * s1, char * s2);</code>
Returns	The strtok function returns a pointer to the first character of a token, or a null pointer if there is no token.
Description	<p>A sequence of calls to the strtok function breaks the string pointed to by s1 into a sequence of tokens, each of which is delimited by a character from the string pointed to by s2. The first call in the sequence has s1 as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by s2 may be different from call to call.</p> <p>The first call in the sequence searches the string pointed to by s1 for the first character that is not contained in the current separator string pointed to by s2. If no such character is found, then there are no tokens in the string pointed to by s1 and the strtok function returns a null pointer. If such character is found, it is the start of the first token.</p> <p>The strtok function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by s1, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The strtok function saves a pointer to the following character, from which the next search for a token will start.</p> <p>Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.</p>
Example	<pre>char x[10] ; void main(){ strcpy(x, strtok("mikroE1", "Ek")); strcpy(x, strtok(0, "kE")); }</pre>

Conversions Library

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

Library Routines

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

```
ByteToStr
ShortToStr
WordToStr
IntToStr
LongToStr
FloatToStr
```

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

```
Bcd2Dec
Dec2Bcd
Bcd2Dec16
Dec2Bcd16
```

ByteToStr

Prototype	<code>void ByteToStr(unsigned short number, char *output);</code>
Description	Function creates an output string out of a small unsigned number (numerical value less than 0x100). Output string has fixed width of 3 characters; remaining positions on the left (if any) are filled with blanks.
Example	<pre>unsigned short t = 24; char txt[4]; //... ByteToStr(t, txt); // txt is " 24" (one blank here)</pre>

ShortToStr

Prototype	<code>void ShortToStr(short number, char *output);</code>
Description	Function creates an output string out of a small signed number (numerical value less than 0x100). Output string has fixed width of 4 characters; remaining positions on the left (if any) are filled with blanks.
Example	<pre>short t = -24; char txt[5]; //... ByteToStr(t, txt); // txt is " -24" (one blank here)</pre>

WordToStr

Prototype	<code>void WordToStr(unsigned number, char *output);</code>
Description	Function creates an output string out of an unsigned number (numerical value of unsigned type). Output string has fixed width of 5 characters; remaining positions on the left (if any) are filled with blanks.
Example	<pre>unsigned t = 437; char txt[6]; //... WordToStr(t, txt); // txt is " 437" (two blanks here)</pre>

IntToStr

Prototype	<code>void IntToStr(int number, char *output);</code>
Description	Function creates an output string out of a signed number (numerical value of int type). Output string has fixed width of 6 characters; remaining positions on the left (if any) are filled with blanks.
Example	<pre>int j = -4220; char txt[7]; //... IntToStr(j, txt); // txt is " -4220" (one blank here)</pre>

LongToStr

Prototype	<code>void LongToStr(long number, char *output);</code>
Description	Function creates an output string out of a large signed number (numerical value of long type). Output string has fixed width of 11 characters; remaining positions on the left (if any) are filled with blanks.
Example	<pre>long jj = -3700000; char txt[12]; //... LongToStr(jj, txt); // txt is " -3700000" (three blanks here)</pre>

FloatToStr

Prototype	<code>void FloatToStr(float number, char *output);</code>
Description	Function creates an output string out of a floating-point number. The output string contains a normalized format of the number (mantissa between 0 and 1) with sign at the first position. Mantissa has fixed format of six digits, 0.dxxxx; i.e. there will always be 5 digits following the dot. The output string must be at least 13 characters long.
Example	<pre>float ff = -374.2; char txt[13]; //... FloatToStr(ff, txt); // txt is "-0.37420e3"</pre>

Bcd2Dec

Prototype	<code>unsigned short Bcd2Dec(unsigned short bcdnum);</code>
Returns	Returns converted decimal value.
Description	Converts 8-bit BCD numeral bcdnum to its decimal equivalent.
Example	<pre>unsigned short a; ... a = Bcd2Dec(0x52); // equals 52</pre>

Dec2Bcd

Prototype	<code>unsigned short Dec2Bcd(unsigned short decnum);</code>
Returns	Returns converted BCD value.
Description	Converts 8-bit decimal value <code>decnum</code> to BCD.
Example	<pre> unsigned short a; ... a = Dec2Bcd(52); // equals 0x52 </pre>

Bcd2Dec16

Prototype	<code>unsigned Bcd2Dec16(unsigned bcdnum);</code>
Returns	Returns converted decimal value.
Description	Converts 16-bit BCD numeral <code>bcdnum</code> to its decimal equivalent.
Example	<pre> unsigned a; ... a = Bcd2Dec16(1234); // equals 4660 </pre>

Dec2Bcd16

Prototype	<code>unsigned Dec2Bcd16(unsigned decnum);</code>
Returns	Returns converted BCD value.
Description	Converts 16-bit decimal value <code>decnum</code> to BCD.
Example	<pre> unsigned a; ... a = Dec2Bcd16(4660); // equals 1234 </pre>

Trigonometry Library

mikroC implements fundamental trigonometry functions. These functions are implemented as lookup tables, and return the result as integer, multiplied by 1000 and rounded up.

Library Routines

SinE3
CosE3

SinE3

Prototype	<code>int SinE3(unsigned angle_deg);</code>
Returns	Function returns the sine of input parameter, multiplied by 1000 (1E3) and rounded up to the nearest integer. The range of return values is from -1000 to 1000.
Description	Function takes parameter <code>angle_deg</code> which represents angle in degrees, and returns its sine multiplied by 1000 and rounded up to the nearest integer. The function is implemented as a lookup table; maximum error obtained is ± 1 .
Example	<code>res = SinE3(45); // result is 707</code>

CosE3

Prototype	<code>int CosE3(unsigned angle_deg);</code>
Returns	Function returns the cosine of input parameter, multiplied by 1000 (1E3) and rounded up to the nearest integer. The range of return values is from -1000 to 1000.
Description	Function takes parameter <code>angle_deg</code> which represents angle in degrees, and returns its cosine multiplied by 1000 and rounded up to the nearest integer. The function is implemented as a lookup table; maximum error obtained is ± 1 .
Example	<code>res = CosE3(196); // result is -193</code>

Sprint Library

Library for sprint functions.

Note: In addition to ANSI C standard mikroC provides limited versions `sprinti`, `sprintf` that take a less ROM, RAM and may be convenient in some cases for PIC.

Library Routines

`sprintf`
`sprintf`
`sprinti`

sprintf

Description: Function formats a series of strings and numeric values and stores the resulting string in *buffer*.

Note: format string must be in the CONST area. `sprintf` function is not supported for p12 and p16 PIC MCU families.

The *fmtstr* 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 *buffer* 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 references the first argument after *fmtstr* and converts and outputs it using the format specification. The second format specification accesses the second argument after *fmtstr*, 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. Format specifications have the following format:

`% [flags] [width] [.precision] [{ h | 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 the argument is interpreted as a character, a string, a number, or a 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	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	The 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 %%.

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 a + 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
#	Prefixes 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. The # flag is ignored in all other cases.
*	Ignore format specifier.

The *width* field is a non-negative number that specifies the minimum number of characters printed. If the number of characters in the output value is less than width, blanks are added on the left or right (when the - flag is specified) to pad to the minimum width. If width is prefixed with a 0, 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, the number of significant digits, or the 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	Meaning of 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 the 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.

The optional characters h and l or L may immediately precede the *conversion_type* to respectively specify short or 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*.

sprintf

Prototype	<pre>int sprintf (char *buffer, /* storage buffer */ const char *fmtstr, /* format string */ ...); /* additional arguments */</pre>
Returns	Function returns the number of characters actually written to buffer.
Description	Same as printf, except it doesn't support float-type numbers.

sprintfi

Prototype	<pre>int sprintfi (char *buffer, /* storage buffer */ const char *fmtstr, /* format string */ ...); /* additional arguments */</pre>
Returns	Function returns the number of characters actually written to buffer.
Description	Same as printf, except it doesn't support long integer type numbers.

SPI Graphic LCD Library

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

Important: When using SPI Library routines, you are required to specify the actual SPI module, either SPI1 or SPI2 in `Spi_Glcd_Init`.

Note: `SPI_Init()` must be called before initializing SPI GLCD.

