manual

C COMPILER FOR MICROCHIP PIC MICROCONTROLLERS

mikroC

Making it simple

User's manual

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 mikroEelktronika. The PDF-edition of the manual can be printed for private or local use, but not for distribution. Modifying the manual or the compiler is strictly prohibited.

HIGH RISK ACTIVITIES

The mikroC 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 - 2005.

This manual covers mikroC version 2.1 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@mikroelektronika.co.yu. 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.mikroelektronika.co.yu
E-mail: office@mikroelektronika.co.yu

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.
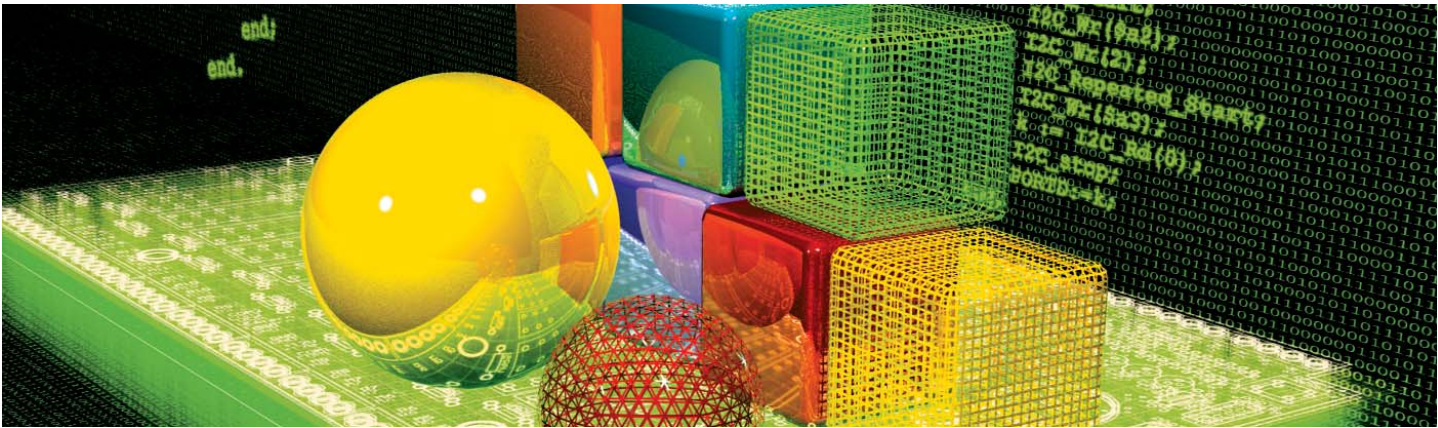
mikroC User's manual

Table of Contents

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.

Watch Window

Code Explorer

Code Editor

Error Window

Code Assistant

Breakpoints Window
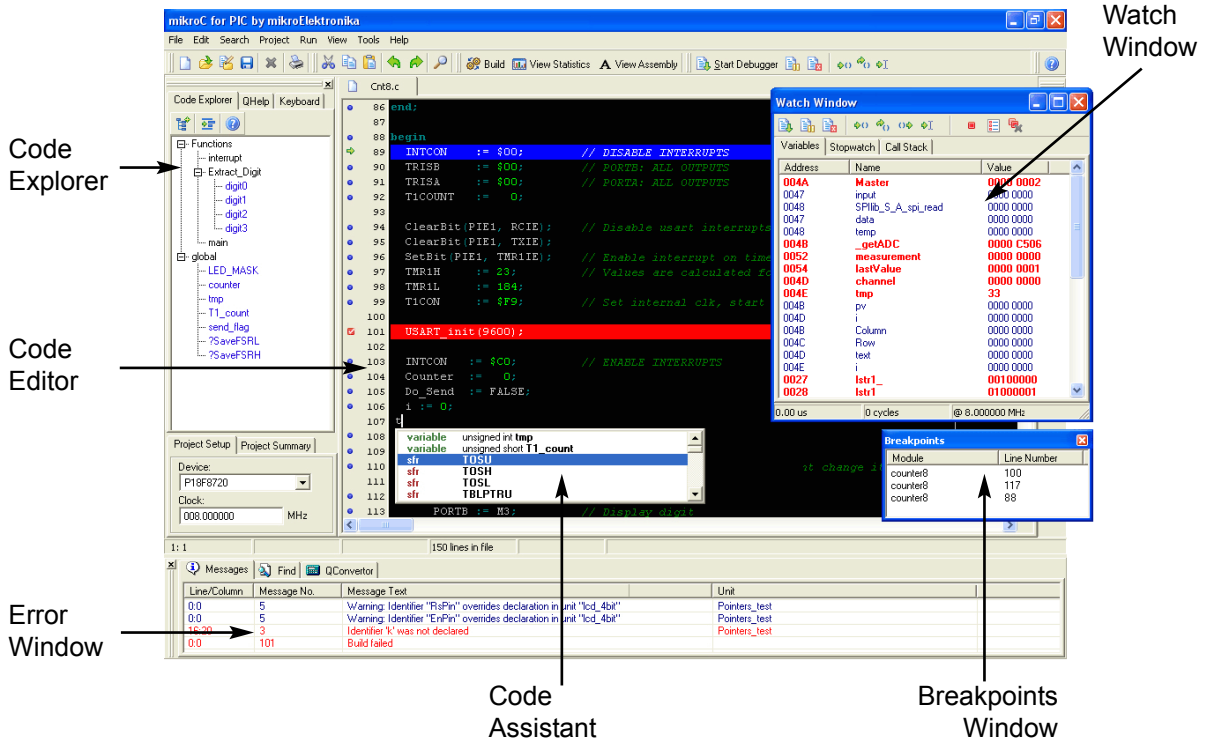
mikroC allows you to quickly develop and deploy complex applications:

- 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.
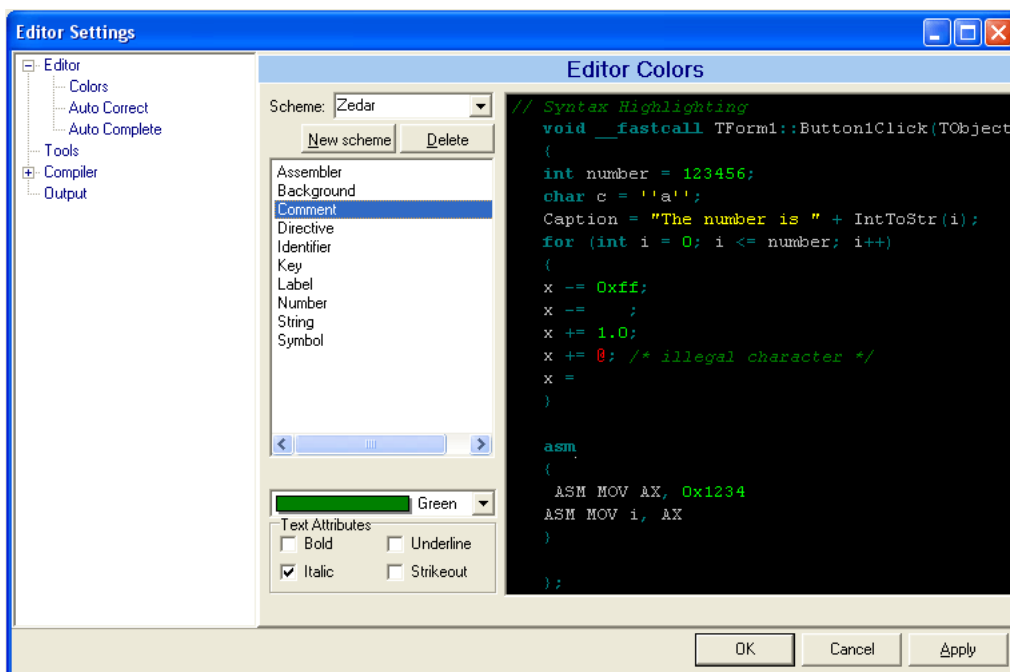
Code Assistant [CTRL+SPACE]

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.
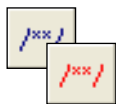
Comment / Uncomment Icon.

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.
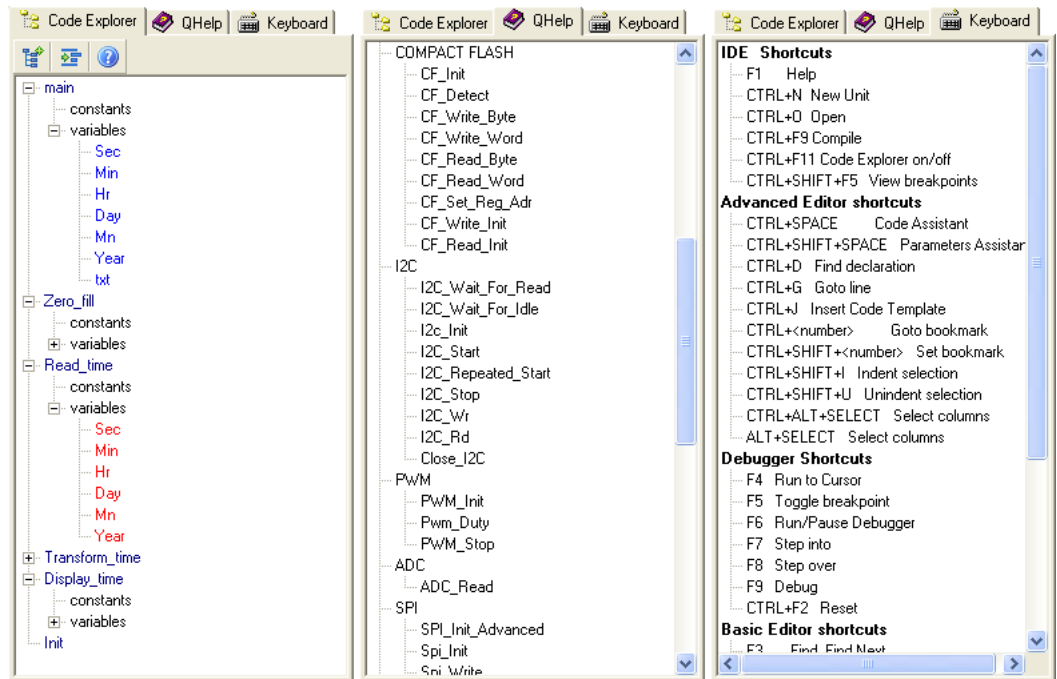
DEBUGGER

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 pro-
gram. It displays variables and special function registers of PIC MCU, their
addresses and values. Values are updated as you go through the simulation.
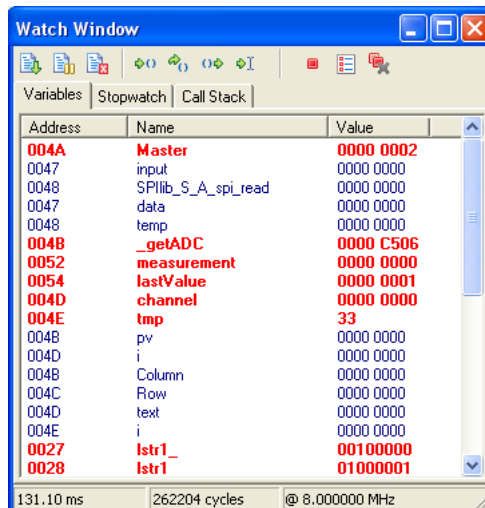
Double clicking one of the items opens a window in which you can assign a new
value to the selected variable or register and change number formatting.

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.
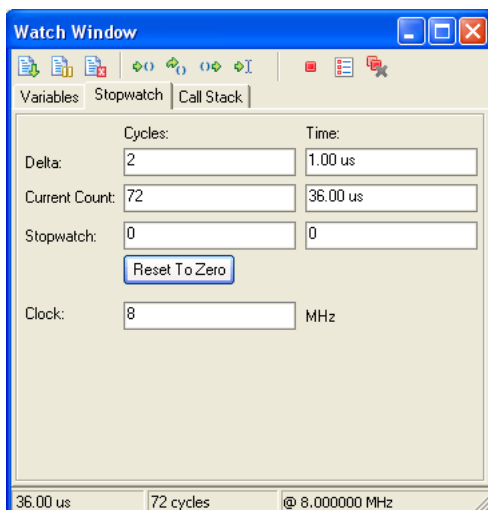
Call Stack Window

The Call Stack Window keeps track of depth and order of nested routine calls in program simulation. Check the Nested Calls Limitations for more information.
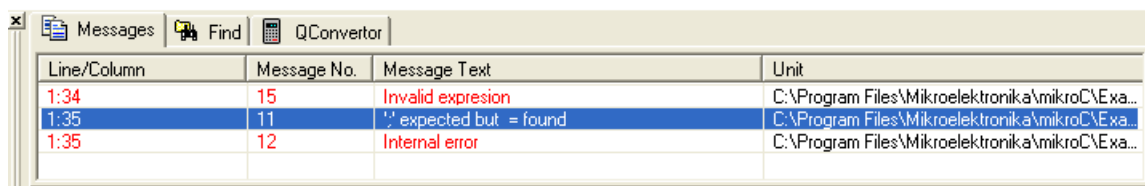
Note: Real scenarios may differ from the simulation, depending on runtime program parameters.
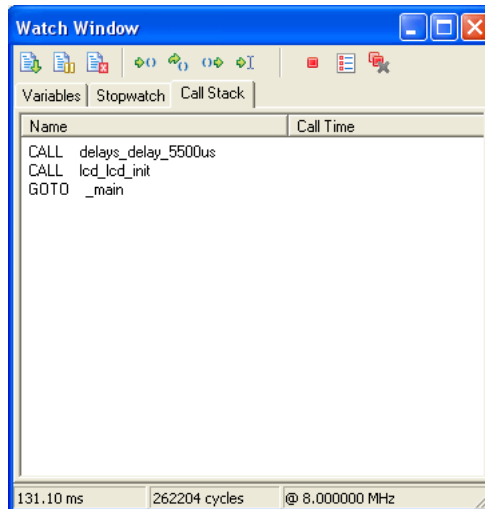
ERROR WINDOW

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 interefere 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.
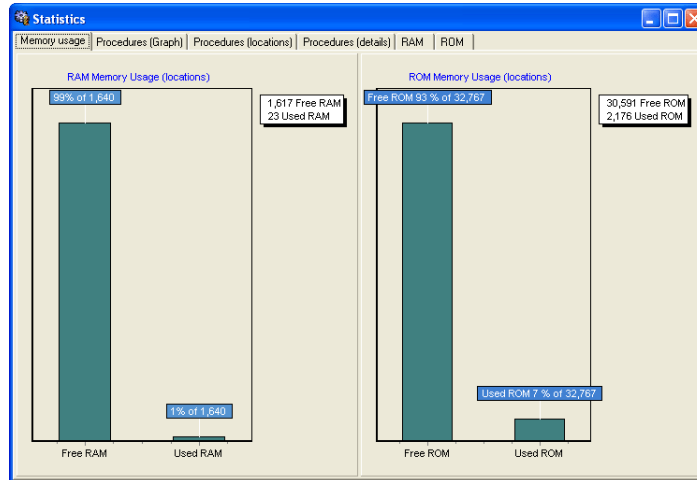
Procedures (Graph) Window
Displays functions in form of histogram, according to their memory allotment.
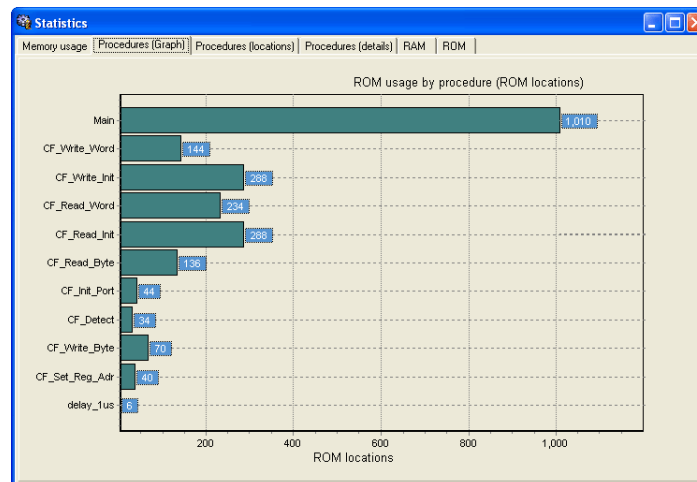
Procedures (Locations) Window

Displays how functions are distributed in microcontroller's memory.
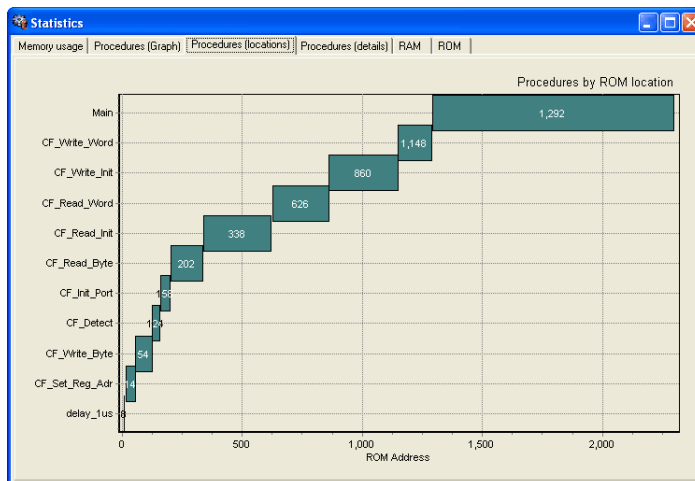
Procedures (Details) Window

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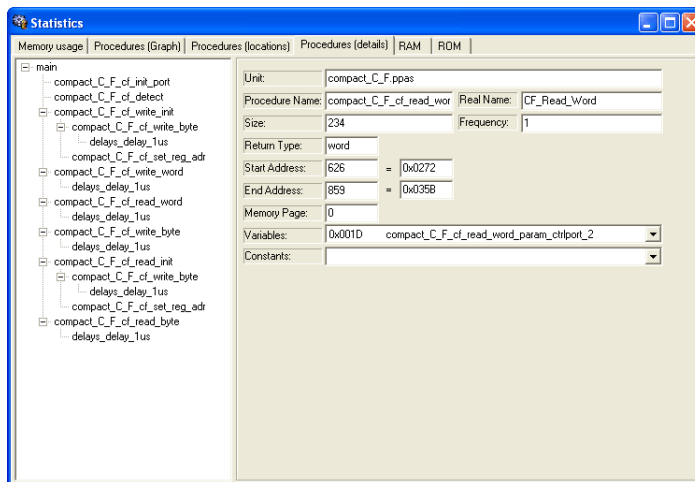
size, start and end address, calling frequency, return type, etc.