Library Routines

Basic routines:

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

Advanced routines:

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

Spi_Glcd_Init

Prototype	<code>void SPI_Glcd_Init(char DeviceAddress, unsigned int * rstport, unsigned int rstpin, unsigned int * cspport, unsigned int cspin);</code>
Description	Initializes Graphic LCD 128x64 via SPI. RstPort and RstPin - Sets pin connected on reset pin of spi expander. CSPort and CSPin - Sets pin connected on CS pin of spi expander. device address - address of spi expander (hardware setting of A0, A1 and A2 pins (connected on VCC or GND) on spi expander).
Requires	Note: SPI_Init() must be called before initializing SPI GLCD. This procedure needs to be called before using other routines of SPI GLCD library.
Example	<code>Spi_Glcd_Init(0, &PORTC, 0, &PORTC, 1);</code>

Spi_Glcd_Set_Side

Prototype	<code>void SPI_Glcd_Set_Side(char x_pos);</code>
Description	Selects side of GLCD, left or right. Parameter x specifies the side: values from 0 to 63 specify the left side, and values higher than 64 specify the right side. Use the functions Spi_Glcd_Set_Side, Spi_Glcd_Set_X, and Spi_Glcd_Set_Page to specify an exact position on GLCD. Then, you can use Spi_Glcd_Write_Data or Spi_Glcd_Read_Data on that location.
Requires	GLCD needs to be initialized. See Spi_Glcd_Init.
Example	<code>Spi_Glcd_Select_Side(0); Spi_Glcd_Select_Side(10);</code>

Spi_Glcd_Set_Page

Prototype	<code>void Spi_Glcd_Set_Page(char page);</code>
Description	Selects page of GLCD, technically a line on display; parameter <code>page</code> can be 0..7.
Requires	GLCD needs to be initialized. See <code>Spi_Glcd_Init</code> .
Example	<code>Spi_Glcd_Set_Page(5);</code>

Spi_Glcd_Set_X

Prototype	<code>void SPI_Glcd_Set_X(char x_pos);</code>
Description	Positions to <code>x</code> dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See <code>Spi_Glcd_Init</code> .
Example	<code>Spi_Glcd_Set_X(25);</code>

Spi_Glcd_Read_Data

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

Spi_Glcd_Write_Data

Prototype	<code>void Spi_Glcd_Write_Data(char data);</code>
Description	Writes data to the current location in GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized. See <code>Spi_Glcd_Init</code> .
Example	<code>Spi_Glcd_Write_Data(data)</code>

Spi_Glcd_Fill

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

Spi_Glcd_Dot

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

Spi_Glcd_Line

Prototype	<code>void SPI_Glcd_Line(int x_start, int y_start, int x_end, int y_end, char color);</code>
Description	Draws a line on the GLCD from (x1, y1) to (x2, y2). Parameter <code>color</code> determines the dot state: 0 draws an empty line (clear dots), 1 draws a full line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See <code>Spi_Glcd_Init</code> .
Example	<code>Spi_Glcd_Line(0, 63, 50, 0, 2);</code>

Spi_Glcd_V_Line

Prototype	<code>void Spi_Glcd_V_Line(char y_start, char y_end, char x_pos, char color);</code>
Description	Draws a vertical line on the GLCD from (x, y1) to (x, y2). Parameter <code>color</code> determines the dot state: 0 draws an empty line (clear dots), 1 draws a solid line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See <code>Spi_Glcd_Init</code> .
Example	<code>Spi_Glcd_V_Line(0, 63, 0, 1);</code>

Spi_Glcd_H_Line

Prototype	<code>void Spi_Glcd_H_Line(char x_start, char x_end, char y_pos, char color);</code>
Description	Draws a horizontal line on the GLCD from (x1, y) to (x2, y). Parameter <code>color</code> determines the dot state: 0 draws an empty line (clear dots), 1 draws a solid line (put dots), and 2 draws a “smart” line (invert each dot).
Requires	GLCD needs to be initialized. See <code>Spi_Glcd_Init</code> .
Example	<code>Spi_Glcd_H_Line(0, 127, 0, 1);</code>

Spi_Glcd_Rectangle

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

Spi_Glcd_Box

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

Spi_Glcd_Circle

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

Spi_Glcd_Set_Font

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

Spi_Glcd_Write_Char

Prototype	<code>void SPI_Glcd_Write_Char(char chr1, char x_pos, char page_num, char color);</code>
Description	<p>Prints character at page (one of 8 GLCD lines, 0..7), <code>x</code> dots away from the left border of display. Parameter <code>color</code> defines the "fill": 0 writes a "white" letter (clear dots), 1 writes a solid letter (put dots), and 2 writes a "smart" letter (invert each dot).</p> <p>Use routine <code>Spi_Glcd_Set_Font</code> to specify font, or the default 5x7 font (included with the library) will be used.</p>
Requires	GLCD needs to be initialized, see <code>Spi_Glcd_Init</code> . Use the <code>Spi_Glcd_Set_Font</code> to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	<pre>Spi_Glcd_Write_Char('C', 0, 0, 1);</pre>

Spi_Glcd_Write_Text

Prototype	<code>void SPI_Glcd_Write_Text(char text[], char x_pos, char page_num, char color);</code>
Description	<p>Prints <code>text</code> at page (one of 8 GLCD lines, 0..7), <code>x</code> dots away from the left border of display. Parameter <code>color</code> defines the “fill”: 0 prints a “white” letters (clear dots), 1 prints solid letters (put dots), and 2 prints “smart” letters (invert each dot).</p> <p>Use routine <code>Spi_Glcd_Set_Font</code> to specify font, or the default 5x7 font (included with the library) will be used.</p>
Requires	GLCD needs to be initialized, see <code>Spi_Glcd_Init</code> . Use the <code>Spi_Glcd_Set_Font</code> to specify the font for display; if no font is specified, the default 5x8 font supplied with the library will be used.
Example	<code>Spi_Glcd_Write_Text('Hello world!', 0, 0, 1);</code>

Spi_Glcd_Image

Prototype	<code>void Spi_Glcd_Image(const char * image);</code>
Description	Displays bitmap image on the GLCD. Parameter <code>image</code> should be formatted as an array of 1024 bytes. Use the mikroPascal’s integrated Bitmap-to-LCD editor (menu option Tools > Graphic LCD Editor) to convert image to a constant array suitable for display on GLCD.
Requires	GLCD needs to be initialized. See <code>Spi_Glcd_Init</code> .
Example	<code>Spi_Glcd_Image(my_image);</code>

Library Example

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

```
extern const unsigned short
truck_bmp[ ];

char ii;
unsigned int jj;
char *someText;

void delay2S() {
    Delay_ms(2000);
}

void main() {
    ADCON1 |= 0x0F;

    Spi_Init();    // Initialize SPI module

    Spi_Glcd_Init(0,&PORTC, 0, &PORTC, 1);
    Spi_Glcd_Fill(0xAA);
    delay2S();
    while(1) {
        Spi_Glcd_Fill(0x00);
        Spi_Glcd_Image( truck_bmp );
        delay2S();

        for(jj = 1; jj <= 40; jj++)
            Spi_Glcd_Dot(jj,jj,1);
        delay2S();

        Spi_Glcd_Fill(0x00);
        Spi_Glcd_Line(120, 1, 5,60, 1);
        delay2S();
        Spi_Glcd_Line(12, 42, 5,60, 1);
        delay2S();

        Spi_Glcd_Rectangle(12, 20, 93,57, 1);
        delay2S();

        //continues..
    }
}
```

```
//continues..

Spi_Glcd_Line(120, 12, 12,60, 1);
delay2S();

Spi_Glcd_H_Line(5, 40, 6, 1);
delay2S();
Spi_Glcd_Line(0, 12, 120, 60, 1);
Spi_Glcd_V_Line(7, 63, 127, 1);
delay2S();

for(ii = 1; ii <= 10; ii++)
    Spi_Glcd_Circle(63, 32, 3*ii, 1);

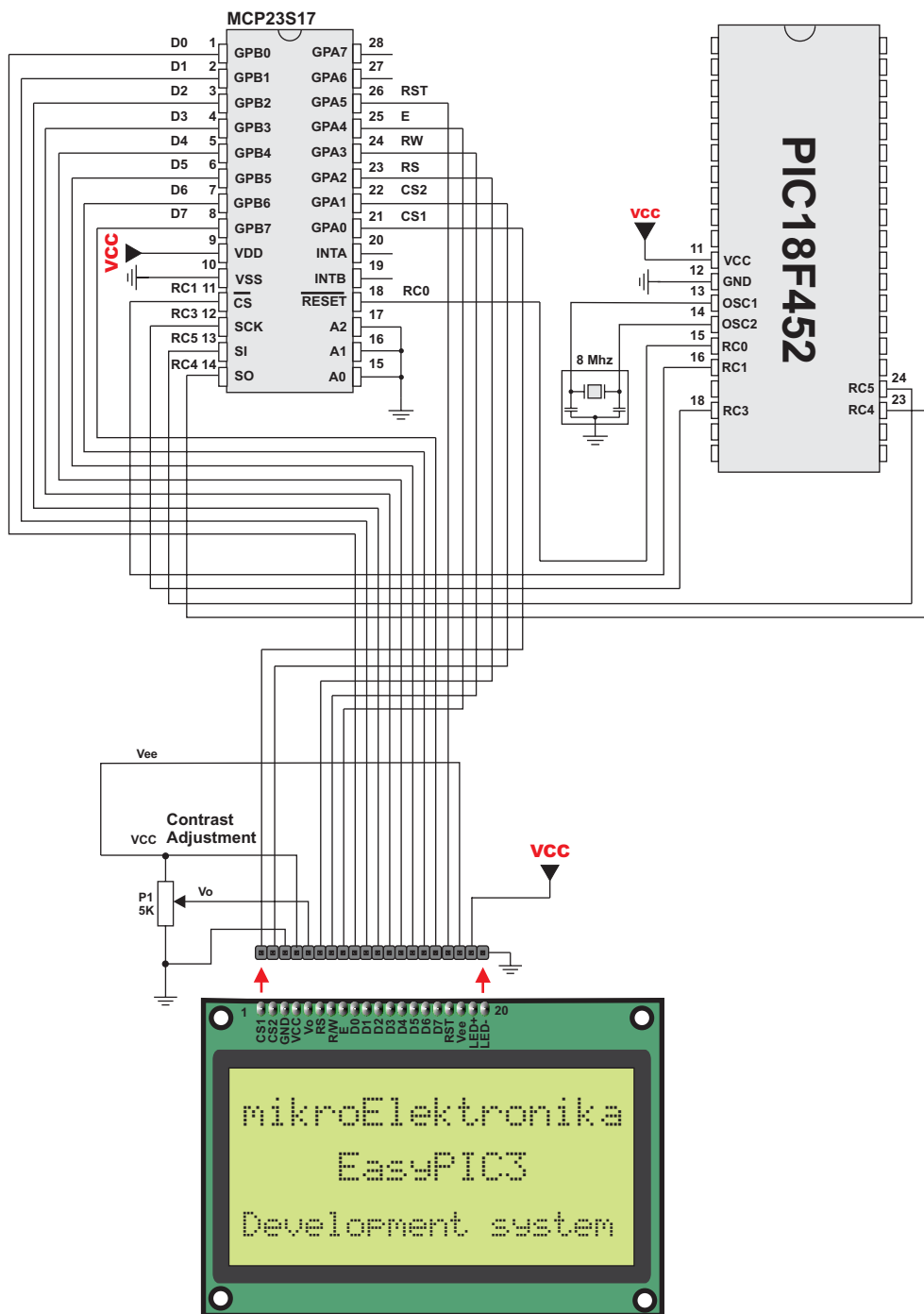
delay2S();
Spi_Glcd_Box(12, 20, 70, 57, 2);
delay2S();

Spi_Glcd_Fill(0x00);

Spi_Glcd_Set_Font(&System3x6, 3, 6, 32);
someText = "SMALL FONT: 3X6";
Spi_Glcd_Write_Text(someText, 20, 5, 1);

Spi_Glcd_Set_Font(&FontSystem5x8, 5, 8, 32);
someText = "Large Font 5x8";
Spi_Glcd_Write_Text(someText, 3, 4, 1);
delay2S();
}
}
```

Hardware Connection



Port Expander Library

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

Note: PIC need to have a hardware SPI module.

Note: `SPI_Init()` must be called before initializing Port Expander

Library Routines

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

Expander_Init

Prototype	<code>void Expander_Init(char ModuleAddress, unsigned int * rstport, unsigned int rstpin, unsigned int * csport, unsigned int cspin);</code>
Description	Establishes SPI communication with the expander and initializes the expander. <code>RstPort</code> and <code>RstPin</code> - Sets pin connected on reset pin of spi expander. <code>CSPort</code> and <code>CSPin</code> - Sets pin connected on CS pin of spi expander. <code>moduleaddress</code> - address of spi expander (hardware setting of A0, A1 and A2 pins (connected on VCC or GND) on spi expander).
Requires	This procedure needs to be called before using other routines of PORT Expander library. <code>SPI_Init()</code> must be called before initializing Port Expander.
Example	<code>Expander_Init(0, &PORTC, 0, &PORTC, 1);</code>

PortExpanderSelect

Prototype	<code>void PortExpanderSelect;</code>
Description	Selects current port expander.
Requires	PORT Expander must be initialized. See <code>Expander_Init</code> .
Example	<code>PortExpanderSelect;</code>

PortExpanderUnSelect

Prototype	<code>void PortExpanderUnSelect;</code>
Description	Un-Selects current port expander.
Requires	PORT Expander must be initialized. See <code>Expander_Init</code> .
Example	<code>PortExpanderUnSelect;</code>

Expander_Read_Byte

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

Expander_Write_Byte

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

Expander_Set_Mode

Prototype	<code>void Expander_Set_Mode(char ModuleAddress, char Mode);</code>
Returns	Nothing.
Description	Sets port expander mode on ModuleAddress.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	<code>Expander_Set_Mode(1, 0);</code>

Expander_Read_ArrayPortA

Prototype	<code>void Expander_Read_ArrayPortA(char ModuleAddress, char NoBytes, char DestArray[]);</code>
Returns	Nothing.
Description	This routine reads array of bytes (DestArray) from port expander on ModuleAddress and portA. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	<code>Expander_Read_PortA(0, 1, data);</code>

Expander_Read_Array

Prototype	<code>void Expander_Read_Array(char ModuleAddress, char StartAddress, char NoBytes, char *DestArray);</code>
Returns	Nothing.
Description	This routine reads array of bytes (DestArray) from port expander on ModuleAddress and StartAddress. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	<code>Expander_Read_Array(1, 1, 5, data);</code>

Expander_Write_Array

Prototype	<code>void Expander_Write_Array(char ModuleAddress, char StartAddress, char NoBytes, char *SourceArray);</code>
Returns	Nothing.
Description	This routine writes array of bytes (DestArray) to port expander on ModuleAddress and StartAddress. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	<code>Expander_Write_Array(1,1,5,data);</code>

Expander_Read_PortA

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

Expander_Read_ArrayPortB

Prototype	<code>void Expander_Read_ArrayPortB(char ModuleAddress, char NoBytes, char DestArray[]);</code>
Returns	Nothing.
Description	This routine reads array of bytes (DestArray) from port expander on ModuleAddress and portB. NoBytes represents number of read bytes.
Requires	PORT Expander must be initialized. See Expander_Init.
Example	<code>Expander_Read_PortB(0, 8, data);</code>

Expander_Write_PortA

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

Expander_Write_PortB

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

Expander_Set_DirectionPortA

Prototype	<code>void Expander_Set_DirectionPortA(char ModuleAddress, char Data);</code>
Description	Set port expander PortA pin as input or output.
Requires	PORT Expander must be initialized. See <code>Expander_Init</code> .
Example	<code>Expander_Set_DirectionPortA(0, \$FF);</code>

Expander_Set_DirectionPortB

Prototype	<code>void Expander_Set_DirectionPortB(char ModuleAddress, char Data);</code>
Description	Set port expander PortB pin as input or output.
Requires	PORT Expander must be initialized. See <code>Expander_Init</code> .
Example	<code>Expander_Set_DirectionPortB(0, \$FF);</code>

Expander_Set_PullUpsPortA

Prototype	<code>void Expander_Set_PullUpsPortA(char ModuleAddress, char Data);</code>
Description	This routine sets port expander PortA pin as pullup or pulldown.
Requires	PORT Expander must be initialized. See <code>Expander_Init</code> .
Example	<code>Expander_Set_PullUpsPortA(0, \$FF);</code>

Expander_Set_PullUpsPortB

Prototype	<code>void Expander_Set_PullUpsPortB(char ModuleAddress, char Data);</code>
Description	This routine sets port expander PortB pin as pullup or pulldown.
Requires	PORT Expander must be initialized. See <code>Expander_Init</code> .
Example	<code>Expander_Set_PullUpsPortB(0, \$FF);</code>