RAM Window
Summarizes all GPR and SFR registers and their addresses. Also displays symbolic names of variables and their addresses.
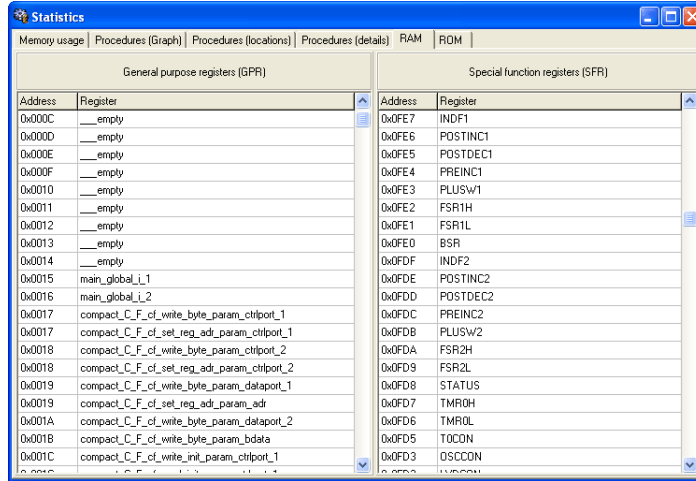
ROM Window
Lists op-codes and their addresses in form of a human readable hex code.
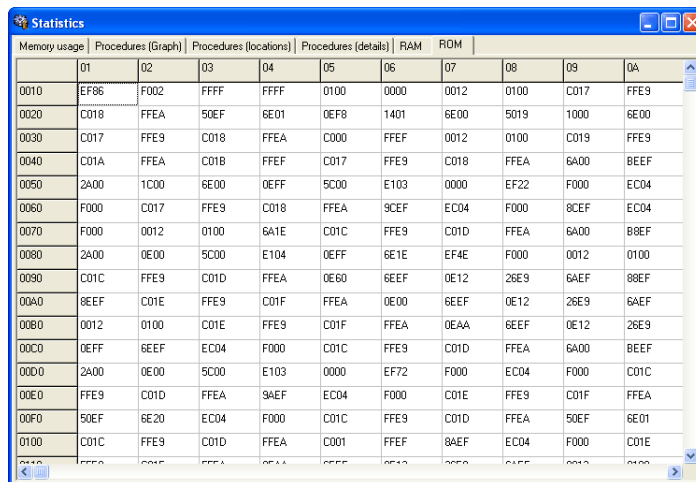
INTEGRATED TOOLS

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.
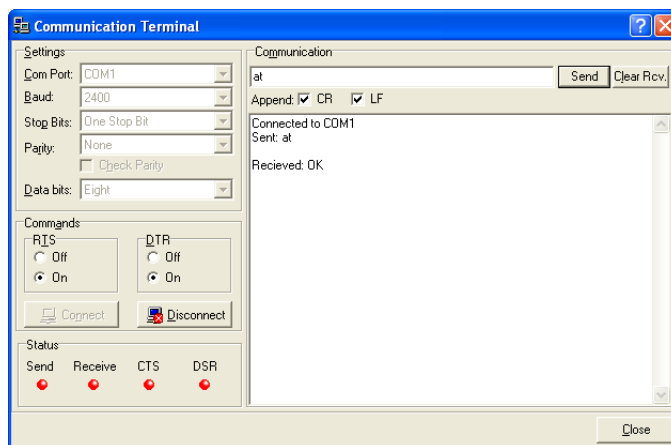
ASCII Chart

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.

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.
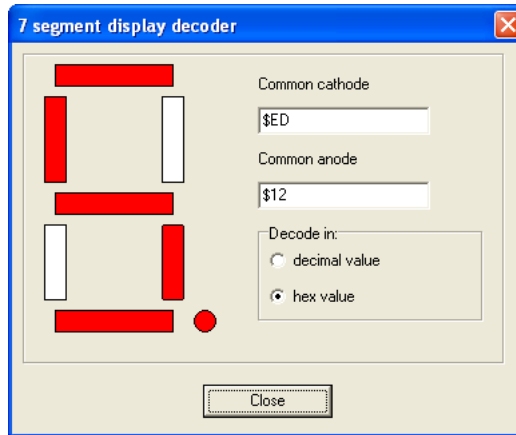
EEPROM Editor

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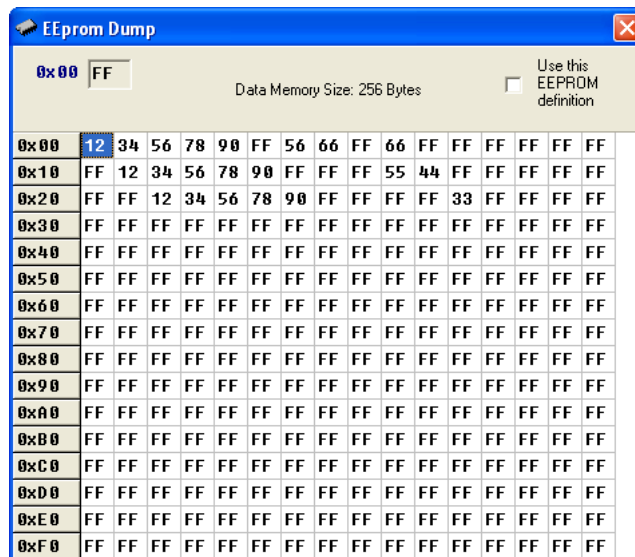
mikroBootloader

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 > Bootoader.
3. Click on Setup Port and select the COM port that will be used. Make sure that BAUD is set to 9600 Kpbs.
4. Click on Open File and select the HEX file you would like to upload.
5. Since the bootcode in the PIC only gives the computer 4-5 sec to connect, you should reset the PIC and then click on the Connect button within 4-5 seconds.
6. The last line in then history window should now read "Connected".
7. To start the upload, just click on the Start Bootloader button.
8. Your program will written to the PIC flash. Bootloader will report an errors that may occur.
9. Reset your PIC and start to execute.

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
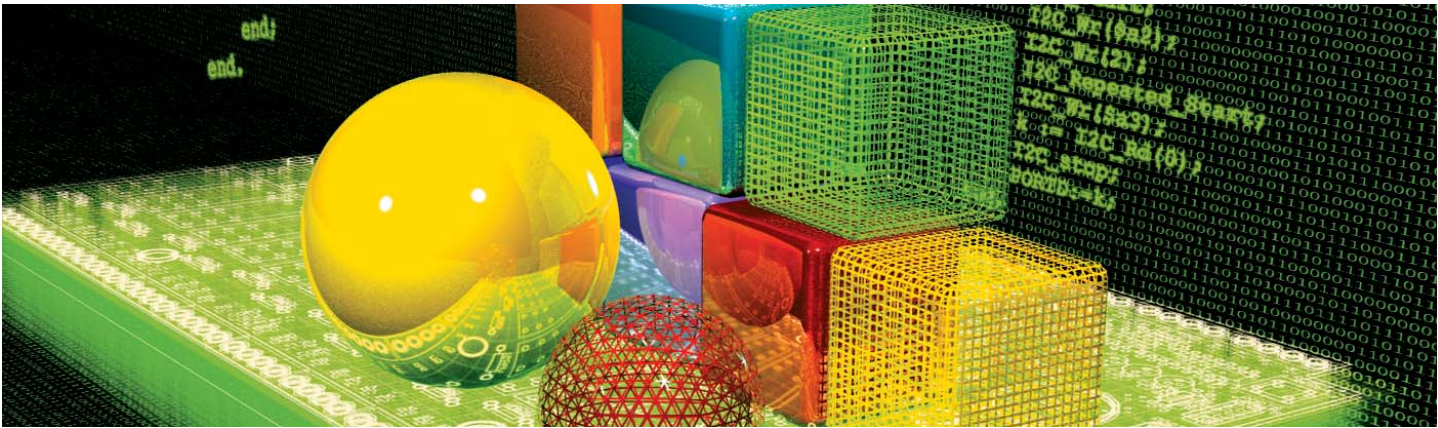
Building Applications

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.

Remove from 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.

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

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 pre-processor 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.

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.

Managing Source Files

Creating a new source file

New 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.

Opening an Existing File

Open File Icon.

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.

Printing an Open File

Print File Icon.

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.

Saving File

Save File Icon.

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.

Saving File Under a Different Name

Save File As.

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.

Closing a File

Close 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 then 0
- Local objects cannot be extern
- Declarator error
- Bad storage class
- Arguments cannot be of void type
- Specifer/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 expresion
- 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 inefficent code: func call in complex expression
- Inefficent code: func call in complex expression
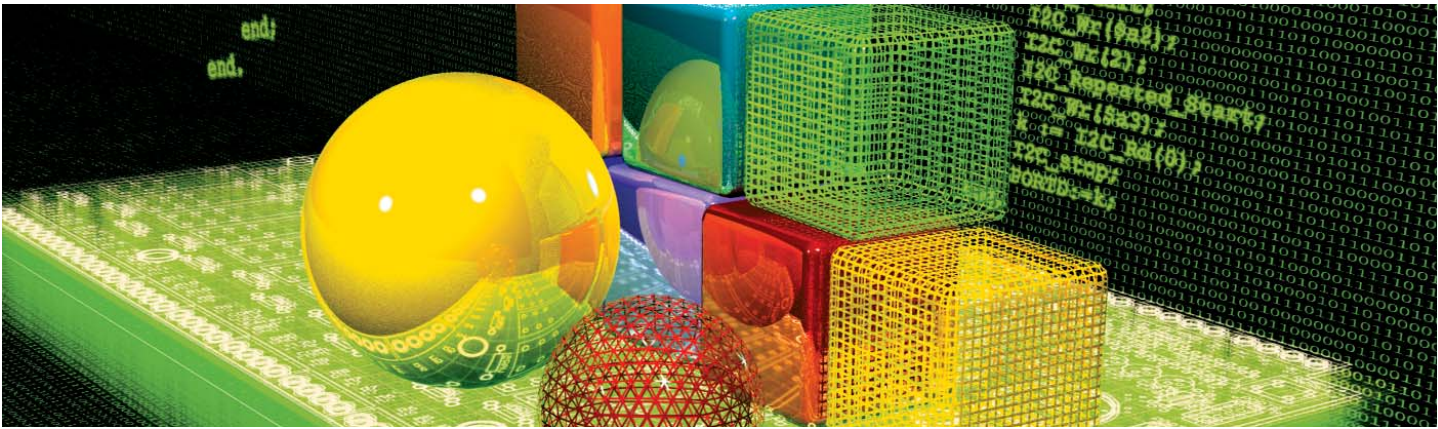
mikroC Language Reference

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 calls, as form of cross-calling, are unsupported by mikroC 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.

The number of allowed nested calls decreases by one if you use any of the following operators in the code: `*  /  %`. It further decreases by one if you use interrupt in the program. If the allowed number of nested calls is exceeded, compiler will report 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 intented to facilitate PIC programming, while others are result of PICmicro hardware limitations:

Function cross-calling and recursion are unsupported due to the PIC's limitations of no easily-usable stack and limited memory.

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

Function calls from within interrupts are a special case. See Interrupts.

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.

Tags scope is specific. Due to separate name space, tags are virtually removed from normal scope rules: they have file scope, but override any block rules.

Ellipsis (...) in formal argument lists is unsupported.

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

Features under construction: pointers to functions, and anonymous structures.

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 `T0IE`, `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).

Device Clock Constants

There are two built-in constants related to device clock: `___FOSC` and `___FCY`. Constant `___FOSC` equals the frequency that is provided by an external oscillator, while `___FCY` is the operating frequency of PIC. Both constants can be used anywhere in the code, and are automatically updated as you change target PIC in your project. Source files that use these constants are recompiled any time the clock is changed in IDE.

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 access bits by their name:

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

Interrupts

Interrupts can be easily handled by means of reserved word interrupt. mikroC implictly 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

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

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

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

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

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

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

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;
}
```

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;
```

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 an hexadecimal constant is derived from its value, according to the rules presented above. For example, `0xC367` will be treated as `unsigned int`.

Binary Constants

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

Octal Constants

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

Floating Point Constants

A floating-point constant consists of:

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

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 be 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:

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

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:

```c
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:

```c
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:

```c
#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 semi-colon is not required after the }, except in structure declarations. Often, the semi-colon 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) evalutates 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:

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

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

```c
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 definiton of the word would encompass functions, variables, symbolic constants, user-defined data types, and labels.

Each value has an associated name and type (also known as a data type). The name is used to access the object. 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.

Tag Scope
Structure, union, and enumeration tags are somewhat specific in mikroC. Due to separate name space, tags are virtually removed from normal scope rules: they have file scope, but override any block rules. Thus, deeply nested declaration of structure is identical to an equivalent global declaration. As a consequence, once that you have defined a tag, you cannot redefine it in any block within file.

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:

```c
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:

```c
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 *fudamental* 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-ly.

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	±1.17549435082E-38 .. ±6.80564774407E38
double	32-bit	±1.17549435082E-38 .. ±6.80564774407E38
long double	32-bit	±1.17549435082E-38 .. ±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 identifer:

```
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 acording 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 ) { ... }
```

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*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 = &[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 complexitiy 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.

mikroC implementation of bit fields requires you to set aside a structure for the purpose, i.e. you cannot have a structure containing bit fields and other objects. Bit fields structure can contain up to 8 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 identifer requires a colon and its width in bits to be explicitly specified. Total width of all components cannot exceed one byte (8 bits).

As an object, bit fields structure takes one byte. Individual fields are packed within byte 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 intializing an object (in declaration) with an object of different type.

In these situations, compiler will provide an automatic implicit conversion of types, without any 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 initializator 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` modifer 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 indepent 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.

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 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.

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 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) >= (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
&	bitwise AND; returns 1 if both bits are 1, otherwise returns 0	9
|	bitwise (inclusive) OR; returns 1 if either or both bits are 1, otherwise returns 0	9
^	bitwise exclusive OR (XOR); returns 1 if the bits are complementary, otherwise 0	10
~	bitwise complement (unary); inverts each bit	10
>>	bitwise shift left; moves the bits to the left, see below	10
<<	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" bytes on the right are assigned zeros. Thus, shifting unsigned operand to left by n positions is equivalent to multiplying it by 2n 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" bytes on the left are assigned zeros (in case of unsigned operand) or the value of the sign bit zeros (in case of signed operand). Shifting operand to right by n positions is equivalent to dividing it by 2n.

```
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
&&	logical AND	5
||	logical OR	4
!	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 comlex 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
   intializes a, divides b by 2, and increments c */

result = (x = 10, y = x + 3, x--, z -= x * 3 - --y);
/* returns computed value of variable z,
   and also computes x and y */
```

Note

Do not confuse comma operator (sequence operator) with the comma punctuator which separates elements in a function argument list and initializator 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 `if` to implement a conditional statement. Syntax of `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 specifed 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 controled by means of preprocessor directives and pre-processor 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 (l/**/r).

Conditional Compilation

Conditional compilation directives are typically used to make source programs easy to change and easy to compile in different execution environments. 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.
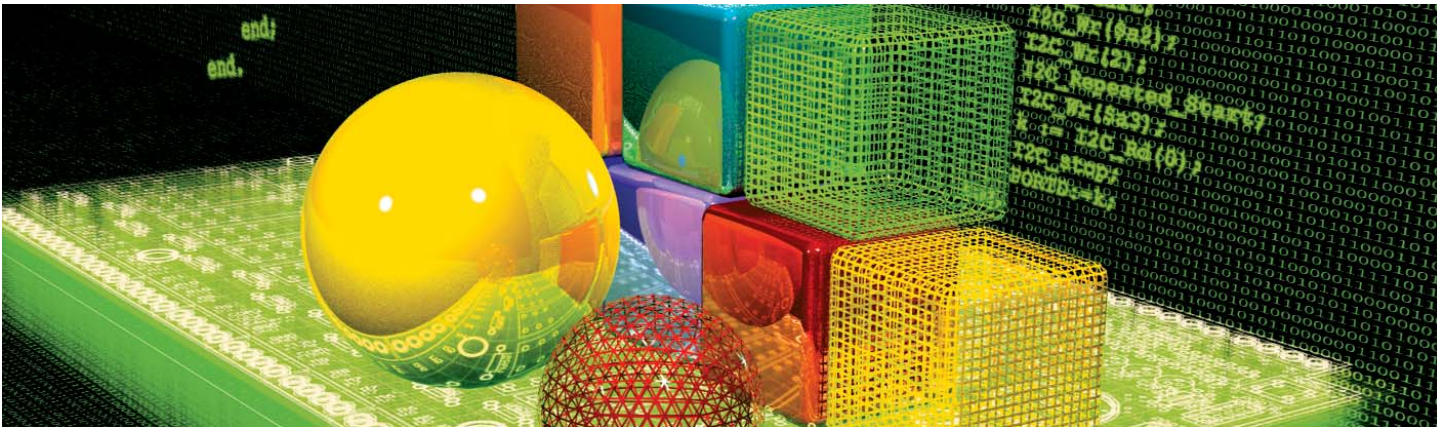
mikroC Libraries

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, 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.

Currently, mikroC includes following built-in functions:

```
Delay_us
Delay_ms
Delay_Cyc
Clock_Khz
```

Delay_us

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

Delay_ms

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

Vdelay_ms

Prototype	`void Vdelay_ms(unsigned time_in_ms);`
Description	Creates a software delay in duration of `time_in_ms` milliseconds (a variable). Generated delay is not as precise as the delay created by `Delay_ms`.

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

Delay_Cyc

Prototype	`void Delay_Cyc(char Cycles_div_by_10);`
Description	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. Note that `Delay_Cyc` is library function rather than a built-in routine; it is presented in this topic for the sake of convenience.
Example	`Delay_Cyc(10);  /* Hundred MCU cycles pause */`

Clock_Khz

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

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:

- ADC Library
- CAN Library
- CANSPI Library
- Compact Flash Library
- Conversions Library
- EEPROM Library
- Ethernet Library
- Flash Memory Library
- Graphic LCD Library
- I2C Library
- Keypad Library
- LCD Library
- LCD8 Library
- Manchester Code Library
- Multi Media Card Library
- OneWire Library
- PS/2 Library
- PWM Library
- RS-485 Library
- Secure Digital Library
- Software I2C Library
- Software SPI Library
- Software UART Library
- Sound Library
- USART Library
- USB HID Library
- Util Library
- ANSI C Standard Libraries
- Conversions Library
- Trigonometry Library

ADC Library

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

Adc_Read

Prototype	`unsigned Adc_Read(char channel);`
Returns	10-bit unsigned value read from the specified ADC channel.
Description	Initializes PIC's internal ADC module to work with RC clock. Clock determines the time period necessary for performing AD conversion (min 12TAD). Parameter `channel` represents the channel from which the analog value is to be acquired. For channel-to-pin mapping please refer to documentation for the appropriate PIC MCU.
Requires	PIC MCU with built-in ADC module. You should consult the Datasheet documentation for specific device (most devices from PIC16/18 families have it). 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). The function is currently unsupported by the following PICmicros: P18F2331, P18F2431, P18F4331, and P18F4431.

Example	`unsigned tmp;`
	`...`
	`tmp = Adc_Read(1);  /* read analog value from channel 1 */`

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
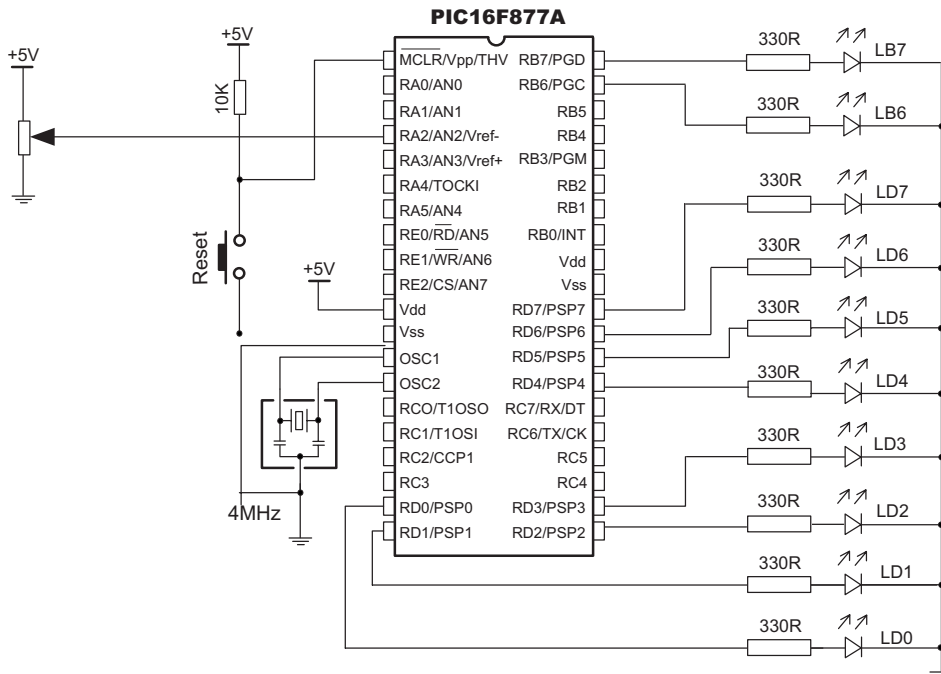
CAN Library

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	`void CANSetOperationMode(char mode, char wait_flag);`
Description	Sets CAN to requested mode, i.e. copies mode to CANSTAT. Parameter `mode` needs to be one of CAN_OP_MODE constants (see CAN constants). Parameter `wait_flag` needs to be either 0 or 0xFF: If set to 0xFF, this is a blocking call – the function won't "return" until the requested mode is set. If 0, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use function `CANGetOperationMode` to verify correct operation mode before performing mode specific operation.
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	`CANSetOperationMode(CAN_MODE_CONFIG, 0xFF);`

CANGetOperationMode

Prototype	`char CANGetOperationMode(void);`
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	`if (CANGetOperationMode() == CAN_MODE_NORMAL) { ... };`

CANInitialize

Prototype	`void CANInitialize(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);`

Description	Initializes CAN. All pending transmissions are aborted. Sets all mask registers to 0 to allow all messages.
	
	Filter registers are set according to flag value:
	
	`if (CAN_CONFIG_FLAGS & CAN_CONFIG_VALID_XTD_MSG != 0)`
	`  // Set all filters to XTD_MSG`
	`else if (config & CONFIG_VALID_STD_MSG != 0)`
	`  // Set all filters to STD_MSG`
	`else`
	`  // Set half the filters to STD, and the rest to XTD_MSG`
	
	Parameters:
	
	`SJW` as defined in 18XXX8 datasheet (1–4)
	`BRP` as defined in 18XXX8 datasheet (1–64)
	`PHSEG1` as defined in 18XXX8 datasheet (1–8)
	`PHSEG2` as defined in 18XXX8 datasheet (1–8)
	`PROPSEG` as defined in 18XXX8 datasheet (1–8)
	`CAN_CONFIG_FLAGS` is formed from predefined constants (see CAN constants).

Requires	CAN must be in Config mode; otherwise the function will be ignored.

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`

CANSetBaudRate

Prototype	`void CANSetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);`
Description	Sets CAN baud rate. Due to complexity of CAN protocol, you cannot simply force a bps value. Instead, use this function when CAN is in Config mode. Refer to datasheet for details. Parameters: `SJW` as defined in 18XXX8 datasheet (1–4) `BRP` as defined in 18XXX8 datasheet (1–64) `PHSEG1` as defined in 18XXX8 datasheet (1–8) `PHSEG2` as defined in 18XXX8 datasheet (1–8) `PROPSEG` as defined in 18XXX8 datasheet (1–8) `CAN_CONFIG_FLAGS` is formed from predefined constants (see CAN constants)
Requires	CAN must be in Config mode; otherwise the function will be ignored.

Example	`init = CAN_CONFIG_SAMPLE_THRICE   &`
	`       CAN_CONFIG_PHSEG2_PRG_ON   &`
	`       CAN_CONFIG_STD_MSG         &`
	`       CAN_CONFIG_DBL_BUFFER_ON   &`
	`       CAN_CONFIG_VALID_XTD_MSG   &`
	`       CAN_CONFIG_LINE_FILTER_OFF;`
	`...`
	`CANSetBaudRate(1, 1, 3, 3, 1, init);`

CANSetMask

Prototype	`void CANSetMask(char CAN_MASK, long value, char CAN_CONFIG_FLAGS);`
Description	Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers. Parameters: CAN_MASK is one of predefined constant values (see CAN constants); value is the mask register value; CAN_CONFIG_FLAGS selects type of message to filter, either CAN_CONFIG_XTD_MSG or CAN_CONFIG_STD_MSG.
Requires	CAN must be in Config mode; otherwise the function will be ignored.

Example	`// Set all mask bits to 1, i.e. all filtered bits are relevant:`
	
	`CANSetMask(CAN_MASK_B1, -1, CAN_CONFIG_XTD_MSG);`
	
	`/* Note that -1 is just a cheaper way to write 0xFFFFFFFF.`
	`   Complement will do the trick and fill it up with ones. */`

CANSetFilter

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

CANRead

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

CANWrite

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

Example	`char tx, data; long id;`
	`tx = CAN_TX_PRIORITY_0 & CAN_TX_XTD_FRAME;`
	`CANWrite(id, data, 2, tx);`

CAN Constants

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

CAN_OP_MODE

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

```
#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

```c
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 ..
```

```c
// .. 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
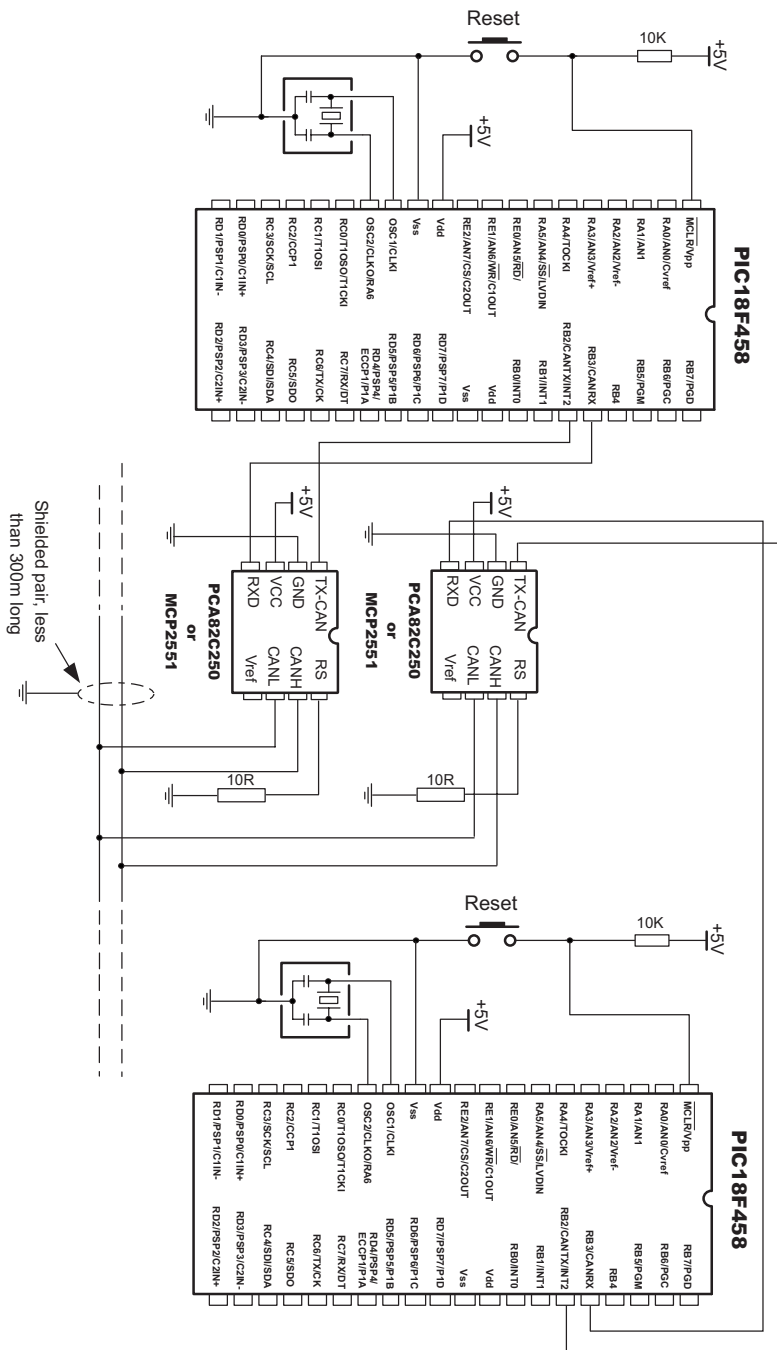
CANSPI Library

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.

Library Routines

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

Following routines are for the internal use by compiler only:

```
RegsToCANSPIID
CANSPIIDToRegs
```

CANSPISetOperationMode

Prototype	`void CANSPISetOperationMode(char mode, char wait_flag);`
Description	Sets CAN to requested mode, i.e. copies mode to CANSTAT. Parameter `mode` needs to be one of CAN_OP_MODE constants (see CAN constants, page 145). Parameter `wait_flag` needs to be either 0 or 0xFF: If set to 0xFF, this is a blocking call – the function won't "return" until the requested mode is set. If 0, this is a non-blocking call. It does not verify if CAN module is switched to requested mode or not. Caller must use function `CANSPIGetOperationMode` to verify correct operation mode before performing mode specific operation.
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	`CANSPISetOperationMode(CAN_MODE_CONFIG, 0xFF);`

CANSPIGetOperationMode

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

CANSPIInitialize

Prototype	```void CANSPIInitialize(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);```

Description	Initializes CANSPI. All pending transmissions are aborted. Sets all mask registers to 0 to allow all messages.
	
	Filter registers are set according to flag value:
	
	```if (CAN_CONFIG_FLAGS & CAN_CONFIG_VALID_XTD_MSG != 0)```
	``` // Set all filters to XTD_MSG```
	```else if (config & CONFIG_VALID_STD_MSG != 0)```
	``` // Set all filters to STD_MSG```
	```else```
	``` // Set half the filters to STD, and the rest to XTD_MSG```
	
	Parameters:
	
	SJW as defined in 18XXX8 datasheet (1–4)
	BRP as defined in 18XXX8 datasheet (1–64)
	PHSEG1 as defined in 18XXX8 datasheet (1–8)
	PHSEG2 as defined in 18XXX8 datasheet (1–8)
	PROPSEG as defined in 18XXX8 datasheet (1–8)
	CAN_CONFIG_FLAGS is formed from predefined constants (see CAN constants, page 145).

Requires	CANSPI must be in Config mode; otherwise the function will be ignored.

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;```
	```...```
	```CANSPIInitialize(1, 1, 3, 3, 1, init); // initialize CANSPI```

CANSPISetBaudRate

Prototype	`void CANSPISetBaudRate(char SJW, char BRP, char PHSEG1, char PHSEG2, char PROPSEG, char CAN_CONFIG_FLAGS);`
Description	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. Parameters: SJW as defined in 18XXX8 datasheet (1–4) BRP as defined in 18XXX8 datasheet (1–64) PHSEG1 as defined in 18XXX8 datasheet (1–8) PHSEG2 as defined in 18XXX8 datasheet (1–8) PROPSEG as defined in 18XXX8 datasheet (1–8) CAN_CONFIG_FLAGS is formed from predefined constants (see CAN constants)
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.

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;
	...
	CANSPISetBaudRate(1, 1, 3, 3, 1, init);
	``

CANSPISetMask

Prototype	`void CANSPISetMask(char CAN_MASK, long value, char CAN_CONFIG_FLAGS);`
Description	Function sets mask for advanced filtering of messages. Given value is bit adjusted to appropriate buffer mask registers. Parameters: `CAN_MASK` is one of predefined constant values (see CAN constants); `value` is the mask register value; `CAN_CONFIG_FLAGS` selects type of message to filter, either `CAN_CONFIG_XTD_MSG` or `CAN_CONFIG_STD_MSG`.
Requires	CANSPI must be in Config mode; otherwise the function will be ignored.

Example	`// 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. */`

CANSPISetFilter

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

CANSPIRead

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

CANSPIWrite

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

Example	`char tx, data; long id;`
	`tx = CAN_TX_PRIORITY_0 & CAN_TX_XTD_FRAME;`
	`CANSPIWrite(id, data, 2, tx);`

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.

```c
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 ..
```

```c
// .. continued

  // Initialize external CAN module
  CANSPIInitialize(1,1,3,3,1,aa);

  // 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

Shielded pair, less
than 300m long
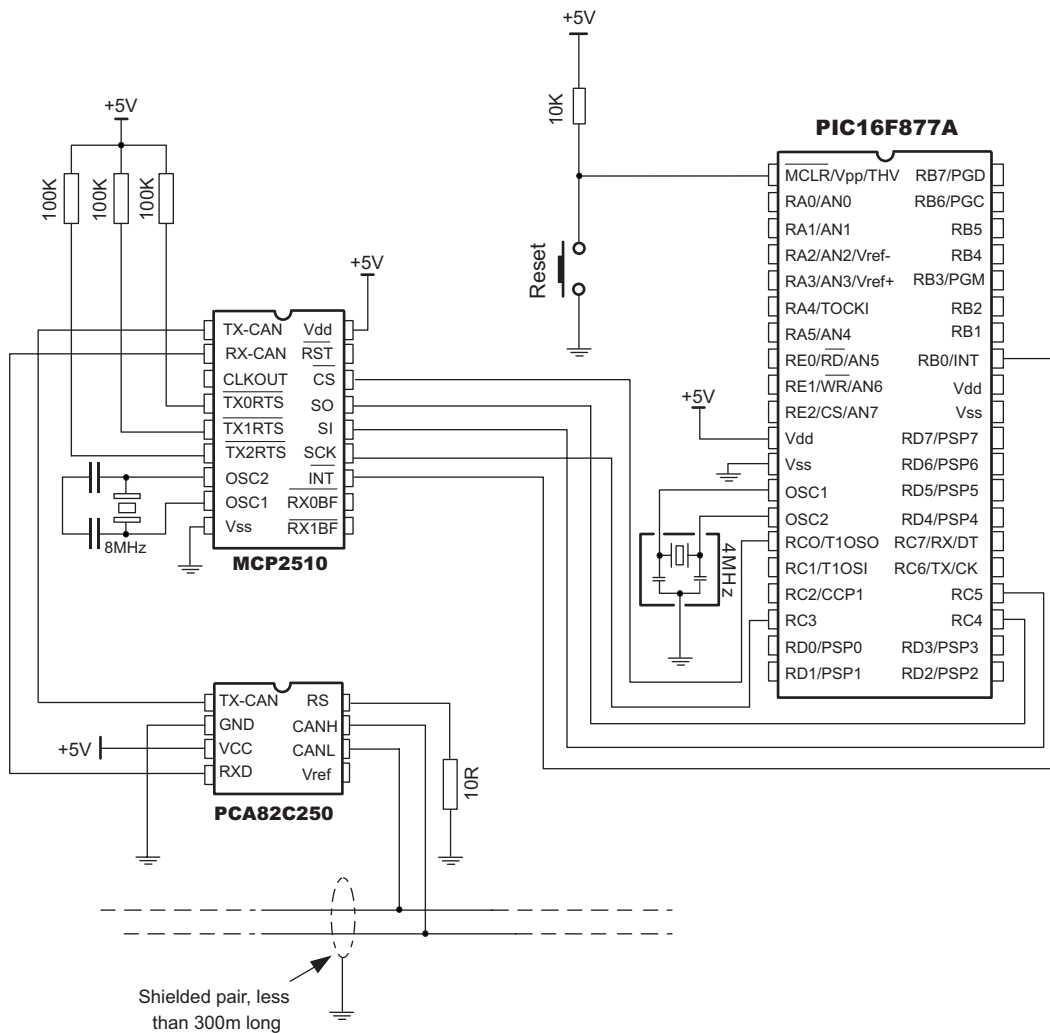
Compact Flash Library

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_Read_Word
Cf_Write_Init
Cf_Write_Byte
Cf_Write_Word

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
```

Function `Cf_Set_Reg_Adr` is for compiler internal purpose only.

Cf_Init

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

Cf_Detect

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

Cf_Total_Size

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

Cf_Enable

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

Cf_Disable

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

Cf_Read_Init

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

Cf_Read_Byte

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

Cf_Read_Word

Prototype	`unsigned Cf_Read_Word (void);`
Returns	Returns word (16-bit) from CF.
Description	Reads one word from CF.
Requires	CF must be initialized for read operation. See `Cf_Read_Init`.
Example	`PORTC = Cf_Read_Word();  // Read word and display it on PORTC`

Cf_Write_Init

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

Cf_Write_Byte

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

Cf_Write_Word

Prototype	`void Cf_Write_Word(int data);`
Description	Writes one word (`data`) to CF. All 512 bytes are transferred to a buffer.
Requires	CF must be initialized for write operation. See `Cf_Write_Init`.
Example	`Cf_Write_Word(1000);`

Cf_Find_File

Prototype	`void Cf_Find_File(char find_first, char *file_name);`
Description	Routine looks for files on CF card. Parameter `find_first` 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 `file_name`. If no file is found, the string will be filled with zeroes.
Requires	Ports must be initialized. See `Cf_Init`.
Example	`Cf_Find_File(1, file);` `if (file[0] <> 0) { ... // if first file found, handle it`

Cf_File_Write_Init

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

Cf_File_Write_Byte

Prototype	`void Cf_File_Write_Byte(char data);`
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	`// Write 50,000 zeroes (bytes) to file:` `for (i = 0; i < 50000; i++) Cf_File_Write_Byte(48);`

Cf_Read_Sector

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

Cf_Write_Sector

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

Cf_Set_File_Date

Prototype	**void** Cf_Set_File_Date(**int** year, **char** month,day,hours,min,sec);
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	*// April 1st 2005, 18:07:00* Cf_Set_File_Date(2005,4,1,18,7,0);

Cf_File_Write_Complete

Prototype	**void** Cf_File_Write_Complete(**char** filename[8], **char** *extension);
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	Cf_File_Write_Complete("MY_FILE1","txt");

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.

```c
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);
  }
}
```

Next example waits until the CF card is inserted, and when plugged, it creates 5 text files on the card. Each file will be appended the same timestamp.

```c
unsigned short index;
unsigned i1;
char *fname, *ext;

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

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

  Delay_ms(50);                   // Wait until the card is stabilized
} //~

void main() {
  ext = "TXT";
  index = 0;                      // Index of file to be written

  while (index < 5) {
    PORTC = 0;
    Init();
    PORTC = index;

    Cf_File_Write_Init();         // Initialization for writing to new file

    i1 = 0;

    // Write 50,000 bytes to file
    while (i1 < 50000) {
      Cf_File_Write_Byte(48 + index);
      i1++;
    }

    fname = "MY_TEST1";           // Name must be 8 character long in uppercase
    fname[8] = 48 + index;        // Ensure that files have different name

    Cf_Set_File_Date(2005,1,1,0,0,0);     // Append a timestamp
    Cf_File_Write_Complete(fname, ext);   // Close the file

    index++;
  }
  PORTC = 0xFF;
} //~!
```

HW Connection
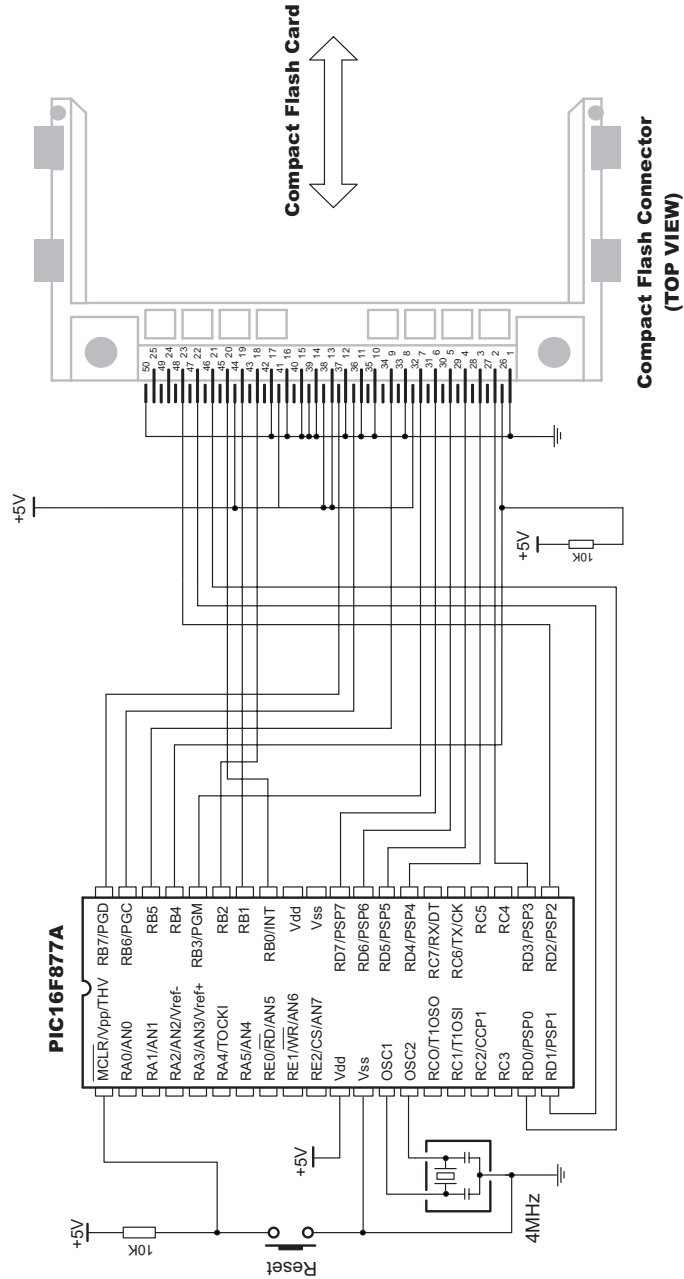
EEPROM Library

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	`char Eeprom_Read(char address);`
Returns	Returns byte from the specified address.
Description	Reads data from the specified address. Parameter `address` is of byte type, which means it can address only 256 locations. For PIC18 micros with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately.
Requires	Requires EEPROM module. Ensure minimum 20ms delay between successive use of routines `Eeprom_Write` and `Eeprom_Read`. Although PIC will write the correct value, `Eeprom_Read` might return an undefined result.