Library Example

The example demonstrates how to communicate to port expander MCP23S17.

```
unsigned char i;

void main(){
    ADCON1 |= 0x0f;
    TRISB = 0x00;
    LATB = 0xFF;
    Delay_ms(2000);

    Spi_Init();    // Initialize SPI module

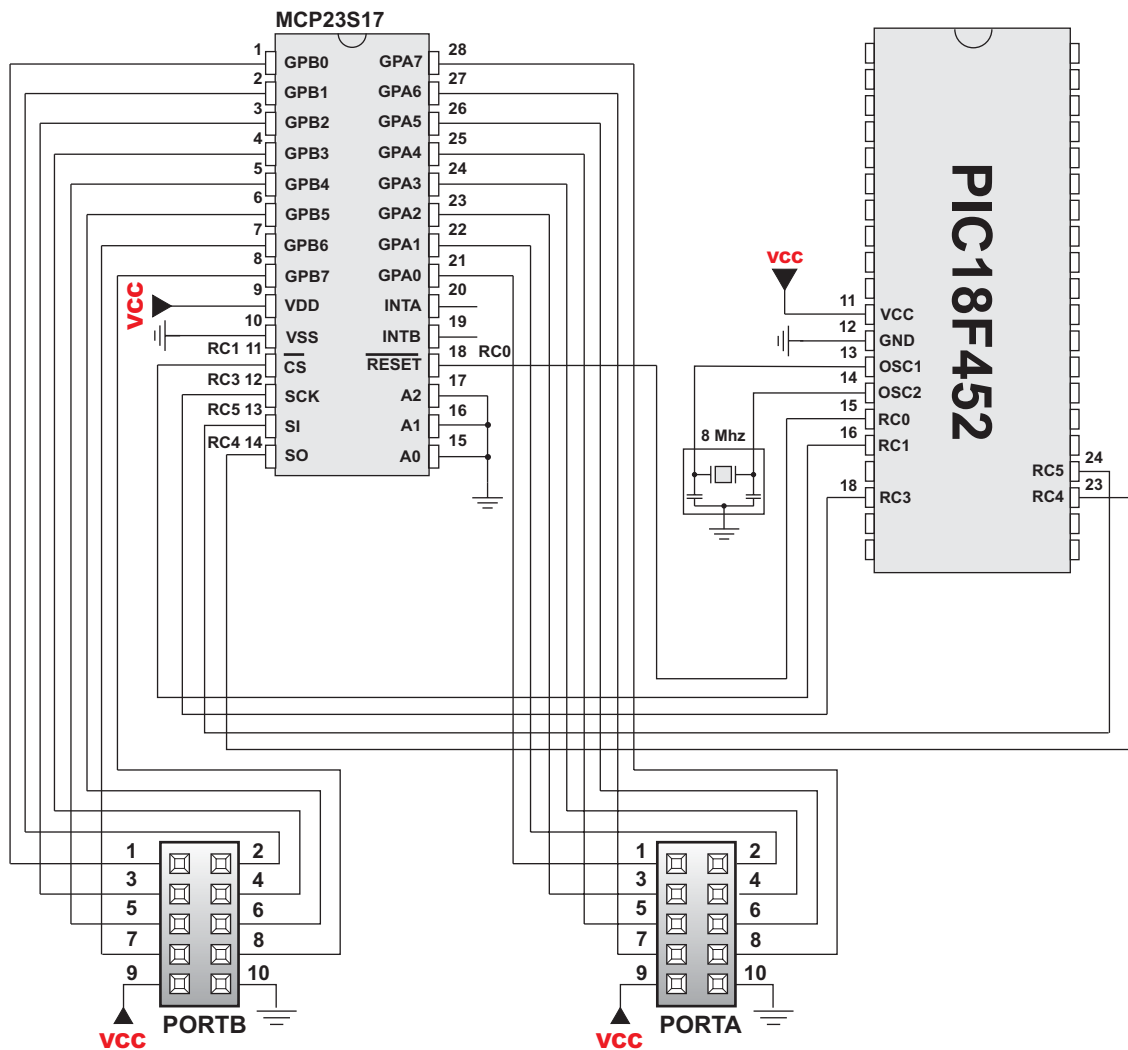
    Expander_Init(0, &PORTC, 0, &PORTC, 1);    // initialize port expander

    Expander_Set_DirectionPortA(0, 0);    // set expander's porta to be output

    Expander_Set_DirectionPortB(0,0xFF);    // set expander's porta to be input
    Expander_Set_PullUpsPortB(0,0xFF);
    // set pull ups to all of the expander's portb pins

    i = 0;
    while(1) {
        Expander_Write_PortA(0, i++);    // write i to expander's porta
        LATB = Expander_Read_PortB(0);
        // read expander's portb and write it to PIC's LATB
        Delay_ms(20);
    }
}
```

Hardware Connection



SPI LCD Library (4-bit interface)

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

Note: Spi_Init(); must be called before initializing SPI LCD.

Library Routines

```
Spi_Lcd_Config
Spi_Lcd_Init
Spi_Lcd_Out
Spi_Lcd_Out_Cp
Spi_Lcd_Chr
Spi_Lcd_Chr_Cp
Spi_Lcd_Cmd
```

Spi_Lcd_Config

Prototype	<code>void Spi_Lcd_Config(char DeviceAddress, unsigned char * rstport, unsigned char rstpin, unsigned char * csport, unsigned char cspin);</code>
Description	Initializes LCD via SPI interface with pin settings (Reset pin and Chip Select pin) you specify.
Requires	<code>Spi_Init();</code> must be called before initializing SPI LCD.
Example	<code>Spi_Lcd_Config(0, &PORTB, 1, &PORTB, 0);</code>

Spi_Lcd_Init

Prototype	<code>void Spi_Lcd_Init();</code>
Description	Initializes LCD at port with default pin settings (see the connection scheme at the end of the chapter).
Requires	<code>Spi_Init();</code> must be called before initializing SPI LCD.
Example	<code>Spi_Lcd_Init();</code>

Spi_Lcd_Out

Prototype	<code>void Spi_Lcd_Out(char row, char column, char *text);</code>
Description	Prints text on LCD at specified row and column (parameters row and col). Both string variables and literals can be passed as text.
Requires	Port with LCD must be initialized. See <code>Spi_Lcd_Config</code> or <code>Spi_Lcd_Init</code> .
Example	<code>Spi_Lcd_Out(1, 3, "Hello!");</code>

Spi_Lcd_Out_Cp

Prototype	<code>void Spi_Lcd_Out_CP(char *text);</code>
Description	Prints text on LCD at current cursor position. Both string variables and literals can be passed as text.
Requires	Port with LCD must be initialized. See <code>Spi_Lcd_Config</code> or <code>Spi_Lcd_Init</code> .
Example	<code>Spi_Lcd_Out_Cp("Here!"); // Print "Here!" at current cursor position</code>

Spi_Lcd_Chr

Prototype	<code>void Spi_Lcd_Chr(char Row, char Column, char Out_Char);</code>
Description	Prints character on LCD at specified row and column (parameters row and col). Both variables and literals can be passed as character.
Requires	Port with LCD must be initialized. See Spi_Lcd_Config or Spi_Lcd_Init.
Example	<code>Spi_Lcd_Chr(2, 3, "i");</code>

Spi_Lcd_Chr_Cp

Prototype	<code>void Spi_Lcd_Chr_CP(char Out_Char);</code>
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as character.
Requires	Port with LCD must be initialized. See Spi_Lcd_Config or Spi_Lcd_Init.
Example	<code>Spi_Lcd_Chr_Cp("e"); // Print "e" at current cursor position</code>

Spi_Lcd_Cmd

Prototype	<code>void Spi_Lcd_Cmd(char out_char);</code>
Description	Sends command to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown below.
Requires	Port with LCD must be initialized. See Spi_Lcd_Config or Spi_Lcd_Init.
Example	<code>Spi_Lcd_Cmd(LCD_Clear); // Clear LCD display</code>