Example	`char take;`
	 `...`
	 `take = Eeprom_Read(0x3F);`

Eeprom_Read

Prototype	`void Eeprom_Write(char address, char data);`
Description	Writes data to the specified address. Parameter `address` is of byte type, which means it can address only 256 locations. For PIC18 micros with more EEPROM data locations, it is programmer's responsibility to set SFR EEADRH register appropriately. Be aware that all interrupts will be disabled during execution of EEPROM_Write routine (GIE bit of INTCON register will be cleared). Routine will set this bit on exit.
Requires	Requires EEPROM module. Ensure minimum 20ms delay between successive use of routines `Eeprom_Write` and `Eeprom_Read`. Although PIC will write the correct value, `Eeprom_Read` might return an undefined result.
Example	`Eeprom_Write(0x32);`

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	**void** Eth_Init(**char** *addrP, **char** *dataP, **char** *ctrlP, **char** pinReset, **char** pinIOW, **char** pinIOR);
Description	Performs initialization of Ethernet card and library. This includes: - 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. Parameter addrP is a pointer to address port, which handles the addressing lines. Parameter dataP is pointer to data port. Parameter ctrlP is the control port. Parameter pinReset is the reset/enable pin for the ethernet card chip (on control port). Parameter pinIOW is the I/O Write request control pin. Parameter pinIOR is the I/O read request control pin.
Requires	As specified for the entire library (please see top of this page).
Example	Eth_Init(&PORTB, &PORTD, &PORTE, 2, 1, 0);

Eth_Set_Ip_Address

Prototype	**void** Eth_Set_Ip_Address(**char** ip1, **char** ip2, **char** ip3, **char** ip4);
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 Eth_Init). You can change your IP address at any time, anywhere in the code.
Example	*// Set IP address 192.168.20.25* Eth_Set_Ip_Address(192u, 168u, 20u, 25u);

Eth_Set_Inport

Prototype	`unsigned short Eth_Inport(unsigned short address);`
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 `Eth_Init`.
Example	`udp_length |= Eth_Inport(NIC_DATA);`

Eth_Scan_For_Event

Prototype	`unsigned Eth_Scan_For_Event(unsigned short *next_ptr);`
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 `Eth_Init`. 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	`unsigned short Eth_Get_Ip_Hdr_Len(void);`
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 `Eth_Init`. The function must be called in a proper sequence, i.e. immediately after determining that the packet received is the IP packet.
Example	`// Receive IP Header` `opt_len = Eth_Get_Ip_Hdr_Len() - 20;`

Eth_Load_Ip_Packet

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

Eth_Get_Hdr_Chksum

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

Eth_Get_Source_Ip_Address

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

Eth_Get_Dest_Ip_Address

Prototype	`void Eth_Get_Dest_Ip_Address(void);`
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 `Eth_Init`. Also, a proper sequence of calls must be obeyed (see the `Ip_Event` function in the supplied Ethernet example).
Example	`Eth_Get_Dest_Ip_Address();`

Eth_Arp_Response

Prototype	`void Eth_Arp_Response(void);`
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	`Eth_Arp_Response();`

Eth_Get_Icmp_Info

Prototype	`void Eth_Get_Icmp_Info(void);`
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 `Eth_Init`. Also, this function must be called in a proper sequence, and before the `Eth_Ping_Response`.
Example	`Eth_Get_Icmp_Info();`

Eth_Ping_Response

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

Eth_Get_Udp_Source_Port

Prototype	`unsigned Eth_Get_Udp_Source_Port(void);`
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	`udp_source_port = Eth_Get_Udp_Source_Port();`

Eth_Get_Udp_Dest_Port

Prototype	`unsigned Eth_Get_Udp_Dest_Port(void);`
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 `Eth_Get_Udp_Source_Port` function.
Example	`udp_dest_port = Eth_Get_Udp_Dest_Port();`

Eth_Get_Udp_Port

Prototype	`unsigned short Eth_Get_Udp_Port(void);`
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 (`Eth_Set_Udp_Port`), 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 `Eth_Init`), and the UDP port propely set (see `Eth_Set_Udp_Port`). This library currently supports working with only one UDP port (socket) at a time.

Example	`if (udp_dest_port == Eth_Get_Udp_Port()) {`
	`    ... // Respond to action`
	`}`

Eth_Set_Udp_Port

Prototype	`void Eth_Set_Udp_Port(unsigned udp_port);`
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	`Eth_Set_Udp_Port(10001);`

Eth_Send_Udp

Prototype	`void Eth_Send_Udp(char *msg);`
Description	Sends the prepared UDP message (`msg`), of up to 16 bytes (characters). 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.
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	`Eth_Send_Udp(udp_tx_message);`

Eth_Load_Tcp_Header

Prototype	`void Eth_Load_Tcp_Header(void);`
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** | ```c
// 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)
  //...
}
``` |

Eth_Get_Tcp_Hdr_Offset

Prototype	`unsigned short Eth_Get_Tcp_Hdr_Offset(void);`
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 `Eth_Load_Tcp_Header`, since it initializes the private variables used for this function.
Example	`// calculate offset (TCP header length)` `tcp_options = Eth_Get_Tcp_Hdr_Offset() - 20;`

Eth_Get_Tcp_Flags

Prototype	`unsigned short Eth_Get_Tcp_Flags(void);`
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 (syncronize 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 `Eth_Load_Tcp_Header`, since it initializes the private variables used for this function.
Example	`flags = Eth_Get_Tcp_Flags();`

Eth_Set_Tcp_Data

Prototype	`void Eth_Set_Tcp_Data(const unsigned short *data);`
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	```// 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"
" Hello world! \n"
"</body>\n" "</html>";
//...
Eth_Set_Tcp_Data(httpPage1);
//...```

Eth_Tcp_Response

Prototype	`void Eth_Tcp_Response(void);`
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 `Eth_Set_Tcp_Data`.
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 `Eth_Set_Tcp_Data`.
Example	`Eth_Tcp_Response();`

Library Example

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

HW Connection

Flash Memory Library

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

Library Routines

```
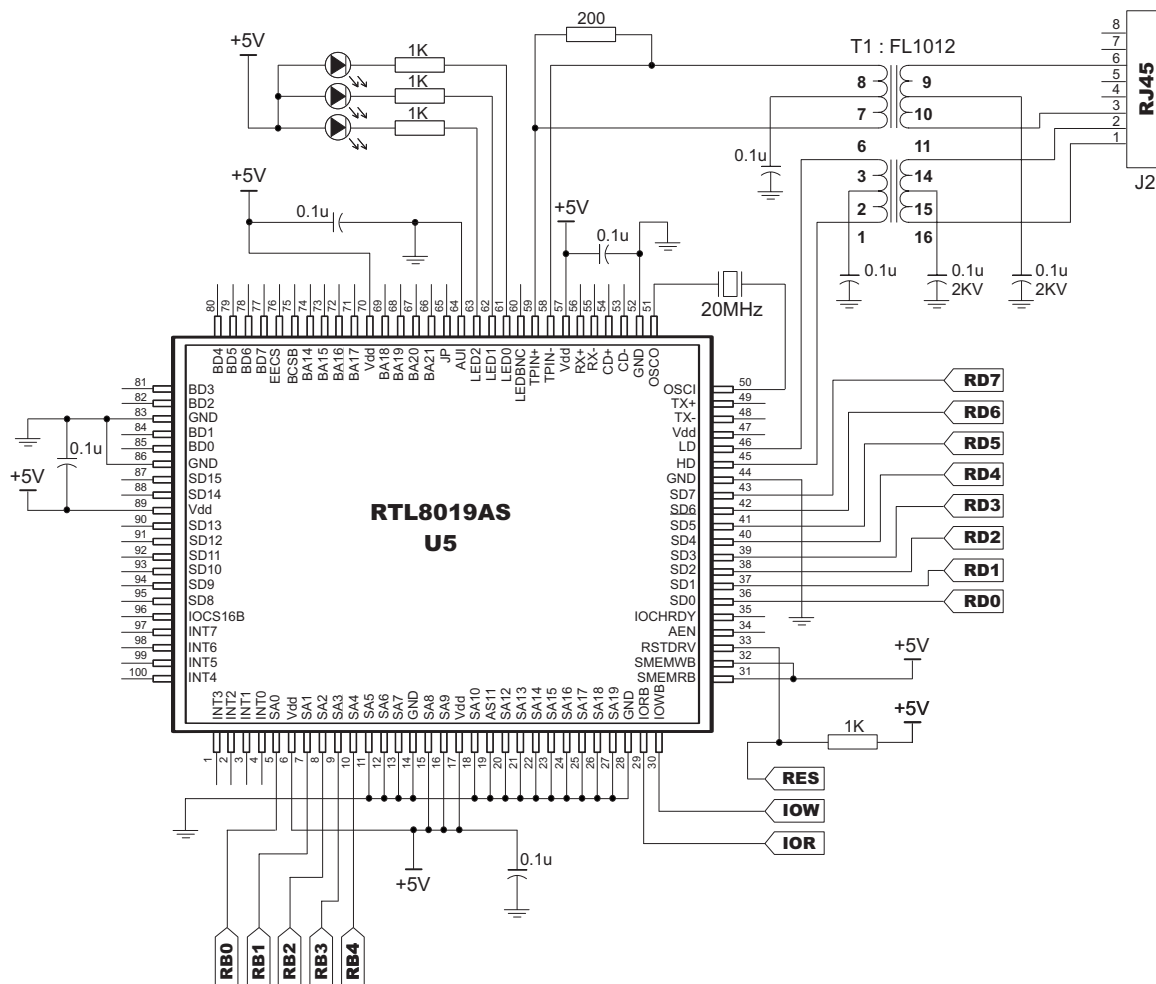Flash_Read
Flash_Write
```

Flash_Read

Prototype	`unsigned Flash_Read(unsigned address); // for PIC16`
	`char Flash_Read(long unsigned address); // for PIC18`

Returns	Returns data byte from Flash memory.
Description	Reads data from the specified address in Flash memory.
Example	`Flash_Read(0x0D00);`

Flash_Write

Prototype	`void Flash_Write(unsigned address, unsigned data);  // for PIC16`
	`void Flash_Write(unsigned long address, char *data); // for PIC18`

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	`// 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);`

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.

```c
char i = 0, j = 0;
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 I2C library which supports the master I²C mode.

Note: This library supports module on PORTB or PORTC, and will not work with modules on other ports. Examples for PICmicros with module on other ports can be found in your mikroC installation folder, subfolder "Examples".

Library Routines

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

I2C_Init

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

I2C_Start

Prototype	`char I2C_Start(void);`
Returns	If there is no error, function returns 0.
Description	Determines if I²C bus is free and issues START signal.
Requires	I²C must be configured before using this function. See `I2C_Init`.
Example	`I2C_Start();`

I2C_Repeated_Start

Prototype	`void I2C_Repeated_Start(void);`
Description	Issues repeated START signal.
Requires	I²C must be configured before using this function. See `I2C_Init`.
Example	`I2C_Repeated_Start();`

I2C_Is_Idle

Prototype	`char I2C_Is_Idle(void);`
Returns	Returns 1 if I²C bus is free, otherwise returns 0.
Description	Tests if I²C bus is free.
Requires	I²C must be configured before using this function. See `I2C_Init`.
Example	`if (I2C_Is_Idle()) {...}`

I2C_Rd

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

I2C_Wr

Prototype	`char I2C_Wr(char data);`
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter data) via I²C bus.
Requires	START signal needs to be issued in order to use this function. See `I2C_Start`.
Example	`I2C_Write(0xA3);`

I2C_Stop

Prototype	`void I2C_Stop(void);`
Description	Issues STOP signal.
Requires	I²C must be configured before using this function. See `I2C_Init`.

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).

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

  I2C_Init(100000);
  I2C_Start();                  // Issue I2C start signal
  I2C_Wr(0xA2);                 // Send byte via I2C  (command to 24cO2)
  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
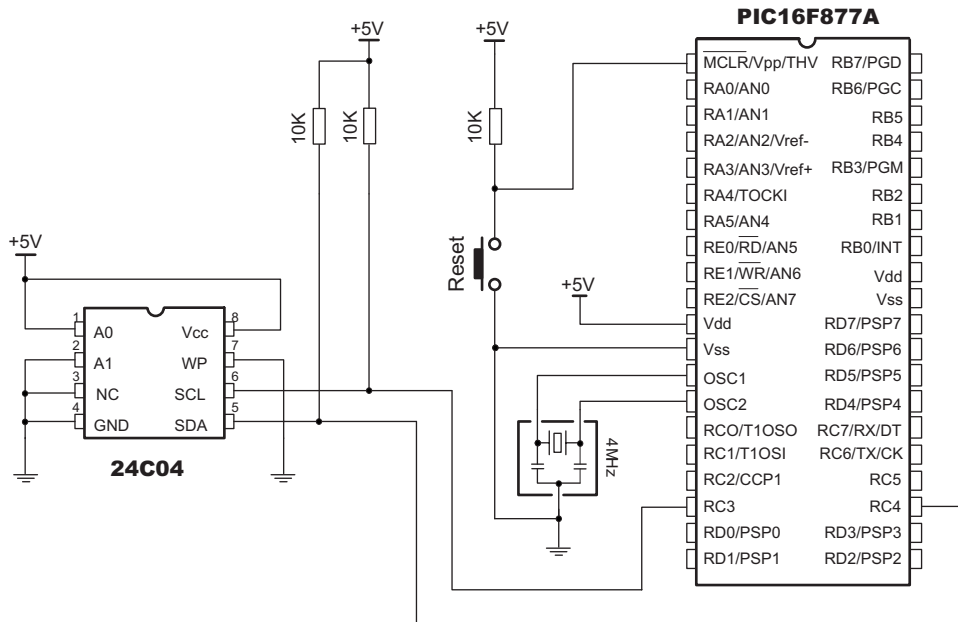
Keypad Library

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

Keypad_Read

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

Keypad_Released

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

Library Example

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

```c
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);

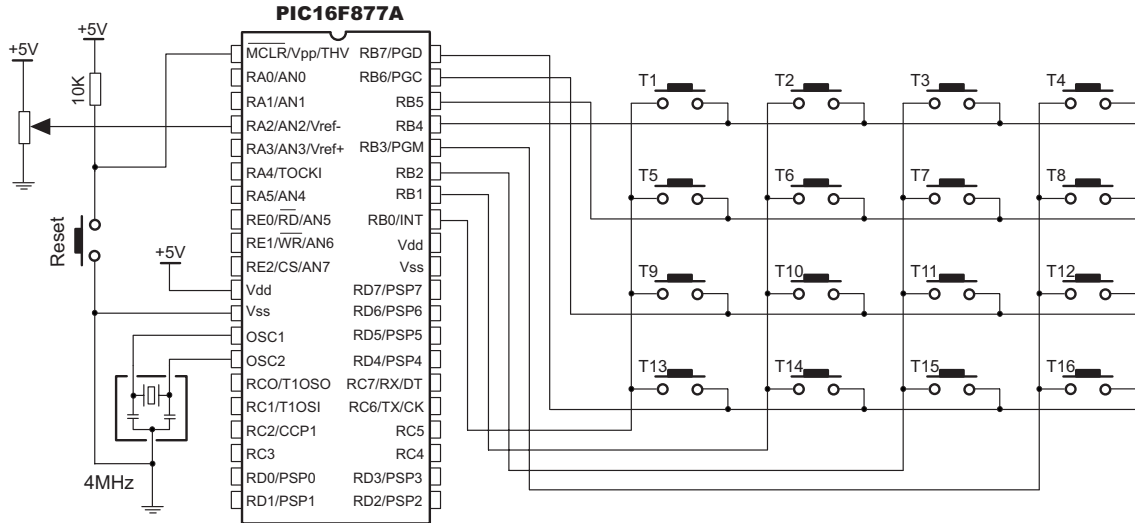    cnt++;

    //--- 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	**void** Lcd_Config(**char** *port, **char** RS, **char** EN, **char** WR, **char** D7, **char** D6, **char** D5, **char** D4);
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	Lcd_Config(PORTD,1,2,0,3,5,4,6);

Lcd_Init

Prototype	`void Lcd_Init(char *port);`
Description	Initializes LCD at port 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	`Lcd_Init(PORTB);`

Lcd_Out

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

Lcd_Out_Cp

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

Lcd_Chr

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

Lcd_Chr_Cp

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

Lcd_Cmd

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

LCD Commands

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

Library Example (default pin settings)