LCD Commands

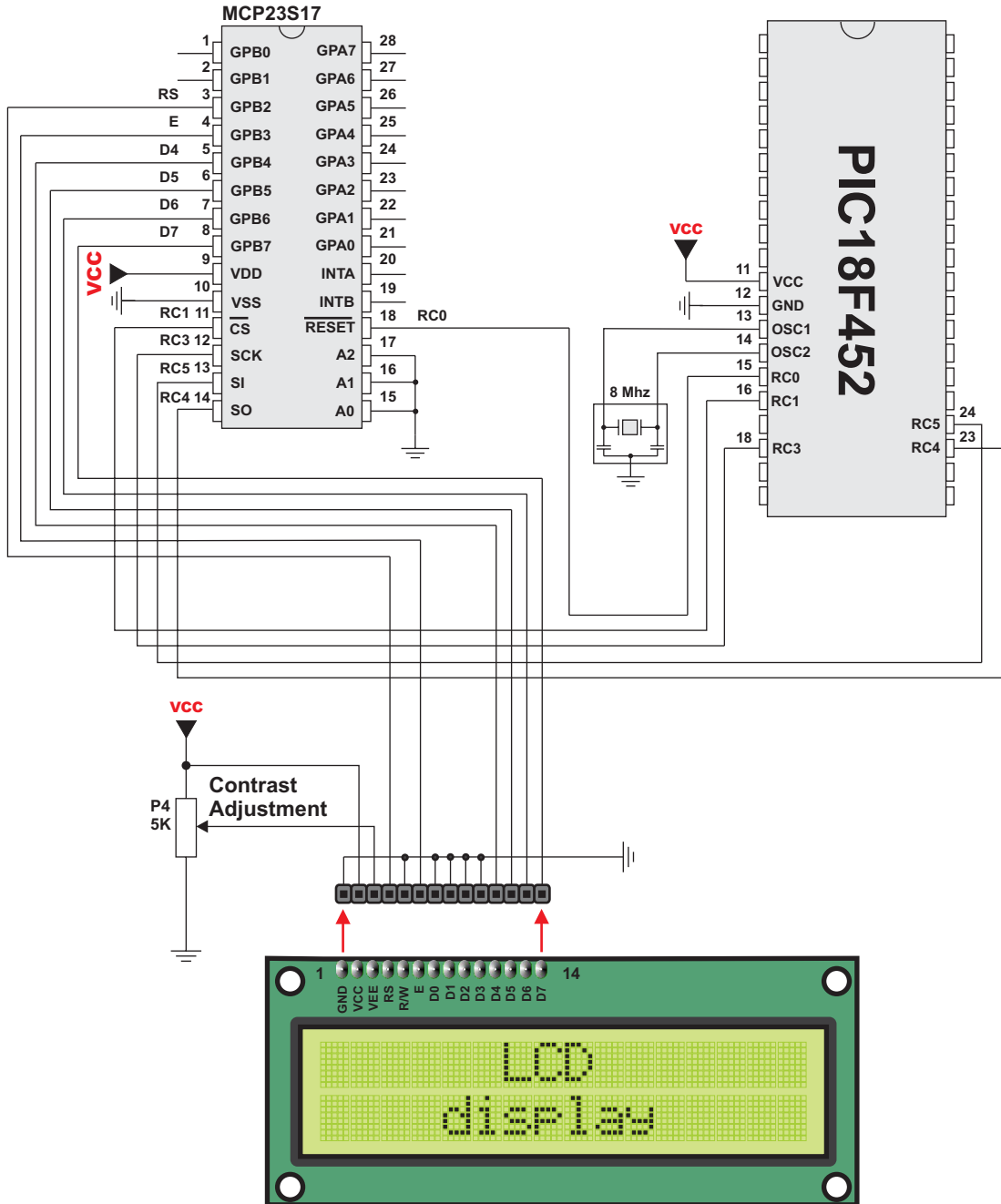
LCD Command	Purpose
LCD_FIRST_ROW	Move cursor to 1st row
LCD_SECOND_ROW	Move cursor to 2nd row
LCD_THIRD_ROW	Move cursor to 3rd row
LCD_FOURTH_ROW	Move cursor to 4th row
LCD_CLEAR	Clear display
LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to original position. Display data RAM is unaffected.
LCD_CURSOR_OFF	Turn off cursor
LCD_UNDERLINE_ON	Underline cursor on
LCD_BLINK_CURSOR_ON	Blink cursor on
LCD_MOVE_CURSOR_LEFT	Move cursor left without changing display data RAM
Lcd_Move_Cursor_Right	Move cursor right without changing display data RAM
LCD_TURN_ON	Turn LCD display on
LCD_TURN_OFF	Turn LCD display off
LCD_SHIFT_LEFT	Shift display left without changing display data RAM
LCD_SHIFT_RIGHT	Shift display right without changing display data RAM

Library Example (default pin settings)

```
char *text = "mikroElektronika";

void main() {
    Spi_Init();                // initialize spi
    Spi_Lcd_Init();           // 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, 7th column
    Spi_Lcd_Chr_CP('!');      // append !
    Spi_Lcd_Out(2,0, text);    // Print text to LCD, 2nd row, 3rd 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
} //~!
```

Hardware Connection



SPI LCD8 (8-bit interface) Library

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

Note: Spi_Init(); must be called before initializing SPI LCD8.

Library Routines

```
Spi_Lcd8_Config
Spi_Lcd8_Init
Spi_Lcd8_Out
Spi_Lcd8_Out_Cp
Spi_Lcd8_Chr
Spi_Lcd8_Chr_Cp
Spi_Lcd8_Cmd
```

Spi_Lcd8_Config

Prototype	<code>void Spi_Lcd8_Config(char DeviceAddress, unsigned char * rstport, unsigned char rstpin, unsigned char * csport, unsigned char cspin);</code>
Description	Initializes LCD via SPI interface with pin settings (Reset pin and Chip Select pin) you specify.
Requires	<code>Spi_Init()</code> ; must be called before initializing SPI LCD8.
Example	<code>Spi_Lcd8_Config(0, &PORTB, 1, &PORTB, 0);</code>

Spi_Lcd8_Init

Prototype	<code>void Spi_Lcd8_Init();</code>
Description	Initializes LCD at Control port (ctrlport) and Data port (dataport) with default pin settings (see the connection scheme at the end of the chapter).
Requires	<code>Spi_Init()</code> ; must be called before initializing SPI LCD8.
Example	<code>Spi_Lcd8_Init();</code>

Spi_Lcd8_Out

Prototype	<code>void Spi_Lcd8_Out(unsigned short row, unsigned short column, char *text);</code>
Description	Prints text on LCD at specified row and column (parameters row and col). Both string variables and literals can be passed as text.
Requires	Ports with LCD must be initialized. See <code>Spi_Lcd8_Config</code> or <code>Spi_Lcd8_Init</code> .
Example	<code>Spi_Lcd8_Out(1, 3, "Hello!"); // Print "Hello!" at line 1, char 3</code>

Spi_Lcd8_Out_Cp

Prototype	<code>void Spi_Lcd8_Out_CP(char *text);</code>
Description	Prints text on LCD at current cursor position. Both string variables and literals can be passed as text.
Requires	Ports with LCD must be initialized. See <code>Spi_Lcd8_Config</code> or <code>Spi_Lcd8_Init</code> .
Example	<code>Spi_Lcd8_Out_Cp("Here!"); // Print "Here!" at current cursor position</code>

Spi_Lcd8_Chr

Prototype	<code>void Spi_Lcd8_Chr(unsigned short row, unsigned short column, char out_char);</code>
Description	Prints character on LCD at specified row and column (parameters row and col). Both variables and literals can be passed as character.
Requires	Ports with LCD must be initialized. See Spi_Lcd8_Config or Spi_Lcd8_Init.
Example	<code>Spi_Lcd8_Out(2, 3, "i"); // Print "i" at line 2, char 3</code>

Spi_Lcd8_Chr_Cp

Prototype	<code>void Spi_Lcd8_Chr_CP(char out_char);</code>
Description	Prints character on LCD at current cursor position. Both variables and literals can be passed as character.
Requires	Ports with LCD must be initialized. See Spi_Lcd8_Config or Spi_Lcd8_Init.
Example	<code>Spi_Lcd8_Chr_Cp("e"); // Print "e" at current cursor position</code>

Spi_Lcd8_Cmd

Prototype	<code>void Spi_Lcd8_Cmd(char out_char);</code>
Description	Sends command to LCD. You can pass one of the predefined constants to the function. The complete list of available commands is shown below.
Requires	Ports with LCD must be initialized. See Spi_Lcd8_Config or Spi_Lcd8_Init.
Example	<code>Spi_Lcd8_Cmd(LCD_Clear); // Clear LCD display</code>

LCD Commands

LCD Command	Purpose
LCD_FIRST_ROW	Move cursor to 1st row
LCD_SECOND_ROW	Move cursor to 2nd row
LCD_THIRD_ROW	Move cursor to 3rd row
LCD_FOURTH_ROW	Move cursor to 4th row
LCD_CLEAR	Clear display
LCD_RETURN_HOME	Return cursor to home position, returns a shifted display to original position. Display data RAM is unaffected.
LCD_CURSOR_OFF	Turn off cursor
LCD_UNDERLINE_ON	Underline cursor on
LCD_BLINK_CURSOR_ON	Blink cursor on
LCD_MOVE_CURSOR_LEFT	Move cursor left without changing display data RAM
Lcd_Move_Cursor_Right	Move cursor right without changing display data RAM
LCD_TURN_ON	Turn LCD display on
LCD_TURN_OFF	Turn LCD display off
LCD_SHIFT_LEFT	Shift display left without changing display data RAM
LCD_SHIFT_RIGHT	Shift display right without changing display data RAM