```c
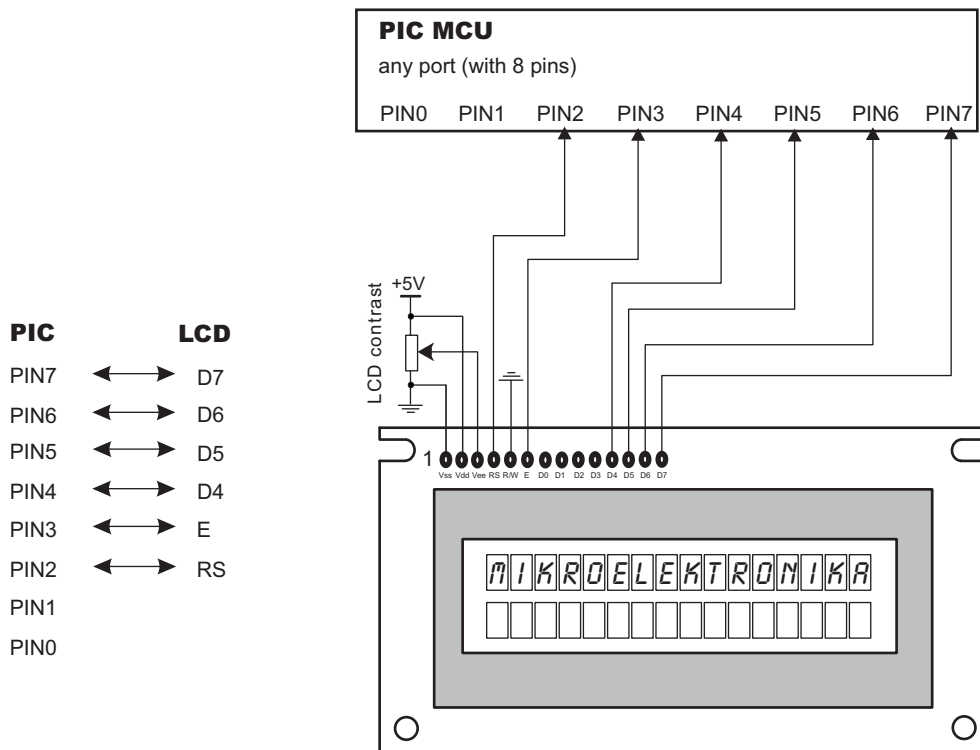char *text = "mikroElektronika";

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

Library Example (custom pin settings)

```c
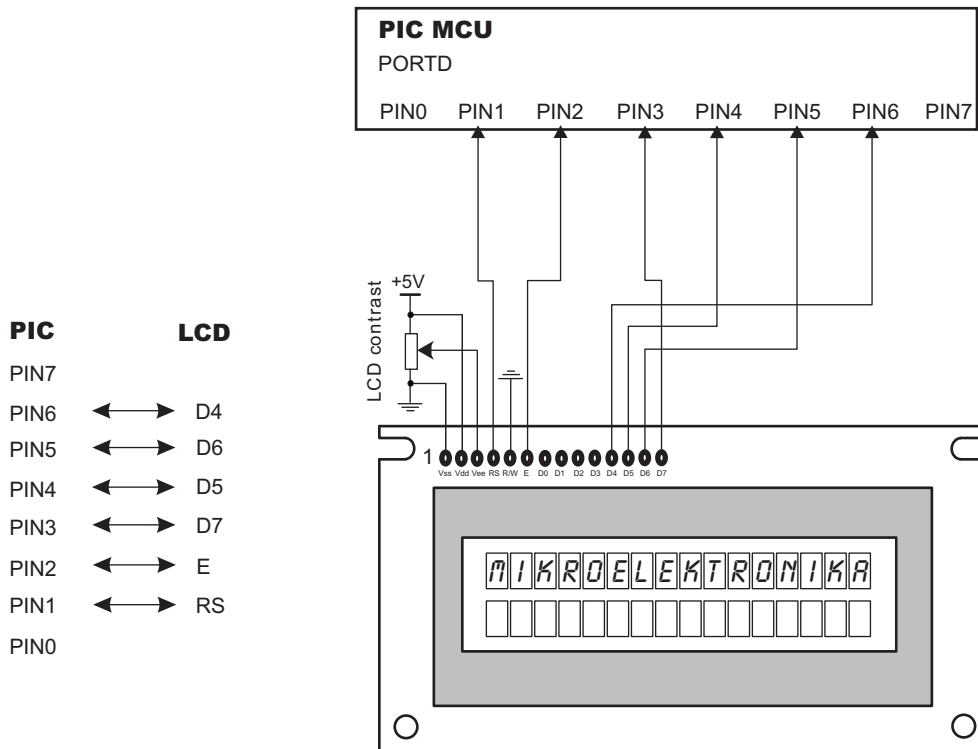char *text = "mikroElektronika";

void main() {
  TRISD = 0;                        // PORTD is output
  Lcd_Config(&PORTD,1,2,0,3,5,4,6); // Initialize LCD on PORTD
  Lcd_Cmd(Lcd_CURSOR_OFF);          // Turn off cursor
  Lcd_Out(1, 1, text);              // Print Text at LCD
}
```

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.

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

Library Routines

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

Lcd8_Config

Prototype	**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);
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	Lcd8_Config(PORTC,PORTD,0,1,2,6,5,4,3,7,1,2,0);

Lcd8_Init

Prototype	**void** Lcd8_Init(**char** *ctrlport, **char** *dataport);
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): E -> ctrlport.3, RS -> ctrlport.2, R/W -> ctrlport.0, D7 -> dataport.7, D6 -> dataport.6, D5 -> dataport.5, D4 -> dataport.4, D3 -> dataport.3, D2 -> dataport.2, D1 -> dataport.1, D0 -> dataport.0
Example	Lcd8_Init(PORTB, PORTC);

Lcd8_Out

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

Lcd8_Out_Cp

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

Lcd8_Chr

Prototype	**void** Lcd8_Chr(**char** row, **char** col, **char** character);
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 Lcd8_Config or Lcd8_Init.
Example	Lcd8_Out(2, 3, 'i'); *// Print 'i' at line 2, char 3*

Lcd8_Chr_Cp

Prototype	**void** Lcd8_Chr_Cp(**char** character);
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 Lcd8_Config or Lcd8_Init.
Example	Lcd8_Out_Cp('e'); *// Print 'e' at current cursor position*

Lcd8_Cmd

Prototype	**void** Lcd8_Cmd(**char** command);
Description	Sends command 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 Lcd8_Config or Lcd8_Init.
Example	Lcd8_Cmd(Lcd_Clear); *// Clear LCD display*

Library Example (default pin settings)

```c
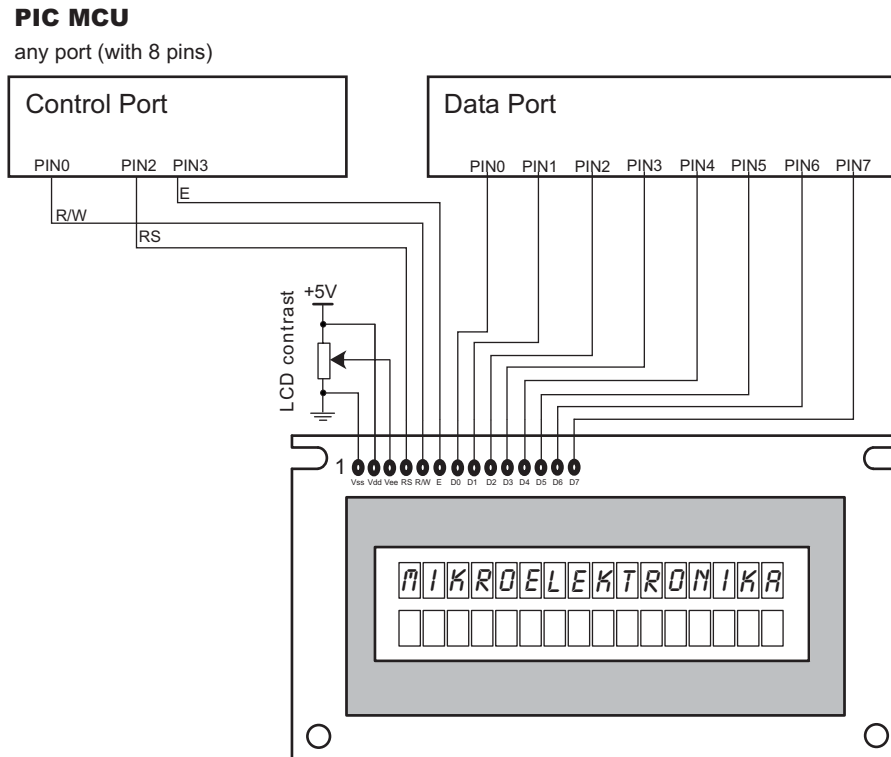char *text = "mikroElektronika";

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

PIC MCU

any port (with 8 pins)

Library Example (custom pin settings)

```c
char *text = "mikroElektronika";
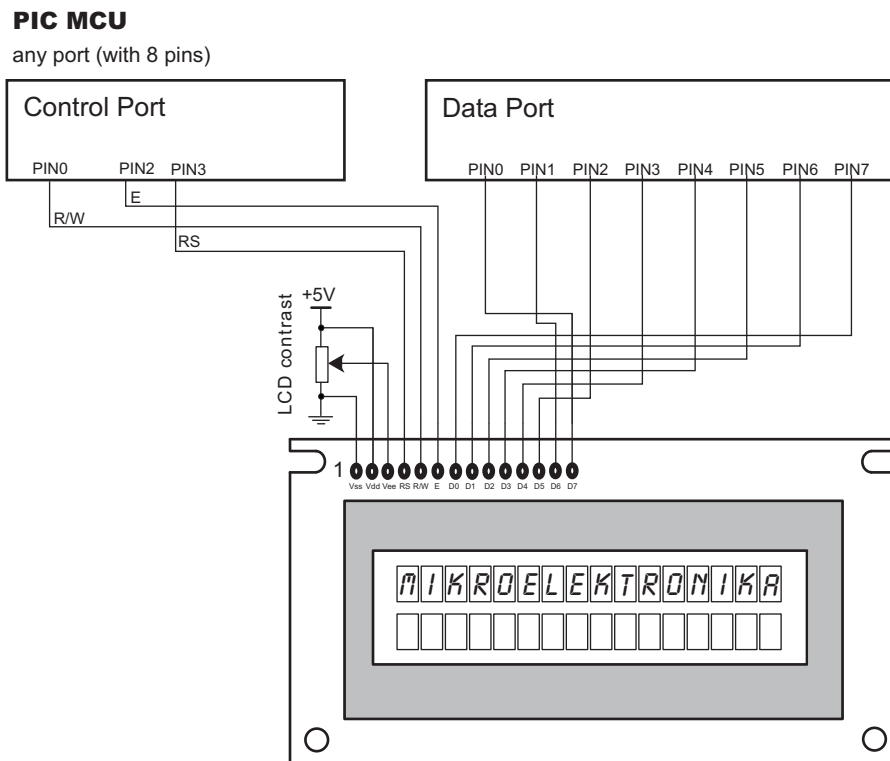
void main() {
  TRISB = 0;                       // PORTB is output
  TRISD = 0;                       // PORTD is output

  // Initialize LCD at PORTB and PORTD with custom pin settings
  Lcd8_Config(&PORTB,&PORTD,3,2,0,0,1,2,3,4,5,6,7);

  Lcd8_Cmd(Lcd_CURSOR_OFF);        // Turn off cursor
  Lcd8_Out(1, 1, text);            // Print text at 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.

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

Library Routines

Basic routines:

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

Advanced routines:

```
Glcd_Fill
Glcd_Dot
Glcd_Line
Glcd_V_Line
Glcd_H_Line
Glcd_Rectangle
Glcd_Box
Glcd_Circle
Glcd_Set_Font
Glcd_Write_Char
Glcd_Write_Text
Glcd_Image
Glcd_Partial_Image
```

Glcd_Init

Prototype	`void Glcd_Init(unsigned char *ctrl_port, char cs1, char cs2, char rs, char rw, char rst, char en, unsigned char *data_port);`
Description	Initializes GLCD at lower byte of `data_port` with pin settings you specify. Parameters `cs1`, `cs2`, `rs`, `rw`, `rst`, and `en` can be pins of any available port. This function needs to be called befored using other routines of GLCD library.
Example	`Glcd_Init(PORTB, PORTC, 3, 5, 7, 1, 2);`

Glcd_Disable

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

Glcd_Set_Side

Prototype	`void Glcd_Set_Side(unsigned short x);`
Description	Selects side of GLCD, left or right. Parameter `x` specifies the side: values from 0 to 63 specify the left side, and values higher than 64 specify the right side. Use the functions `Glcd_Set_Side`, `Glcd_Set_X`, and `Glcd_Set_Page` to specify an exact position on GLCD. Then, you can use `Glcd_Write_Data` or `Glcd_Read_Data` on that location.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Select_Side(0);`

Glcd_Set_Page

Prototype	`void Glcd_Set_Page(`**`unsigned short`**` page);`
Description	Selects page of GLCD, technically a line on display; parameter `page` can be 0..7.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Set_Page(5);`

Glcd_Set_X

Prototype	`void Glcd_Set_X(`**`unsigned short`**` x_pos);`
Description	Positions to x dots from the left border of GLCD within the given page.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Set_X(25);`

Glcd_Read_Data

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

Glcd_Write_Data

Prototype	**void** Glcd_Write_Data(**unsigned short** data);
Description	Writes data to the current location in GLCD memory and moves to the next location.
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Write_Data(data);

Glcd_Fill

Prototype	**void** Glcd_Fill(**unsigned short** pattern);
Description	Fills the GLCD memory with byte pattern. To clear the GLCD screen, use Glcd_Fill(0); to fill the screen completely, use Glcd_Fill($FF).
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Fill(0); // Clear screen

Glcd_Dot

Prototype	**void** Glcd_Dot(**unsigned short** x, **unsigned short** y, **char** color);
Description	Draws a dot on the GLCD at coordinates (x, y). Parameter color determines the dot state: 0 clears dot, 1 puts a dot, and 2 inverts dot state.
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Dot(0, 0, 2); // Invert the dot in the upper left corner

Glcd_Line

Prototype	`void Glcd_Line(int x1, int y1, int x2, int y2, char color);`
Description	Draws a line on the GLCD from `(x1, y1)` to `(x2, y2)`. Parameter `color` determines the dot state: 0 draws an empty line (clear dots), 1 draws a full line (put dots), and 2 draws a "smart" line (invert each dot).
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_Line(0, 63, 50, 0, 2);`

Glcd_V_Line

Prototype	`void Glcd_V_Line(unsigned short y1, unsigned short y2, unsigned short x, char color);`
Description	Similar to GLcd_Line, draws a vertical line on the GLCD from `(x, y1)` to `(x, y2)`.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_V_Line(0, 63, 0, 1);`

Glcd_H_Line

Prototype	`void Glcd_H_Line(unsigned short x1, unsigned short x2, unsigned short y, char color);`
Description	Similar to GLcd_Line, draws a horizontal line on the GLCD from `(x1, y)` to `(x2, y)`.
Requires	GLCD needs to be initialized. See `Glcd_Init`.
Example	`Glcd_H_Line(0, 127, 0, 1);`

Glcd_Rectangle

Prototype	`void Glcd_Rectangle(unsigned short x1, unsigned short y1, unsigned short x2, unsigned short y2, char color);`
Description	Draws a rectangle on the GLCD. Parameters (x1, y1) set the upper left corner, (x2, y2) set the bottom right corner. Parameter color defines the border: 0 draws an empty border (clear dots), 1 draws a solid border (put dots), and 2 draws a "smart" border (invert each dot).
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	`Glcd_Rectangle(10, 0, 30, 35, 1);`

Glcd_Box

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

Glcd_Circle

Prototype	**void** Glcd_Circle(**int** x, **int** y, **int** radius, **char** color);
Description	Draws a circle on the GLCD, centered at (x, y) with radius. Parameter color defines the circle line: 0 draws an empty line (clear dots), 1 draws a solid line (put dots), and 2 draws a "smart" line (invert each dot).
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Circle(63, 31, 25, 2);

Glcd_Set_Font

Prototype	**void** Glcd_Set_Font(**const char** *font, **unsigned short** font_width, **unsigned short** font_height);
Description	Sets font for routines Glcd_Write_Char and Glcd_Write_Text. Parameter font needs to formatted in an array of byte. Parameters font_width and font_height specify the width and height of characters in dots. Font width should not exceed 128 dots, and font height shouldn't exceed 8 dots. You can create your own fonts by following the guidelines given in file "GLcd_Fonts.c". This file contains the default fonts for GLCD, and is located in your installation folder, "Extra Examples" > "GLCD".
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	*// Use the array "myfont_5x8" with custom 5x8 font:* Glcd_Set_Font(myfont_5x8, 5, 8);

Glcd_Write_Char

Prototype	**void** Glcd_Write_Char(**unsigned short** character, **unsigned short** x, **unsigned short** page, **char** color);
Description	Prints character at page (one of 8 GLCD lines, 0..7), x dots away from the left border of display. Parameter color defines the "fill": 0 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 Glcd_Init.
Example	Glcd_Write_Char('C', 0, 0, 1);

Glcd_Write_Text

Prototype	**void** Glcd_Write_Text(**char** *text, **unsigned short** x, **unsigned short** page, **unsigned short** color);
Description	Prints text at page (one of 8 GLCD lines, 0..7), x dots away from the left border of display. Parameter color defines the "fill": 0 prints a "white" letters (clear dots), 1 prints solid letters (put dots), and 2 prints "smart" letters (invert each dot).
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Write_Text("Hello world!", 0, 0, 1);

Glcd_Image

Prototype	**void** Glcd_Image(**const char** *image);
Description	Displays bitmap image on the GLCD. Parameter image 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 Glcd_Init.
Example	Glcd_Image(my_image);

Glcd_Partial_Image

Prototype	**void** Glcd_Partial_Image(**unsigned short** x1, **unsigned short** y1, **unsigned short** x2, **unsigned short** y2, **unsigned short** color, **const char** *image);
Description	Displays partial bitmap image on the GLCD. Parameter image should be formatted as an array of 1024 bytes. Parameters (x1, y1) set the upper left corner, and (x2, y2) set the lower right corner of the clip. Parameter color defines the fill: 0 draws a "white" image (clear dots), 1 draws a "black" image (put dots), and 2 draws an inverted image (invert each dot). Use the mikroC's integrated Bitmap-to-LCD editor (menu option Tools > Graphic LCD Editor) to convert image to a constant array suitable for display on GLCD.
Requires	GLCD needs to be initialized. See Glcd_Init.
Example	Glcd_Partial_Image(0, 0, 32, 64, 1, my_image);

Library Example

The following drawing demo tests advanced routines of GLCD library.

```c
unsigned short j, k;

void main() {
  Glcd_Init(PORTB, 2, 0, 3, 5, 7, 1, PORTD);

  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);
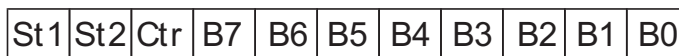    }
    Delay_ms(4000);
  } while (1);
}//~!
```

Hardware Connection
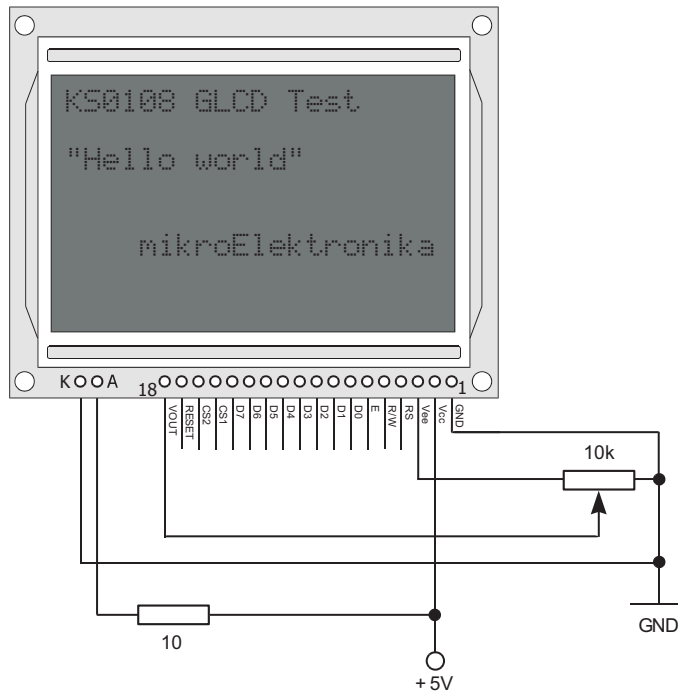
Manchester Code Library

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

| St1 | St2 | Ctr | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |

Bi-phase coding

1   0

2.4ms   Example of transmission

1 1 0 0 0 1 0 0 0 1 1
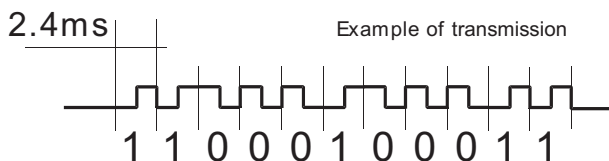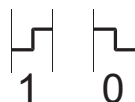
Notes: Manchester receive routines are blocking calls (`Man_Receive_Config`, `Man_Receive_Init`, `Man_Receive`). This means that PIC will wait until the task is performed (e.g. byte is received, synchronization achieved, etc). Routines for receiving are limited to a baud rate scope from 340 ~ 560 bps.