Library Example (default pin settings)

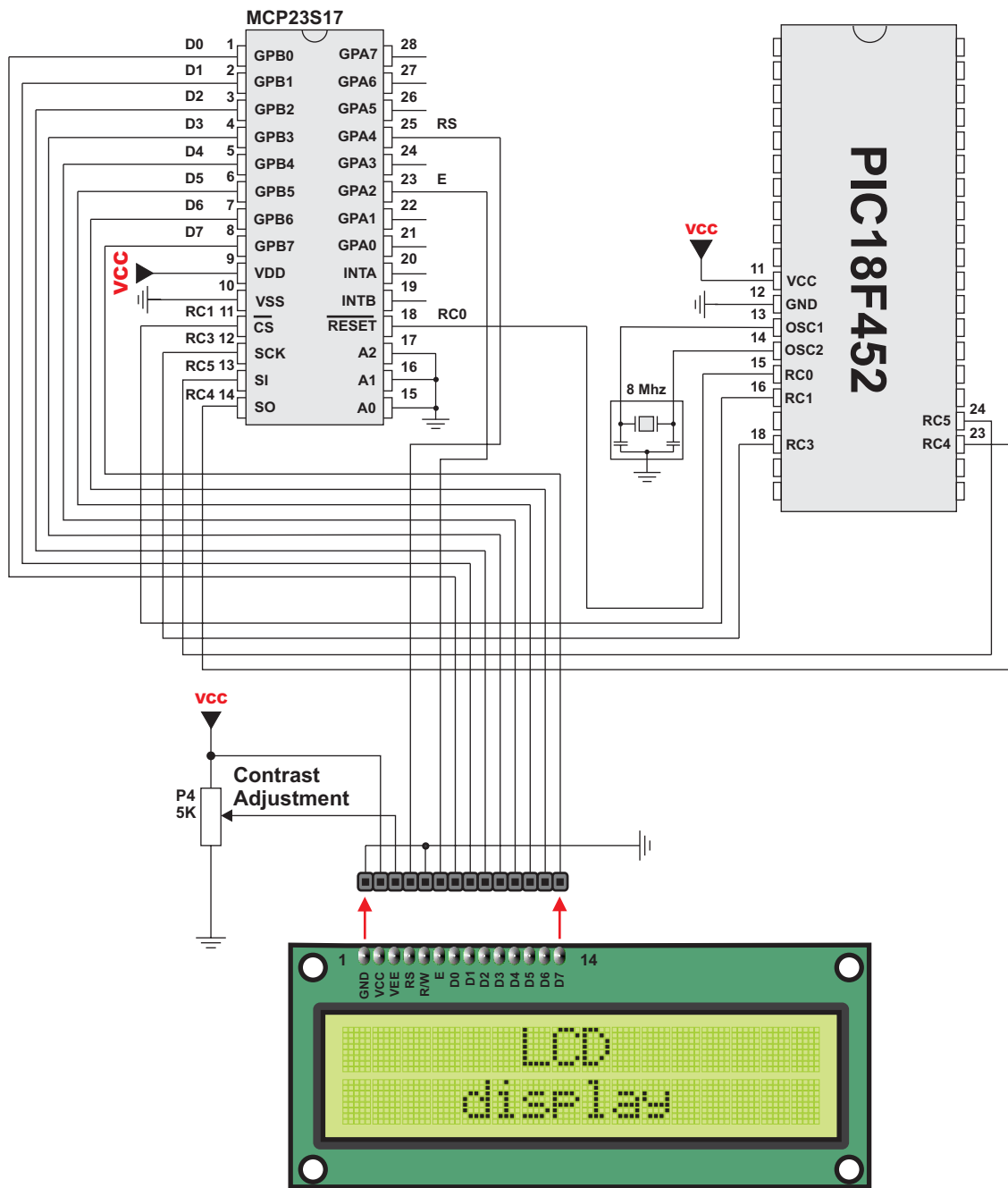
```

char *text = "mikroE";

void main() {
    Spi_Init(); // initialize spi interface
    Spi_Lcd8_Init(); // initialize 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, 7th column...
    Spi_Lcd8_Chr_CP('!'); // append '!'
    Spi_Lcd8_Out(2,0, "mikroelektronika");// Print text to LCD, 2nd row, 3rd 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
} //~!

```

Hardware Connection



SPI T6963C Graphic LCD Library

mikroC for PIC provides a library for drawing and writing on Toshiba T6963C Graphic LCD (various sizes) via SPI interface.

Note: Spi_Init; must be called before initializing SPI LCD.

Library Routines

```
Spi_T6963C_Config  
Spi_T6963C_writeData  
Spi_T6963C_writeCommand  
Spi_T6963C_setPtr  
Spi_T6963C_waitReady  
Spi_T6963C_fill  
Spi_T6963C_dot  
Spi_T6963C_write_char  
Spi_T6963C_write_text  
Spi_T6963C_line  
Spi_T6963C_rectangle  
Spi_T6963C_box  
Spi_T6963C_circle  
Spi_T6963C_image  
Spi_T6963C_sprite  
Spi_T6963C_set_cursor  
Spi_T6963C_clearBit  
Spi_T6963C_setBit  
Spi_T6963C_negBit  
Spi_T6963C_displayGrPanel  
Spi_T6963C_displayTxtPanel  
Spi_T6963C_setGrPanel  
Spi_T6963C_setTxtPanel  
Spi_T6963C_panelFill  
Spi_T6963C_grFill  
Spi_T6963C_txtFill  
Spi_T6963C_cursor_height  
Spi_T6963C_graphics  
Spi_T6963C_text  
Spi_T6963C_cursor  
Spi_T6963C_cursor_blink  
Spi_T6963C_Config_240x128  
Spi_T6963C_Config_240x64
```

Spi_T6963C_Config

Prototype	<pre>void Spi_T6963C_Config(unsigned int width, unsigned char height, unsigned char fntW, char DeviceAddress, unsigned char * rstport, unsigned char rstpin, unsigned char * csport, unsigned char cspin, unsigned char wr, unsigned char rd, unsigned char cd, unsigned char rst);</pre>
Description	<p>Initializes the Graphic Lcd controller. This function must be called before all Spi T6963C Library Routines.</p> <p>width - Number of horizontal (x) pixels in the display. height - Number of vertical (y) pixels in the display. fntW - Font width, number of pixels in a text character, must be set accordingly to the hardware. data - Address of the port on which the Data Bus is connected. cntrl - Address of the port on which the Control Bus is connected. wr - !WR line bit number in the *cntrl port. rd - !RD line bit number in the *cntrl port. cd - !CD line bit number in the *cntrl port. rst - !RST line bit number in the *cntrl port. DeviceAddress - Device Address.</p> <p>Display RAM : The library doesn't know the amount of available RAM. The library cuts the RAM into panels : a complete panel is one graphics panel followed by a text panel, The programmer has to know his hardware to know how much panel he has.</p>
Requires	<p>Spi_Init(); must be called before initializing SPI Toshiba T6963C Graphic LCD.</p>
Example	<pre>Spi_T6963C_Config(240, 64, 8, &PORTB, 1, &PORTB, 0, 0, 1, 3, 4, 0); /* * init display for 240 pixel width and 64 pixel height * 8 bits character width * reset pin on PORTB.1 * chip select pin on PORTB.0 * bit 0 is !WR * bit 1 is !RD * bit 3 is !CD * bit 4 is RST * chip enable, reverse on, 8x8 font internally set in library * device address is 0 */</pre>

Spi_T6963C_writeData

Prototype	<code>void Spi_T6963C_writeData(unsigned char data);</code>
Description	Routine that writes data to Spi T6963C controller.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_writeData(AddrL);</code>

Spi_T6963C_writeCommand

Prototype	<code>void Spi_T6963C_writeCommand(unsigned char data);</code>
Description	Routine that writes command to Spi T6963C controller
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_writeCommand(T6963C_CURSOR_POINTER_SET);</code>

Spi_T6963C_setPtr

Prototype	<code>void Spi_T6963C_setPtr(unsigned int addr, unsigned char t);</code>
Description	This routine sets the memory pointer p for command c.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_setPtr(T6963C_grHomeAddr + start, T6963C_ADDRESS_POINTER_SET);</code>

Spi_T6963C_waitReady

Prototype	<code>void Spi_T6963C_waitReady();</code>
Description	This routine pools the status byte, and loops until ready.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_waitReady();</code>

Spi_T6963C_fill

Prototype	<code>void Spi_T6963C_fill(unsigned char data, unsigned int start, unsigned int len);</code>
Description	This routine fills length with bytes to controller memory from start address.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_fill(0x33,0x00FF,0x000F);</code>