Library Routines

```
Man_Receive_Config
Man_Receive_Init
Man_Receive
Man_Send_Config
Man_Send_Init
Man_Send
```

Man_Receive_Config

Prototype	`void Man_Receive_Config(char *port, char rxpin);`
Description	The function prepares PIC for receiving signal. You need to specify the `port` and `rxpin` (0–7) of input signal. In case of multiple errors on reception, you should call `Man_Receive_Init` once again to enable synchronization.
Example	`Man_Receive_Config(&PORTD, 6);`

Man_Receive_Init

Prototype	`void Man_Receive_Init(char *port);`
Description	The function prepares PIC for receiving signal. You need to specify the `port`; `rxpin` is pin 6 by default. In case of multiple errors on reception, you should call `Man_Receive_Init` once again to enable synchronization.
Example	`Man_Receive_Init(&PORTD);`

Man_Receive

Prototype	`void Man_Receive(char *error);`
Returns	Returns one byte from signal.
Description	Function extracts one byte from signal. If signal format does not match the expected, `error` flag will be set to 255.
Requires	To use this function, you must first prepare the PIC for receiving. See `Man_Receive_Config` or `Man_Receive_Init`.
Example	`temp = Man_Receive(error);` `if (error) { ... /* error handling */ }`

Man_Send_Config

Prototype	`void Man_Send_Config(char *port, char txpin);`
Description	The function prepares PIC for sending signal. You need to specify `port` and `txpin` (0–7) for outgoing signal. Baud rate is const 500 bps.
Example	`Man_Send_Config(&PORTD, 0);`

Man_Send_Init

Prototype	`void Man_Receive_Init(char *port);`
Description	The function prepares PIC for sending signal. You need to specify `port` for outgoing signal; `txpin` is pin 0 by default. Baud rate is const 500 bps.
Example	`Man_Send_Init(&PORTD);`

Man_Send

Prototype	`void Man_Send(unsigned short data);`
Description	Sends one byte (`data`).
Requires	To use this function, you must first prepare the PIC for sending. See `Man_Send_Config` or `Man_Send_Init`.

Example	`unsigned short msg;`
	`...`
	`Man_Send(msg);`

Library Example

```c
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;              //   syncronize 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
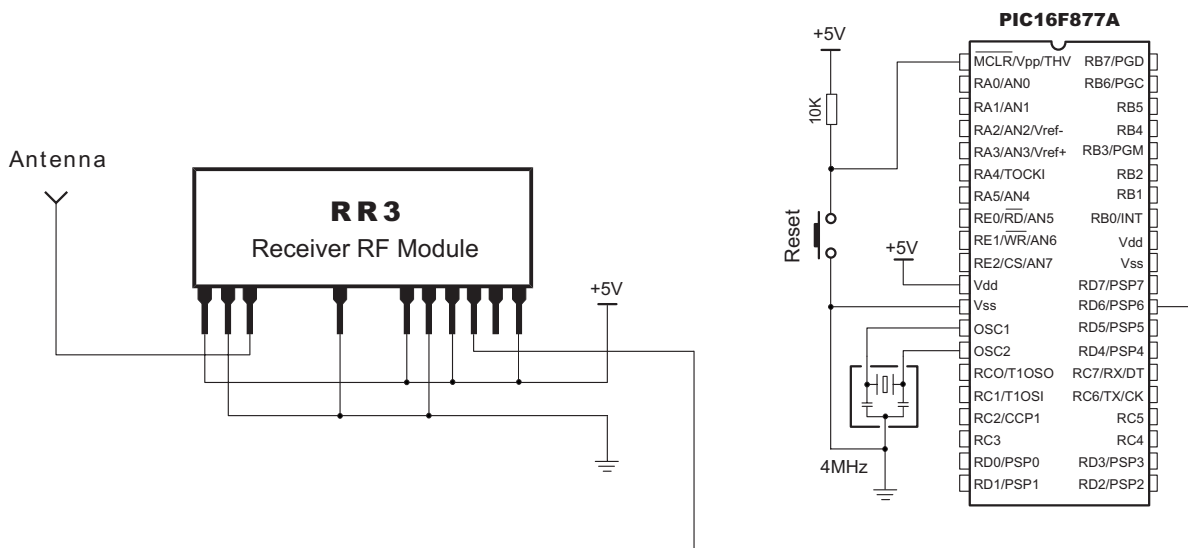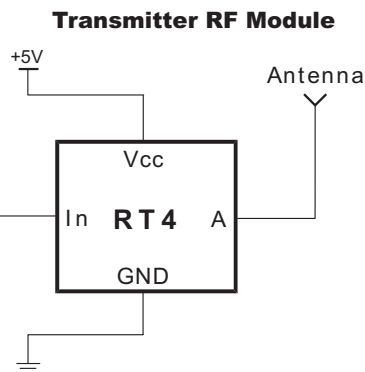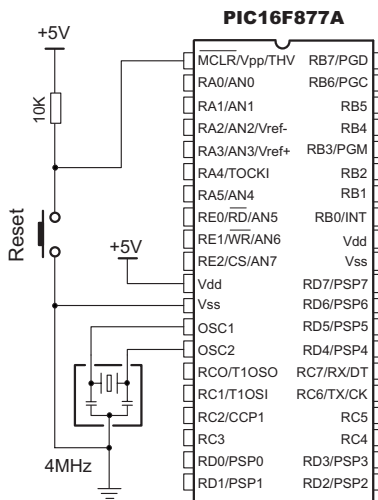
Multi Media Card Library

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

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 T1 table is corrupted.

Library Routines

```
Mmc_Init
Mmc_Read_Sector
Mmc_Write_Sector
Mmc_Read_Cid
Mmc_Read_Csd

Mmc_Fat_Init
Mmc_Fat_Assign
Mmc_Fat_Reset
Mmc_Fat_Rewrite
Mmc_Fat_Append
Mmc_Fat_Read
Mmc_Fat_Write
Mmc_Set_File_Date
```

Mmc_Init

Prototype	`unsigned short Mmc_Init(char *port, char pin);`
Returns	Returns 0 if MMC card is present and successfully initialized, otherwise returns 1.
Description	Initializes MMC through hardware SPI communication, 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 1 if MMC card is present and successfully initialized, otherwise returns 0.
Example	`while (Mmc_Init()) ;  // Loop until MMC is initialized`

Mmc_Read_Sector

Prototype	**unsigned short** Mmc_Read_Sector(**unsigned long** sector, **char** *data);
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads one sector (512 bytes) from MMC card at sector address sector. Read data is stored in the array data. Function returns 0 if read was successful, or 1 if an error occurred.
Requires	Library needs to be initialized, see Mmc_Init.
Example	error = Mmc_Read_Sector(sector, data);

Mmc_Write_Sector

Prototype	**unsigned short** Mmc_Write_Sector(**unsigned long** sector,**char** *data);
Returns	Returns 0 if write was successful; returns 1 if there was an error in sending write command; returns 2 if there was an error in writing.
Description	Function writes 512 bytes of data to MMC card at sector address sector. Function returns 0 if write was successful, or 1 if there was an error in sending write command, or 2 if there was an error in writing.
Requires	Library needs to be initialized, see Mmc_Init.
Example	error = Mmc_Write_Sector(sector, data);

Mmc_Read_Cid

Prototype	`unsigned short Mmc_Read_Cid(unsigned short *data_for_registers);`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CID register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see `Mmc_Init`.
Example	`error = Mmc_Read_Cid(data);`

Mmc_Read_Csd

Prototype	`unsigned short Mmc_Read_Csd(unsigned short *data_for_registers);`
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CSD register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see `Mmc_Init`.
Example	`error = Mmc_Read_Csd(data);`

Mmc_Fat_Init

Prototype	`unsigned short Mmc_Fat_Init(unsigned short *port, unsigned short pin);`
Returns	Returns 0 if MMC card is present and successfully initialized, otherwise returns 1.
Description	Initializes hardware SPI communication; designated CS line for communication is RC2. The function returns 0 if MMC card is present and successfully initialized, otherwise returns 1. This function needs to be called before using other functions of MMC FAT library.
Example	`// Loop until MMC FAT is initialized at RC2` `while (Mmc_Fat_Init(&PORTC, 2)) ;`

Mmc_Fat_Assign

Prototype	`void Mmc_Fat_Assign(char *filename);`
Description	This routine designates ("assigns") the file we'll be working with. Function looks for the file specified by the `filename` 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 `filename` must be 8 + 3 characters in uppercase.
Requires	Library needs to be initialized; see `Mmc_Fat_Init`.

Example	`// 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");`

Mmc_Fat_Reset

Prototype	**void** Mmc_Fat_Reset(**unsigned long** *size);
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 size stores the size of the assigned file, in bytes.
Requires	Library needs to be initialized; see Mmc_Fat_Init.
Example	Mmc_Fat_Reset(&filesize);

Mmc_Fat_Rewrite

Prototype	**void** Mmc_Fat_Rewrite(**void**);
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 Mmc_Fat_Init.
Example	Mmc_Fat_Rewrite();

Mmc_Fat_Append

Prototype	**void** Mmc_Fat_Append(**void**);
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 Mmc_Fat_Init.
Example	Mmc_Fat_Append();

Mmc_Fat_Read

Prototype	`void Mmc_Fat_Read(unsigned short *data);`
Description	Function reads the byte at which the file pointer points to and stores data into parameter `data`. The file pointer automatically increments with each call of `Mmc_Fat_Read`.
Requires	File pointer must be initialized; see `Mmc_Fat_Reset`.
Example	`Mmc_Fat_Read(&mydata);`

Mmc_Fat_Write

Prototype	`void Mmc_Fat_Write(char *fdata, unsigned data_len);`
Description	Function writes a chunk of `data_len` bytes (`fdata`) to the currently assigned file, at the position of the file pointer.
Requires	File pointer must be initialized; see `Mmc_Fat_Append` or `Mmc_Fat_Rewrite`.
Example	`Mmc_Fat_Write(txt, 21);` `Mmc_Fat_Write("Hello\nworld", 1);`

Mmc_Set_File_Date

Prototype	`void Mmc_Set_File_Date(unsigned year, char month, char day,` `char hours, char min, char sec);`
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 `Mmc_Fat_Append` or `Mmc_Fat_Rewrite`.
Example	`// April 1st 2005, 18:07:00` `Mmc_Set_File_Date(2005, 4, 1, 18, 7, 0);`

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.

```c
unsigned i;
unsigned short tmp;
unsigned short data[512];

void main() {

  Usart_Init(9600);

  // 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);

  /** Now to check sectors 55 and 56 **/

  // 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.

```c
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
  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);         // 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);         // Assign a file
    Mmc_Fat_Append();
    Mmc_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
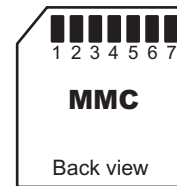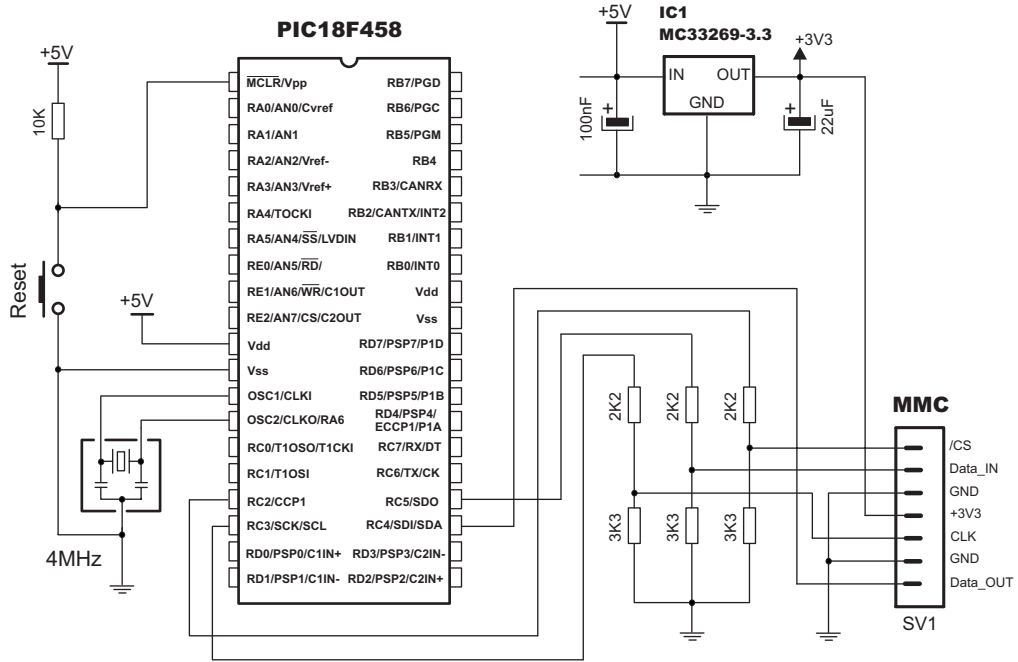
OneWire Library

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 Fosc 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	`char Ow_Reset(char *port, char pin);`
Returns	Returns 0 if DS1820 is present, 1 if not present.
Description	Issues OneWire reset signal for DS1820. Parameters `port` and `pin` specify the location of DS1820.
Requires	Works with Dallas DS1820 temperature sensor only.
Example	`Ow_Reset(&PORTA, 5); // reset DS1820 connected to the RA5 pin`

Ow_Read

Prototype	`char Ow_Read(char *port, char pin);`
Returns	Data read from an external device over the OneWire bus.
Description	Reads one byte of data via the OneWire bus.
Example	`tmp = Ow_Read(&PORTA, 5);`

Ow_Write

Prototype	`void Ow_Write(char *port, char pin, char par);`
Description	Writes one byte of data (argument `par`) via OneWire bus.
Example	`Ow_Write(&PORTA, 5, 0xCC);`

Library Example

```c
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
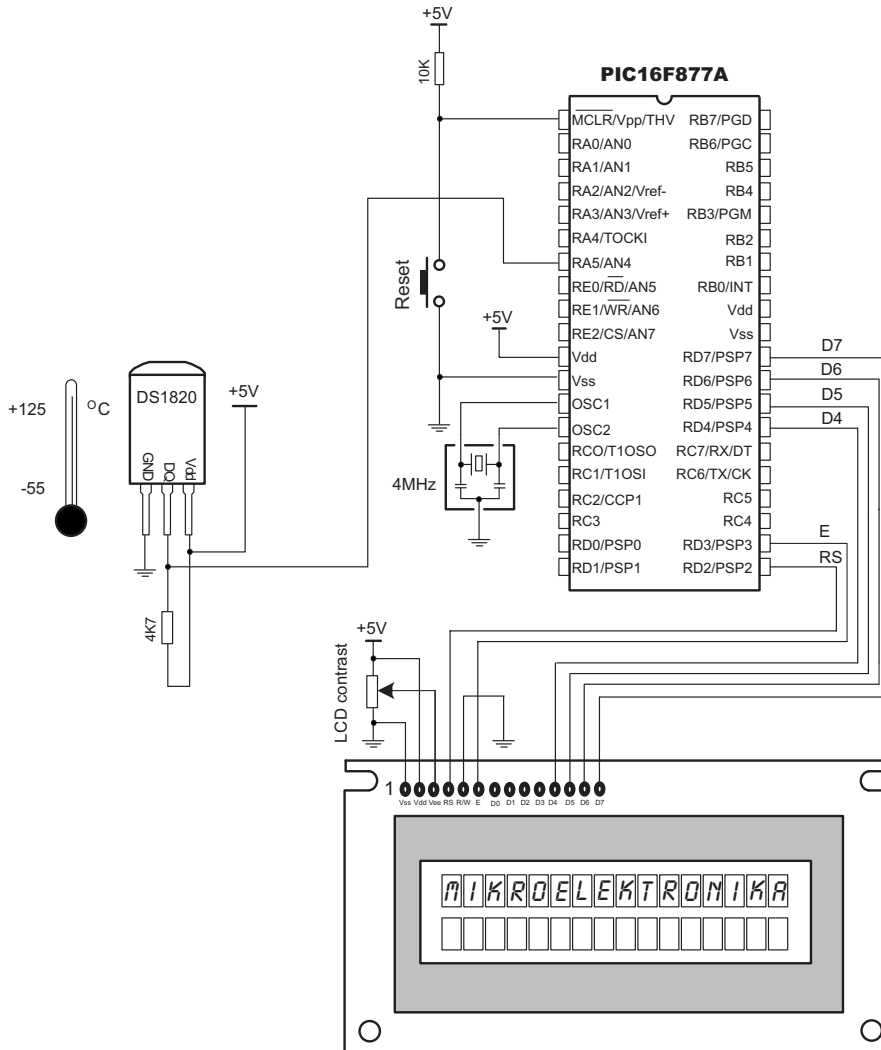
PS/2 Library

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	**void** Ps2_Init(**unsigned short** *port);
Description	Initializes `port` for work with PS/2 keyboard, with default pin settings. Port pin 0 is Data line, and port pin 1 is Clock line. You need to call either `Ps2_Init` or `Ps2_Config` before using other routines of PS/2 library.
Requires	Both Data and Clock lines need to be in pull-up mode.

Ps2_Config

Prototype	**void** Ps2_Config(**char** *port, **char** clock, **char** data);
Description	Initializes `port` for work with PS/2 keyboard, with custom pin settings. Parameters `data` and `clock` specify pins of `port` for Data line and Clock line, respectively. `Data` and `clock` need to be in range 0..7 and cannot point to the same pin. You need to call either `Ps2_Init` or `Ps2_Config` before using other routines of PS/2 library.
Requires	Both Data and Clock lines need to be in pull-up mode.
Example	Ps2_Config(&PORTB, 2, 3);

Ps2_Key_Read

Prototype	`char Ps2_Key_Read(char *value, char *special, char *pressed);`
Returns	Returns 1 if reading of a key from the keyboard was successful, otherwise 0.
Description	The function retrieves information about key pressed. Parameter `value` holds the value of the key pressed. For characters, numerals, punctuation marks, and space, value will store the appropriate ASCII value. Routine "recognizes" the function of Shift and Caps Lock, and behaves appropriately. Parameter `special` is a flag for special function keys (F1, Enter, Esc, etc). If key pressed is one of these, special will be set to 1, otherwise 0. Parameter `pressed` is set to 1 if the key is pressed, and 0 if released.
Requires	PS/2 keyboard needs to be initialized; see `Ps2_Init` or `Ps2_Config`.

Example	`// Press Enter to continue:`
	`do {`
	`  if (Ps2_Key_Read(&value, &special, &pressed)) {`
	`    if ((value == 13) && (special == 1)) break;`
	`  }`
	`} while (1);`

Library Example

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

```c
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: These routines support module on RC2, and won't work with modules on other ports. You can find examples for PICmicros with module on other ports in mikroC installation folder, subfolder "Examples". Also, mikroC does not support enhanced PWM modules.

Library Routines

```
Pwm_Init
Pwm_Change_Duty
Pwm_Start
Pwm_Stop
```

Pwm_Init

Prototype	`void Pwm_Init(long freq);`
Description	Initializes the PWM module with duty ratio 0. Parameter `freq` is a desired PWM frequency in Hz (refer to device data sheet for correct values in respect with Fosc). `Pwm_Init` needs to be called before using other functions from PWM Library.
Requires	You need a CCP module on PORTC to use this library. Check mikroC installation folder, subfolder "Examples", for alternate solutions.
Example	`Pwm_Init(5000);  // Initialize PWM module at 5KHz`

Pwm_Change_Duty

Prototype	`void Pwm_Change_Duty(char duty_ratio);`
Description	Changes PWM duty ratio. Parameter `duty_ratio` takes values from 0 to 255, where 0 is 0%, 127 is 50%, and 255 is 100% duty ratio. Other specific values for duty ratio can be calculated as (Percent*255)/100.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initalized – see `Pwm_Init`.
Example	`Pwm_Change_Duty(192);` `// Set duty ratio to 75%`

Pwm_Start

Prototype	`void Pwm_Start(void);`
Description	Starts PWM.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initalized – see `Pwm_Init`.
Example	`Pwm_Start();`

Pwm_Stop

Prototype	`void Pwm_Stop(void);`
Description	Stops PWM.
Requires	You need a CCP module on PORTC to use this library. To use this function, module needs to be initalized – see `Pwm_Init`.
Example	`Pwm_Stop();`

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
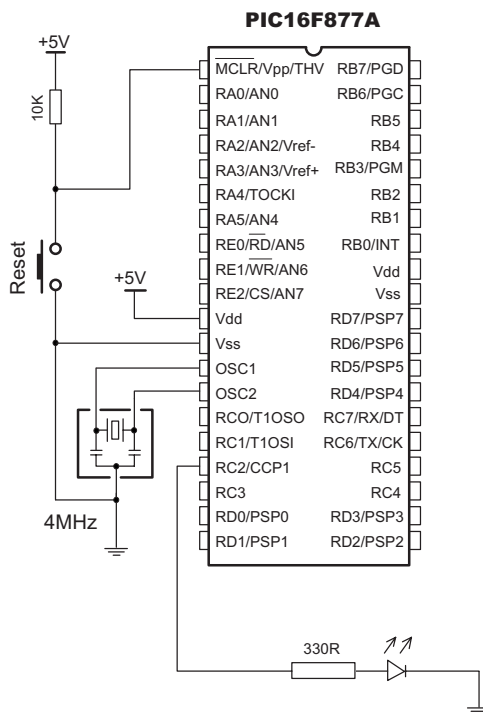
RS-485 Library

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.

Library Routines

```
RS485Master_Init
RS485Master_Receive
RS485Master_Send
RS485Slave_Init
RS485Slave_Receive
RS485Slave_Send
```

RS485Master_Init

Prototype	`void RS485Master_Init(void);`
Description	Initializes PIC MCU as Master in RS-485 communication.
Requires	USART HW module needs to be initialized. See `USART_Init`.
Example	`RS485Master_Init();`

RS485Master_Receive

Prototype	`void RS485Master_Receive(char *data);`
Description	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: `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. Function automatically adjusts `data[4]` and `data[5]` upon every received message. These flags need to be cleared from the program.
Requires	MCU must be initialized as Master in RS-485 communication in order to be assigned an address. See `RS485Master_Init`.

Example	`unsigned short msg[8];`
	`...`
	`RS485Master_Receive(msg);`

RS485Master_Send

Prototype	`void RS485Master_Send(char *data, char datalen, char address);`
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 `RS485Master_Init`. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`unsigned short msg[8];`
	`...`
	`RS485Master_Send(msg, 3, 0x12);`

RS485Slave_Init

Prototype	`void RS485Slave_Init(char address);`
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 `USART_Init`.
Example	`RS485Slave_Init(160); // Initialize MCU as Slave with address 160`

RS485Slave_Receive

Prototype	`void RS485Slave_Receive(char *data);`
Description	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: `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. Function automatically adjusts `data[4]` and `data[5]` upon every received message. These flags need to be cleared from the program.
Requires	MCU must be initialized as Slave in RS-485 communication in order to be assigned an address. See `RS485Slave_Init`.

Example	`unsigned short msg[8];`
	`...`
	`RS485Slave_Read(msg);`

RS485Slave_Send

Prototype	`void RS485Slave_Send(char *data, char datalen);`
Description	Sends `data` from buffer to Master via RS-485; `datalen` is a number of bytes in message (1 <= datalen <= 3).
Requires	MCU must be initialized as Slave in RS-485 communication in order to be assigned an address. See `RS485Slave_Init`. It is programmer's responsibility to ensure (by protocol) that only one device sends data via 485 bus at a time.

Example	`unsigned short msg[8];`
	`...`
	`RS485Slave_Send(msg, 2);`

Library Example

The example demonstrates working with PIC as Slave nod in RS-485 communication. PIC receives only packets addressed to it (address 160 in our example), and general messsages with target address 50. The received data is forwarded to PORTB, and sent back to Master.

```c
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(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

Connecting PC and PIC via RS485 communication line

Up to 32 devices can be connected to RS485 line

Shielded pair less than 300m long

RS232 to RS485 converter
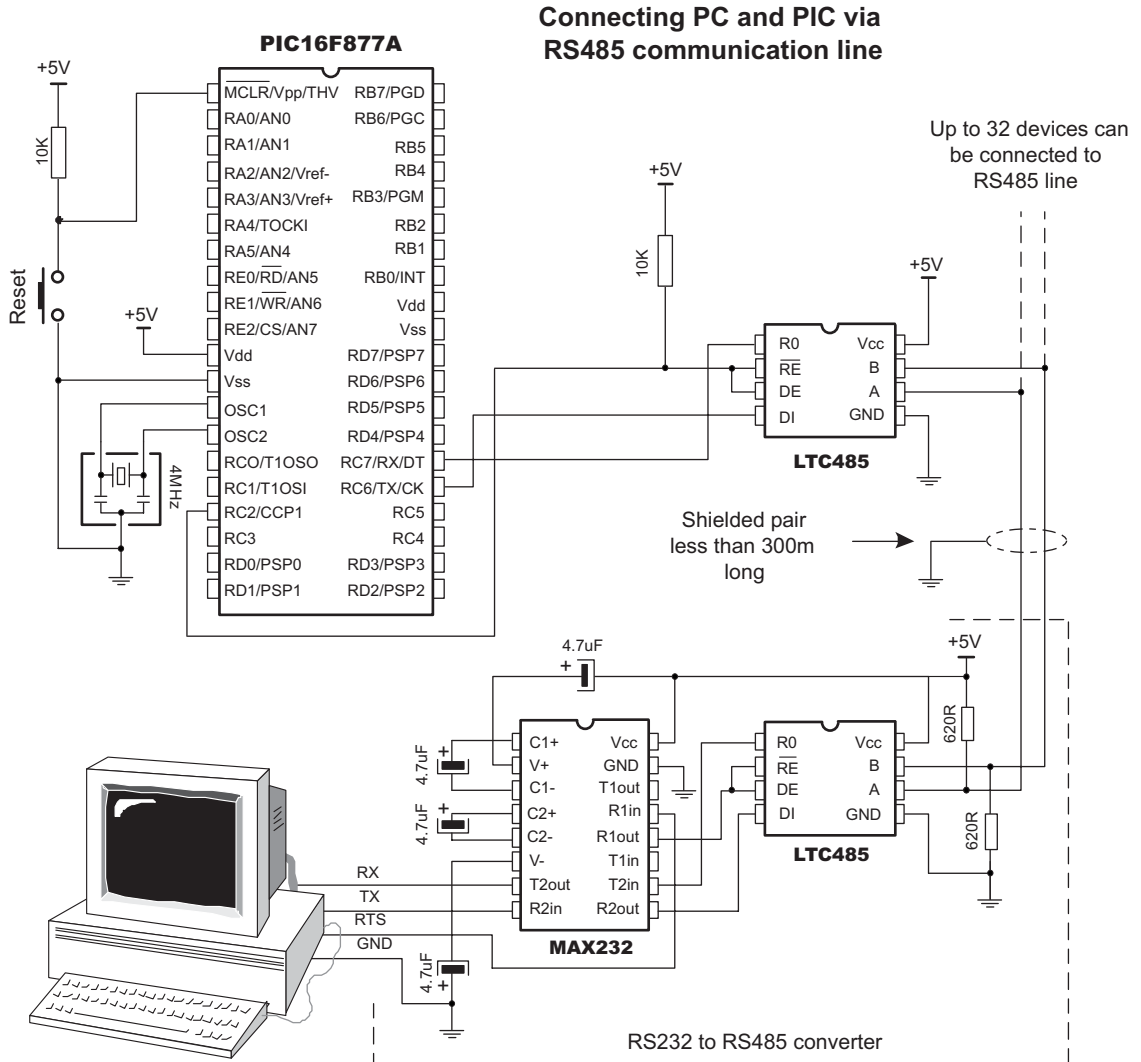
Secure Digital Library

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

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

Note: Secure Digital Library works only with PIC18 family.

Library Routines

```
Sd_Init
Sd_Read_Sector
Sd_Write_Sector
Sd_Read_Cid
Sd_Read_Csd
```

Sd_Init

Prototype	**unsigned short** Sd_Init(**unsigned short** *port, **unsigned short** pin);
Returns	Returns 0 if SD card is present and successfully initialized, otherwise returns 1.
Description	Initializes hardware SPI communication; parameters port and pin designate the CS line used in the communication (parameter pin should be 0..7). The function returns 0 if SD card is present and successfully initialized, otherwise returns 1. Sd_Init needs to be called before using other functions of this library.
Example	error = Sd_Init(&PORTC, 2); // Init with CS line at RC2

Sd_Read_Sector

Prototype	**unsigned short** Sd_Read_Sector(**unsigned long** sector, **char** *data);
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads one sector (512 bytes) from SD card at sector address `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 `Sd_Init`.
Example	`error = Sd_Read_Sector(sector, data);`

Sd_Write_Sector

Prototype	**unsigned short** Sd_Write_Sector(**unsigned long** sector,**char** *data);
Returns	Returns 0 if write was successful; returns 1 if there was an error in sending write command; returns 2 if there was an error in writing.
Description	Function writes 512 bytes of data to SD card at sector address 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 `Sd_Init`.
Example	`error = Sd_Write_Sector(sector, data);`

Sd_Read_Cid

Prototype	**unsigned short** Sd_Read_Cid(**unsigned short** *data_for_registers);
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CID register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see `Sd_Init`.
Example	`error = Sd_Read_Cid(data);`

Sd_Read_Csd

Prototype	**unsigned short** Sd_Read_Csd(**unsigned short** *data_for_registers);
Returns	Returns 0 if read was successful, or 1 if an error occurred.
Description	Function reads CSD register and returns 16 bytes of content into `data_for_registers`.
Requires	Library needs to be initialized, see `Sd_Init`.
Example	`error = Sd_Read_Csd(data);`

Library Example

The following code tests SD library routines. First, we fill the buffer with 512 "M" characters and write it to sector 56; then we repeat the sequence with character "E" at sector 56. Finally, we read the sectors 55 and 56 to check if the write was successful.

```c
unsigned i;
unsigned short tmp;
unsigned short data[512];

void main() {

  Usart_Init(9600);

  // Initialize ports
  tmp = Sd_Init(&PORTC, 2);

  // Fill the buffer with the 'M' character
  for (i = 0; i <= 511; i++) data[i] = 'M';

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

  // Fill the buffer with the 'E' character
  for (i = 0; i <= 511; i++) data[i] = 'E'

  // Write it to SD card, sector 56
  tmp = Sd_Write_Sector(56, data);

  /** Now to check sectors 55 and 56 **/

  // Read from sector 55
  tmp = Sd_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 = Sd_Read_Sector(56, data);

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

}//~!
```

Hardware Connection
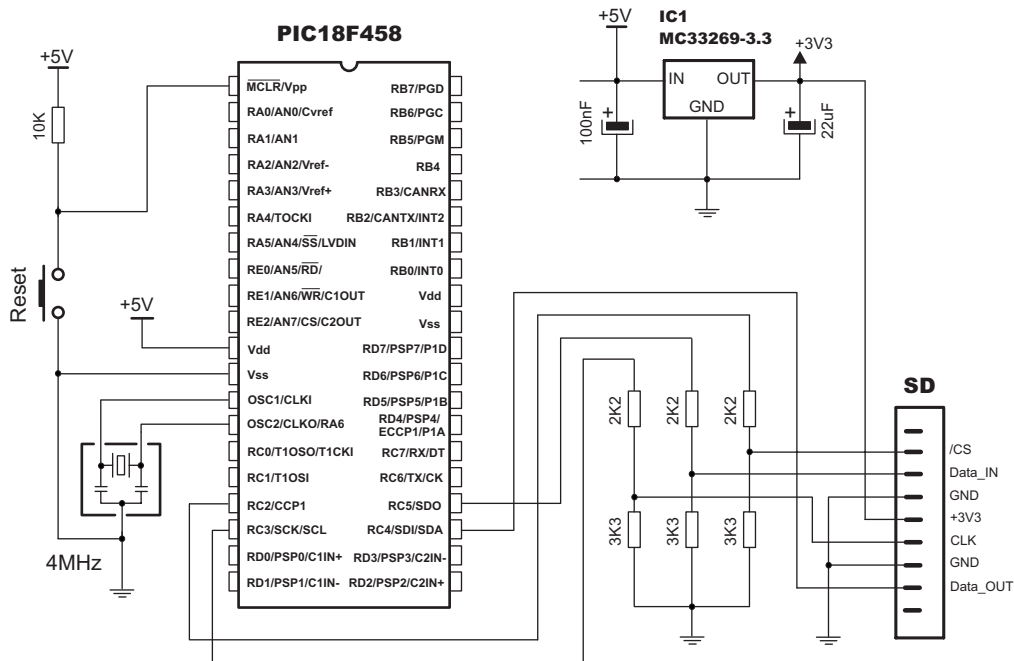
Software I2C Library

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	**void** Soft_I2C_Config(**char *port, const char** SDI, **const char** SD0, **const char** SCK);
Description	Configures software I²C. Parameter `port` specifies port of MCU on which SDA and SCL pins are located. Parameters SCL and SDA need to be in range 0–7 and cannot point at the same pin. Soft_I2C_Config needs to be called before using other functions from Soft I2C Library.
Example	Soft_I2C_Config(PORTB, 1, 2);

Soft_I2C_Start

Prototype	`void Soft_I2C_Start(void);`
Description	Issues START signal. Needs to be called prior to sending and receiving data.
Requires	Soft I²C must be configured before using this function. See `Soft_I2C_Config`.
Example	`Soft_I2C_Start();`

Soft_I2C_Read

Prototype	`char Soft_I2C_Read(char ack);`
Returns	Returns one byte from the slave.
Description	Reads one byte from the slave, and sends not acknowledge signal if parameter ack is 0, otherwise it sends acknowledge.
Requires	START signal needs to be issued in order to use this function. See `Soft_I2C_Start`.
Example	`tmp = Soft_I2C_Read(0); //Read data, send not-acknowledge signal`

Soft_I2C_Write

Prototype	`char Soft_I2C_Write(char data);`
Returns	Returns 0 if there were no errors.
Description	Sends data byte (parameter `data`) via I²C bus.
Requires	START signal needs to be issued in order to use this function. See `Soft_I2C_Start`.
Example	`Soft_I2C_Write(0xA3);`

Soft_I2C_Stop

Prototype	`void Soft_I2C_Stop(void);`
Description	Issues STOP signal.
Requires	START signal needs to be issued in order to use this function. See `Soft_I2C_Start`.
Example	`Soft_I2C_Stop();`

Library Example

```c
/* 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.

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	**void** Soft_Spi_Config(**char** *port, **const char** SDI, **const char** SD0, **const char** SCK);
Description	Configures and initializes software SPI. Parameter port specifies port of MCU on which SDI, SDO, and SCK pins will be located. Parameters SDI, SDO, and SCK need to be in range 0–7 and cannot point at the same pin. Soft_Spi_Config needs to be called before using other functions from Soft SPI Library.
Example	This will set SPI to master mode, clock = 50kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge. SDI pin is RB1, SDO pin is RB2 and SCK pin is RB3: Soft_Spi_Config(PORTB, 1, 2, 3);

Soft_Spi_Read

Prototype	`char Soft_Spi_Read(char buffer);`
Returns	Returns the received data.
Description	Provides clock by sending `buffer` and receives data.
Requires	Soft SPI must be initialized and communication established before using this function. See `Soft_Spi_Config`.
Example	`tmp = Soft_Spi_Read(buffer);`

Soft_Spi_Write

Prototype	`void Soft_Spi_Write(char data);`
Description	Immediately transmits `data`.
Requires	Soft SPI must be initialized and communication established before using this function. See `Soft_Spi_Config`.
Example	`Soft_Spi_Write(1);`

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.

```c
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	**void** Soft_Uart_Init(**unsigned short** *port, **unsigned short** rx, **unsigned short** tx, **unsigned short** baud_rate, **char** inverted);
Description	Initalizes software UART. Parameter port specifies port of MCU on which RX and TX pins are located; parameters rx and tx need to be in range 0–7 and cannot point at the same pin; baud_rate is the desired baud rate. Maximum baud rate depends on PIC's clock and working conditions. Parameter inverted, if set to non-zero value, indicates inverted logic on output. Soft_Uart_Init needs to be called before using other functions from Soft UART Library.
Example	Soft_Uart_Init(&PORTB, 1, 2, 9600, 0);

Soft_Uart_Read

Prototype	`unsigned short Soft_Uart_Read(unsigned short *error);`
Returns	Returns a received byte.
Description	Function receives a byte via software UART. Parameter `error` will be zero if the transfer was successful. This is a non-blocking function call, so you should test the `error` manually (check the example below).
Requires	Soft UART must be initialized and communication established before using this function. See `Soft_Uart_Init`.
| Example | ```
// 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) {...}
``` |

Soft_Uart_Write

Prototype	`void Soft_Uart_Write(char data);`
Description	Function transmits a byte (data) via UART.
Requires	Soft UART must be initialized and communication established before using this function. See `Soft_Uart_Init`.

Be aware that during transmission, software UART is incapable of receiving data – data transfer protocol must be set in such a way to prevent loss of information. |
| Example | ```
char some_byte = 0x0A;
...
Soft_Uart_Write(some_byte);
``` |

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**.

```c
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	`void Sound_Init(char *port, char pin);`
Description	Prepares hardware for output at specified port and pin. Parameter pin needs to be within range 0–7.
Example	`Sound_Init(PORTB, 2);  // Initialize sound on RB2`

Sound_Play

Prototype	`void Sound_Play(char period_div_10, unsigned num_of_periods);`
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 = 1ms = 1000 cycles @ 4MHz. This gives us our first parameter: 1000/10 = 100. Play 150 periods like this: `Sound_Play(100, 150);`

Library Example

The example is a simple demonstration of how to use sound library for playing tones on a piezo speaker. The code can be used with any MCU that has PORTB 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).