Spi_T6963C_dot

Prototype	<code>void Spi_T6963C_dot(int x, int y, unsigned char color);</code>
Description	This routine sets current graphic work panel. It sets the pixel dot (x0, y0). <code>pcolor = T6963C_[WHITE[BLACK]].</code>
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_dot(x0, y0, pcolor);</code>

Spi_T6963C_write_char

Prototype	<code>void Spi_T6963C_write_char(unsigned char c, unsigned char x, unsigned char y, unsigned char mode);</code>
Description	This routine sets current text work panel. It writes char c row x line y. <code>mode = T6963C_ROM_MODE_[OR EXOR AND]</code>
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_write_char("A",22,23,AND);</code>

Spi_T6963C_write_text

Prototype	<code>void Spi_T6963C_write_text(unsigned char *str, unsigned char x, unsigned char y, unsigned char mode);</code>
Description	This sets current text work panel. It writes string str row x line y. <code>mode = T6963C_ROM_MODE_[OR EXOR AND]</code>
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_write_text("GLCD LIBRARY DEMO, WELCOME !", 0, 0, T6963C_ROM_MODE_XOR);</code>

Spi_T6963C_line

Prototype	<code>void Spi_T6963C_line(int px0, int py0, int px1, int py1, unsigned char pcolor);</code>
Description	This routine current graphic work panel. It's draw a line from (x0, y0) to (x1, y1). pcolor = T6963C_[WHITE[BLACK]
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_line(0, 0, 239, 127, T6963C_WHITE);</code>

Spi_T6963C_rectangle

Prototype	<code>void Spi_T6963C_rectangle(int x0, int y0, int x1, int y1, unsigned char pcolor);</code>
Description	It sets current graphic work panel. It draws the border of the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_rectangle(20, 20, 219, 107, T6963C_WHITE);</code>

Spi_T6963C_box

Prototype	<code>void Spi_T6963C_box(int x0, int y0, int x1, int y1, unsigned char pcolor);</code>
Description	This routine sets current graphic work panel. It draws a solid box in the rectangle (x0, y0)-(x1, y1). pcolor = T6963C_[WHITE[BLACK].
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_box(0, 119, 239, 127, T6963C_WHITE);</code>

Spi_T6963C_circle

Prototype	<code>void Spi_T6963C_circle(int x, int y, long r, unsigned char pcolor);</code>
Description	This routine sets current graphic work panel. It draws a circle, center is (x, y), diameter is r. pcolor = T6963C_[WHITE[BLACK]
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_circle(120, 64, 110, T6963C_WHITE);</code>

Spi_T6963C_image

Prototype	<code>void Spi_T6963C_image(const char *pic);</code>
Description	This routine sets current graphic work panel : It fills graphic area with picture pointer by MCU. MCU must fit the display geometry. For example : for a 240x128 display, MCU must be an array of $(240/8)*128 = 3840$ bytes .
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_image(my_image);</code>

Spi_T6963C_sprite

Prototype	<code>void Spi_T6963C_sprite(unsigned char px, unsigned char py, const char *pic, unsigned char sx, unsigned char sy);</code>
Description	This routine sets current graphic work panel. It fills graphic rectangle area (px, py)-(px + sx, py + sy) witch picture pointed by MCU. Sx and sy must be the size of the picture. MCU must be an array of sx*sy bytes.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_sprite(76, 4, einstein, 88, 119); // draw a sprite</code>

Spi_T6963C_set_cursor

Prototype	<code>void Spi_T6963C_set_cursor(unsigned char x, unsigned char y);</code>
Description	This routine sets cursor row x line y.
Requires	Ports must be initialized. See Spi_T6963C_init.
Example	<code>Spi_T6963C_set_cursor(cposx, cposy);</code>

Spi_T6963C_clearBit

Prototype	<code>void Spi_T6963C_clearBit(char b);</code>
Description	Clear control bit.
Requires	Ports must be initialized. See Spi_T6963C_init.
Example	<code>Spi_T6963C_clearBit(b);</code>

Spi_T6963C_setBit

Prototype	<code>void Spi_T6963C_setBit(char b);</code>
Description	Set control bit.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_setBit(b);</code>

Spi_T6963C_negBit

Prototype	<code>void Spi_T6963C_negBit(char b);</code>
Description	Neg control bit.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_negBit(b);</code>

Spi_T6963C_displayGrPanel

Prototype	<code>void Spi_T6963C_waitReady(unsigned int n);</code>
Description	Display graphic panel number n.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_displayGrPanel(n);</code>

Spi_T6963C_displayTxtPanel

Prototype	<code>void Spi_T6963C_displayTxtPanel(unsigned int n);</code>
Description	Display text panel number n.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_displayTxtPanel(n);</code>

Spi_T6963C_setGrPanel

Prototype	<code>void Spi_T6963C_setGrPanel(unsigned int n);</code>
Description	Compute graphic start address for panel number n.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_setGrPanel(n);</code>

Spi_T6963C_setTxtPanel

Prototype	<code>void Spi_T6963C_setTxtPanel(unsigned int n);</code>
Description	Compute text start address for panel number n.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_setTxtPanel(n);</code>

Spi_T6963C_panelFill

Prototype	<code>void Spi_T6963C_panelFill(unsigned int v);</code>
Description	Fill full #n panel with v bitmap (0 to clear).
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_panelFill(v);</code>

Spi_T6963C_grFill

Prototype	<code>void Spi_T6963C_grFill(unsigned int v);</code>
Description	Fill graphic #n panel with v bitmap (0 to clear).
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_grFill(v);</code>

Spi_T6963C_txtFill

Prototype	<code>void Spi_T6963C_txtFill(unsigned int v);</code>
Description	Fill text #n panel with char v + 32 (0 to clear).
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_txtFill(v);</code>

Spi_T6963C_cursor_height

Prototype	<code>void Spi_T6963C_cursor_height(unsigned int n);</code>
Description	Set cursor size.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_cursor_height(n);</code>

Spi_T6963C_graphics

Prototype	<code>void Spi_T6963C_graphics(unsigned int n);</code>
Description	Set graphics on/off.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_graphics(1);</code>

Spi_T6963C_text

Prototype	<code>void Spi_T6963C_text(unsigned int n);</code>
Description	Set text on/off.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_text(1);</code>

Spi_T6963C_cursor

Prototype	<code>void Spi_T6963C_cursor(unsigned int n);</code>
Description	Set cursor on/off.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_cursor(1);</code>

Spi_T6963C_cursor_blink

Prototype	<code>void Spi_T6963C_cursor_blink(unsigned int n);</code>
Description	Set cursor blink on/off.
Requires	GLCD needs to be initialized, see Spi_T6963C_Config.
Example	<code>Spi_T6963C_cursor_blink(0);</code>

Spi_T6963C_Config_240x128

Prototype	procedure Spi_T6963C_Config_240x128 ();
Description	Initialize T6963C based GLCD (240x128 pixels) with default settings for mE GLCD's.
Requires	Spi_Init; must be called before initializing SPI Toshiba T6963C Graphic LCD.
Example	Spi_T6963C_Config_240x128 ();

Spi_T6963C_Config_240x64

Prototype	procedure Spi_T6963C_Config_240x64 ();
Description	Set graphics on/off.
Requires	Initialize T6963C based GLCD (240x64 pixels) with default settings for mE GLCD's.
Example	Spi_T6963C_Config_240x64 ();

Library Example

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

```
#include          "Spi_T6963C.h"

extern const char mc[] ;
extern const char einstein[] ;

void main(void)
{
    unsigned char    panel ;           // current panel
    unsigned int     i ;               // general purpose register
    unsigned char    curs ;           // cursor visibility
    unsigned int     cposx, cposy ;    // cursor x-y position

    TRISC = 0 ;                       // port C is output only
    PORTC = 0b00000000 ; // chip enable, reverse on, 8x8 font

    //continues...
```



```
//continues...

/*
 * init display for 240 pixel width and 128 pixel height
 * 8 bits character width
 * data bus on PORTD
 * control bus on PORTC
 * bit 3 is !WR
 * bit 2 is !RD
 * bit 1 is C!D
 * bit 5 is RST
 */

Spi_Init();
Spi_T6963C_Init_240x128();

/*
 * 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(" GLCD LIBRARY DEMO, WELCOME !", 0,
0, Spi_T6963C_ROM_MODE_XOR) ;
Spi_T6963C_write_text(" EINSTEIN WOULD HAVE LIKED mC", 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

//continued...
```

```

//continues...

/*
 * 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) ;

// draw a sprite
Spi_T6963C_sprite(76, 4, einstein, 88, 119) ;

Spi_T6963C_setGrPanel(1) ; // select other graphic panel

// fill the graphic screen with a picture
Spi_T6963C_image(mc) ;

//continued...

```

```
//continues...

    for(;;)
    {

// if RB1 is pressed, toggle the display between graphic panel 0
// and graphic 1

        if(PORTB & 0b00000010)
        {
            panel++;
            panel &= 1 ;
            Spi_T6963C_displayGrPanel(panel) ;
            Delay_ms(300) ;
        }

// if RB2 is pressed, display only graphic panel
else if(PORTB & 0b00000100)
        {
            Spi_T6963C_graphics(1) ;
            Spi_T6963C_text(0) ;
            Delay_ms(300) ;
        }

// if RB3 is pressed, display only text panel

else if(PORTB & 0b00001000)
        {
            Spi_T6963C_graphics(0) ;
            Spi_T6963C_text(1) ;
            Delay_ms(300) ;
        }

// if RB4 is pressed, display text and graphic panels

else if(PORTB & 0b00010000)
        {
            Spi_T6963C_graphics(1) ;
            Spi_T6963C_text(1) ;
            Delay_ms(300) ;
        }

//continued...
```

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

```

// if RB5 is pressed, change cursor
else if(PORTB & 0b00100000)
{
    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) ;

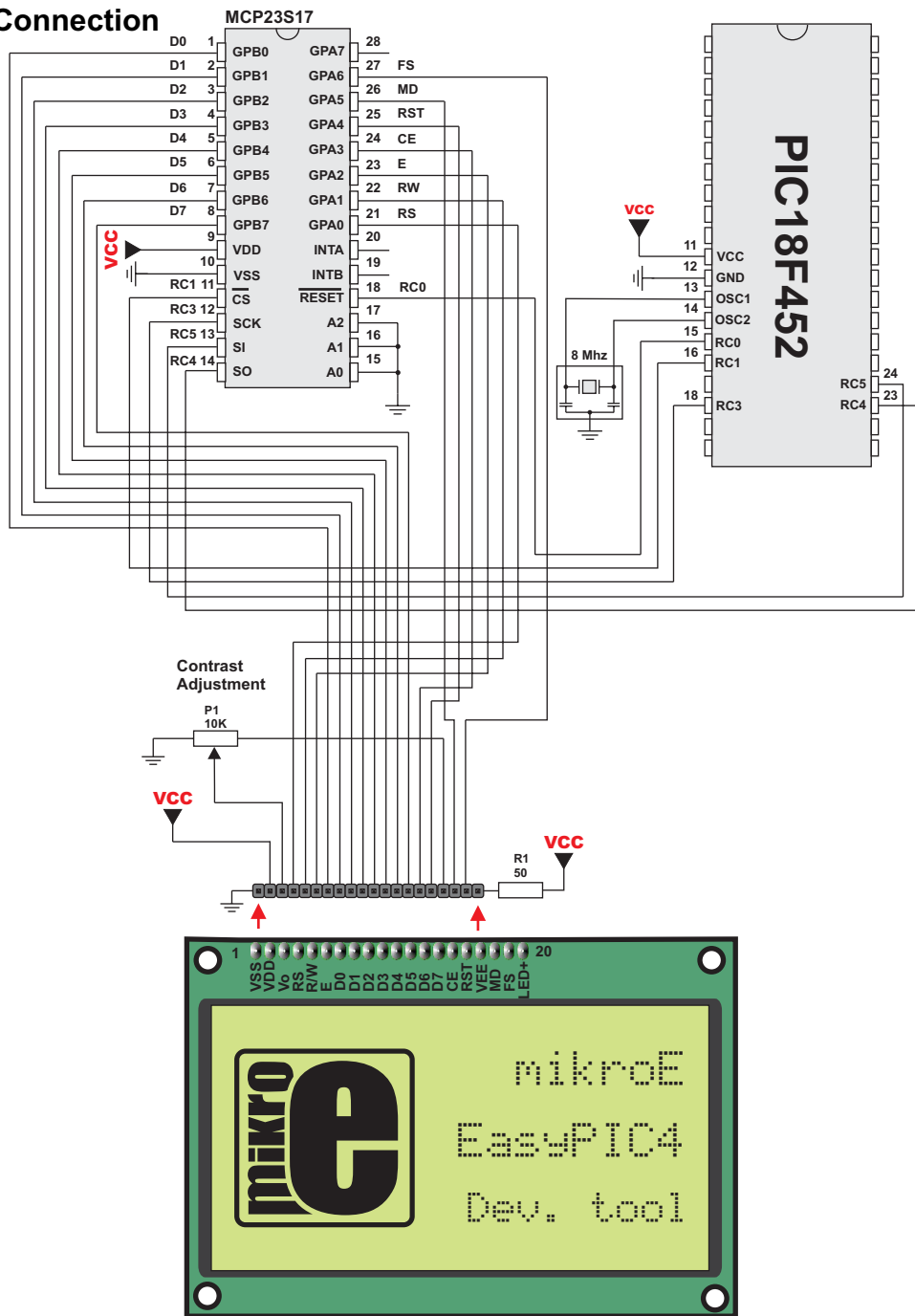
            Spi_T6963C_cursor_blink(0) ;
            break ;
    }
    Delay_ms(300) ;
}

/*
 * 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) ;
}
}

```

Hardware Connection



Toshiba T6963C Graphic LCD (240x128)

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. Due to `pic16` family specific of not being able to read/write stack pointer, the program execution after `Longjmp` invocation 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	<code>int setjmp(jmp_buf env);</code>
Returns	if the return is from direct invocation it returns 0 if the return is from a call to the <code>longjmp</code> it returns nonzero value
Description	This function saves calling position in <code>jmp_buf</code> for later use by <code>longjmp</code> . The parameter <code>env</code> : array of type <code>(jmp_buf)</code> suitable for holding the information needed for restoring calling environment.
Requires	Nothing.
Example	<code>setjmp(buf);</code>

Longjmp

Prototype	<code>void longjmp(jmp_buf env, int val);</code>
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 containing 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	<code>longjmp(buf, 2);</code>

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. This example can be found in mikroC Setjmp example folder.

Time Library

The Time Library contains functions and type definitions for time calculations in UNIX time format. UNIX time counts the number of seconds since an "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, within the accuracy of the local clock.

What is the epoch?

Originally it was defined as the beginning of 1970 GMT. (Jan. 1st 1970 Julian day) GMT, Greenwich Mean Time, is a traditional term for the time zone in England.

The type declared is TimeStruct which is an structure type suitable for time and date storage. Type declaration is contained in timelib.h which can be found in mikroC Time Library Demo example folder.

Library Routines

```
Time_dateToEpoch
Time_epochToDate
Time_dateDiff
```

Time_dateToEpoch

Prototype	<code>long Time_dateToEpoch(TimeStruct *ts);</code>
Returns	This function returns number of seconds.
Description	This function returns the unix epoch : number of seconds since Jan. 1st 1970 0h00mn00s of the time struct pointed by ts.
Requires	Nothing.
Example	<code>Time_dateToEpoch(&ts1);</code>

Time_epochToDate

Prototype	<code>void Time_epochToDate(long e, TimeStruct *ts);</code>
Returns	This function returns number of seconds since 1970.
Description	Convert the unix epoch e (number of seconds since 1970) into a time struct ts.
Requires	Nothing.
Example	<code>Time_epochToDate(epoch, &ts2);</code>

Time_dateDiff

Prototype	<code>long Time_dateDiff(TimeStruct *t1, TimeStruct *t2);</code>
Returns	This function returns return time difference in seconds as a signed long.
Description	This function compares two dates, 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.
Requires	Note: This function is implemented as macro in timelib.h header file which can be found in mikroC Time Library Demo example folder.
Example	<code>diff = Time_dateDiff(&ts1, &ts2);</code>

Library Example

Example demonstrates TimeLibraryDemo (simplified c-like time library for PIC MCU).

```
#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) ;
}
```

Contact us:

If you are experiencing problems with any of our products or you just want additional information, please let us know.

Technical Support for compiler

If you are experiencing any trouble with mikroC, please do not hesitate to contact us - it is in our mutual interest to solve these issues.

Discount for schools and universities

mikroElektronika offers a special discount for educational institutions. If you would like to purchase mikroC for purely educational purposes, please contact us.

Problems with transport or delivery

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

Would you like to become mikroElektronika's distributor?

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

Other

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

mikroElektronika
Admirala Geprata 1B
11000 Belgrade
EUROPE

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

Fax: + 381 (11) 30 66 379

E-mail: office@mikroe.com

Website: www.mikroe.com