```c
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: This library supports module on PORTB or PORTC, and will not work with modules on other ports. Examples for PICmicros with module on other ports can be found in your mikroC installation folder, subfolder "Examples".

Library Routines

```
Spi_Init
Spi_Init_Advanced
Spi_Read
Spi_Write
```

Spi_Init

Prototype	**void** Spi_Init(**void**);
Description	Configures and initializes SPI with default settings. `SPI_Init_Advanced` or `SPI_Init` needs to be called before using other functions from SPI Library. Default settings are: Master mode, clock Fosc/4, clock idle state low, data transmitted on low to high edge, and input data sampled at the middle of interval. For custom configuration, use `Spi_Init_Advanced`.
Requires	You need PIC MCU with hardware integrated SPI.
Example	`Spi_Init();`

Spi_Init_Advanced

Prototype	`void Spi_Init_Advanced(char master, char data_sample, char clock_idle, char transmit_edge);`
Description	Configures and initializes SPI. `Spi_Init_Advanced` or `SPI_Init` needs to be called before using other functions of SPI Library. Parameter `mast_slav` determines the work mode for SPI; can have the values: `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* The `data_sample` determines when data is sampled; can have the values: `DATA_SAMPLE_MIDDLE` *// Input data sampled in middle of interval* `DATA_SAMPLE_END` *// Input data sampled at the end of interval* Parameter `clock_idle` determines idle state for clock; can have the following values: `CLK_IDLE_HIGH` *// Clock idle HIGH* `CLK_IDLE_LOW` *// Clock idle LOW* Parameter `transmit_edge` can have the following values: `LOW_2_HIGH` *// Data transmit on low to high edge* `HIGH_2_LOW` *// Data transmit on high to low edge*
Requires	You need PIC MCU with hardware integrated SPI.
Example	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: `Spi_Init_Advanced(MASTER_OSC_DIV4, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH)`

Spi_Read

Prototype	`char Spi_Read(char buffer);`
Returns	Returns the received data.
Description	Provides clock by sending `buffer` and receives data at the end of period.
Requires	SPI must be initialized and communication established before using this function. See `Spi_Init_Advanced` or `Spi_Init`.

Example	`short take, buffer;`
	`...`
	`take = Spi_Read(buffer);`

Spi_Write

Prototype	`void Spi_Write(char data);`
Description	Writes byte `data` to SSPBUF, and immediately starts the transmission.
Requires	SPI must be initialized and communication established before using this function. See `Spi_Init_Advanced` or `Spi_Init`.
Example	`Spi_Write(1);`

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.

```c
//------------------ 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;
}//~!
```

HW Connection
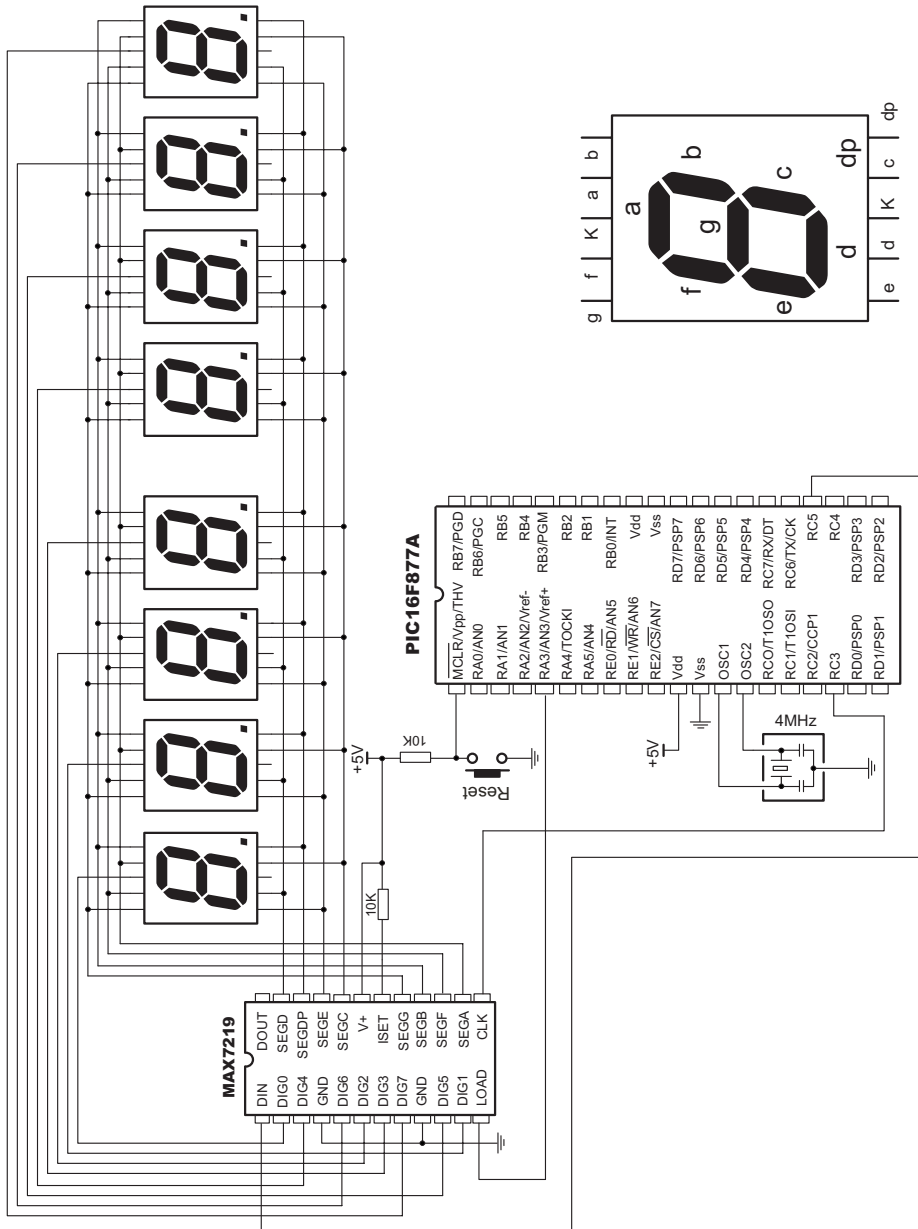
USART Library

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();`

Usart_Init

Prototype	`void Usart_Init(const long baud_rate);`
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	`Usart_Init(2400);  // Establish communication at 2400 bps`

Usart_Data_Ready

Prototype	`char Usart_Data_Ready(void);`
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 `Usart_Init`.

Example	`int receive;`
	`...`
	`// If data is ready, read it:`
	`if (Usart_Data_Ready()) receive = Usart_Read;`

Usart_Read

Prototype	`char Usart_Read(void);`
Returns	Returns the received byte. If byte is not received, returns 0.
Description	Function receives a byte via USART. Use the function `Usart_Data_Ready` to test if data is ready first.
Requires	USART HW module must be initialized and communication established before using this function. See `Usart_Init`.

Example	`int receive;`
	`...`
	`// If data is ready, read it:`
	`if (Usart_Data_Ready()) receive = Usart_Read;`

Usart_Write

Prototype	`char Usart_Write(char data);`
Description	Function transmits a byte (`data`) via USART.
Requires	USART HW module must be initialized and communication established before using this function. See `Usart_Init`.
Example	```int chunk;
...
Usart_Write(chunk);   /* send data chunk via USART */```

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

SUB-D 9-pin connector

4.7uF

C1+   Vcc
V+   GN
C1-   D
C2+   T1out
C2-   R1in
V-   R1out
T2out   T1in
R2in   T2in
       R2out

MAX232

serial cable (1 to 1)

Receive data (Rx)

Send data (Tx)

+5V

PIC16F877A

MCLR/Vpp/THV	RB7/PGD
RA0/AN0	RB6/PGC
RA1/AN1	RB5
RA2/AN2/Vref-	RB4
RA3/AN3/Vref+	RB3/PGM
RA4/TOCKI	RB2
RA5/AN4	RB1
RE0/RD/AN5	RB0/INT
RE1/WR/AN6	Vdd
RE2/CS/AN7	Vss
Vdd	RD7/PSP7
Vss	RD6/PSP6
OSC1	RD5/PSP5
OSC2	RD4/PSP4
RCO/T1OSO	RC7/RX/DT
RC1/T1OSI	RC6/TX/CK
RC2/CCP1	RC5
RC3	RC4
RD0/PSP0	RD3/PSP3
RD1/PSP1	RD2/PSP2

10K

Reset

4MHz

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	**void** Hid_Enable(**unsigned ***readbuff, **unsigned ***writebuff);
Description	Enables USB HID communication. Parameters `readbuff` and `writebuff` are the Read Buffer and the Write Buffer, respectively, which are used for HID communication. This function needs to be called before using other routines of USB HID Library.
Example	Hid_Enable(&rd, &wr);

Hid_Read

Prototype	`unsigned short Hid_Read(void);`
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 `Hid_Enable`.
Example	`get = Hid_Read();`

Hid_Write

Prototype	`void Hid_Write(unsigned *writebuff, unsigned short len);`
Description	Function sends data from `wrbuff` to host. Write Buffer is the same parameter as used in initialization. Parameter `len` should specify a length of the data to be transmitted.
Requires	USB HID needs to be enabled before using this function. See `Hid_Enable`.
Example	`Hid_Write(&wr, len);`

Hid_Disable

Prototype	`void Hid_Disable(void);`
Description	Disables USB HID communication.
Requires	USB HID needs to be enabled before using this function. See `Hid_Enable`.
Example	`Hid_Disable();`

Library Example

The following example continually sends sequence of numbers 0..255 to the PC via Universal Serial Bus.

```
unsigned short m, k;
unsigned short userRD_buffer[64];
unsigned short userWR_buffer[64];

void interrupt() {
  asm CALL _Hid_InterruptProc
  asm nop
}//~

void Init_Main() {
  // Disable all interrupts
  // Disable GIE, PEIE, TMR0IE, INT0IE,RBIE
  INTCON = 0;
  INTCON2 = 0xF5;
  INTCON3 = 0xC0;
  // Disable Priority Levels on interrupts
  RCON.IPEN = 0;
  PIE1 = 0; PIE2 = 0; PIR1 = 0; PIR2 = 0;

  // Configure all ports with analog function as digital
  ADCON1 |= 0x0F;

  // Ports Configuration
  TRISA = 0; TRISB = 0; TRISC = 0xFF; TRISD = 0xFF; TRISE = 0x07;
  LATA = 0; LATB = 0; LATC = 0; LATD = 0; LATE = 0;

  // Clear user RAM
  // Banks [00 .. 07] ( 8 x 256 = 2048 Bytes )
  asm {
    LFSR      FSR0, 0x000
    MOVLW     0x08
    CLRF      POSTINC0, 0
    CPFSEQ    FSR0H, 0
    BRA       $ - 2
  }
```

```
  // Timer 0
  T0CON  = 0x07;
  TMR0H  = (65536-156) >> 8;
  TMR0L  = (65536-156) & 0xFF;
  INTCON.T0IE = 1;                 // Enable T0IE
  T0CON.TMR0ON = 1;
}//~


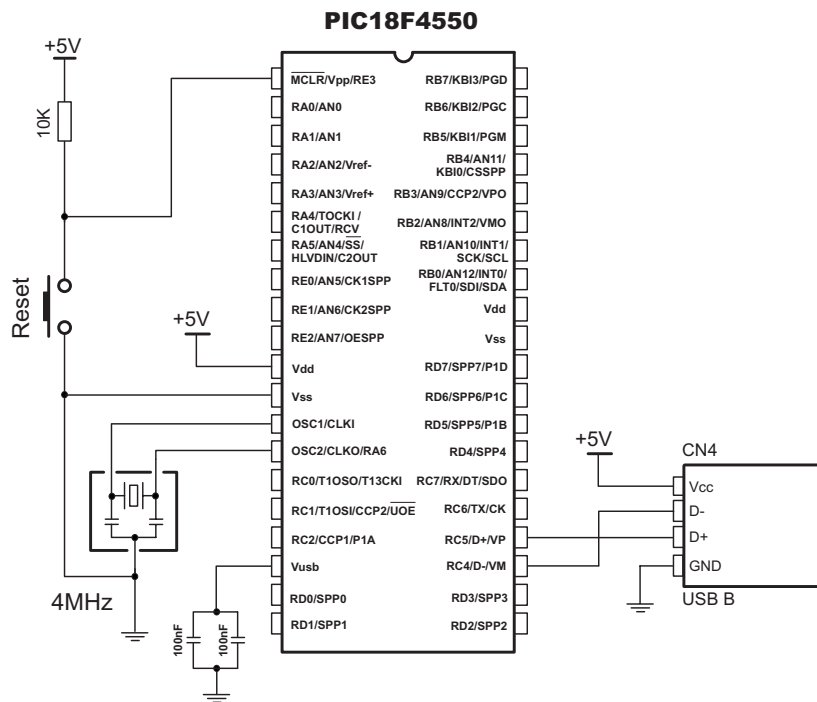/** Main Program Routine **/

void main() {
  Init_Main();
  Hid_Enable(&userRD_buffer, &userWR_buffer);

  do {
    for (k = 0; k < 255; k++) {
      // Prepare send buffer
      userWR_buffer[0] = k;

      // Send the number via USB
      Hid_Write(&userWR_buffer, 1);
    }
  } while (1);

  Hid_Disable();
}//~!
```

HW Connection

Util Library

Util library contains miscellaneous routines useful for project development.

Button

Prototype	`char Button(char *port, char pin, char time, char active_state);`
Returns	Returns 0 or 255.
Description	Function eliminates the influence of contact flickering upon pressing a button (debouncing). Parameters `port` and `pin` specify location of the button; parameter `time` specifies the minimum time pin has to be in active state in order to return TRUE; parameter `active_state` can be either 0 or 1, and it determines if button is active upon logical zero or logical one.

Example	Example reads RB0, to which the button is connected; on transition from 1 to 0 (release of button), PORTD is inverted:
	
	`do {`
	`  if (Button(&PORTB, 0, 1, 1)) oldstate = 1;`
	`  if (oldstate && Button(&PORTB, 0, 1, 0)) {`
	`    PORTD = ~PORTD;`
	`    oldstate = 0;`
	`  }`
	`} while(1);`

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	`char isalnum(char character);`
Description	Function returns 1 if the `character` is alphanumeric (A-Z, a-z, 0-9), otherwise returns zero.

isalpha

Prototype	`char isalpha(char character);`
Description	Function returns 1 if the `character` is alphabetic (A-Z, a-z), otherwise returns zero.

iscntrl

Prototype	`char iscntrl(char character);`
Description	Function returns 1 if the `character` is a control character or delete (decimal 0-31 and 127), otherwise returns zero.

isdigit

Prototype	`char isdigit(char character);`
Description	Function returns 1 if the `character` is a digit (0-9), otherwise returns zero.

isgraph

Prototype	`char isgraph(char character);`
Description	Function returns 1 if the `character` is a printable character, excluding the space (decimal 32), otherwise returns zero.

islower

Prototype	`char islower(char character);`
Description	Function returns 1 if the `character` is a lowercase letter (a-z), otherwise returns zero.

isprint

Prototype	`char isprint(char character);`
Description	Function returns 1 if the `character` is printable (decimal 32-126), otherwise returns zero.

ispunct

Prototype	`char ispunct(char character);`
Description	Function returns 1 if the `character` is punctuation (decimal 32-47, 58-63, 91-96, 123-126), otherwise returns zero.

isspace

Prototype	`char isspace(char character);`
Description	Function returns 1 if the `character` is white space (space, CR, HT, VT, NL, FF), otherwise returns zero.

isupper

Prototype	`char isupper(char character);`
Description	Function returns 1 if the `character` is an uppercase letter (A-Z), otherwise returns 0.

isxdigit

Prototype	`char isxdigit(char character);`
Description	Function returns 1 if the `character` is a hex digit (0-9, A-F, a-f), otherwise returns zero.

toupper

Prototype	`char toupper(int character);`
Description	If the `character` is a lowercase letter (a-z), function returns an uppercase letter. Otherwise, function returns an unchanged input parameter.

tolower

Prototype	`char tolower(int character);`
Description	If the `character` is an uppercase letter (A-Z), function returns a lowercase letter. Otherwise, function returns an unchanged input parameter.

ANSI C Math Library

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	`double acos(double x);`
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	`double asin(double x);`
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	`double atan(double x);`
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	`double atan2(double x);`
Description	This is the two argument arc tangent function. It is similar to computing the arc tangent of y/x, except that the signs of both arguments are used to determine the quadrant of the result, and x is permitted to be zero. The return value is in radians, between -pi and pi (inclusive).

ceil

Prototype	`double ceil(double num);`
Description	Function returns value of parameter num rounded up to the next whole number.

cos

Prototype	`double cos(double x);`
Description	Function returns the cosine of x in radians. The return value is from -1 to 1.

cosh

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

exp

Prototype	`double exp(double x);`
Description	Function returns the value of e — the base of natural logarithms — raised to the power of x (i.e. e^x).

fabs

Prototype	`double fabs(double num);`
Description	Function returns the absolute (i.e. positive) value of num.

floor

Prototype	`double floor(double num);`
Description	Function returns value of parameter `num` rounded down to the nearest integer.

frexp

Prototype	`double frexp(double num, int *exp);`
Description	Function splits a floating-point value `num` into a normalized fraction and an integral power of 2. Return value is the normalized fraction, and the integer `exp` is stored in the object pointed to by `exp`.

ldexp

Prototype	`double ldexp(double num, int exp);`
Description	Function returns the result of multiplying the floating-point number `num` by 2 raised to the power exp (i.e. returns $x*2^{exp}$).

log

Prototype	`double log(double x);`
Description	Function returns the natural logarithm of x (i.e. $\log_e(x)$).

log10

Prototype	`double log10(double x);`
Description	Function returns the base-10 logarithm of x (i.e. $\log_{10}(x)$).

modf

Prototype	`double modf(double num, double *whole);`
Description	Function returns the signed fractional component of num, placing its whole number component into the variable pointed to by whole.

pow

Prototype	`double pow(double x, double y);`
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 `unsigned long`.

sin

Prototype	`double sin(double x);`
Description	Function returns the sine of x in radians. The return value is from -1 to 1.

sinh

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

sqrt

Prototype	`double sqrt(double num);`
Description	Function returns the non negative square root of num.

tan

Prototype	`double tan(double x);`
Description	Function returns the tangent of x in radians. The return value spans the allowed range of floating point in mikroC.

tan

Prototype	`double tanh(double x);`
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	`int abs(int num);`
Description	Function returns the absolute (i.e. positive) value of num.

atof

Prototype	`double atof(char *s)`
Description	Function converts the input string s into a double precision value, and returns the value. Input string s should conform to the floating point literal format, with an optional 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	`int atoi(char *s);`
Description	Function converts the input string `s` into an integer value, and returns the value. Input string `s` should consist exclusively of decimal digits, with an optional whitespace and a sign at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (this includes a null character).

atol

Prototype	`long atol(char *s)`
Description	Function converts the input string `s` into a long integer value, and returns the value. Input string `s` should consist exclusively of decimal digits, with an optional whitespace and a sign at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (this includes a null character).

div

Prototype	`div_t div(int numer, int denom);`
Description	Function computes the result of the division of the numerator `numer` by the denominator `denom`; function returns a structure of type `div_t` comprising quotient (`quot`) and remainder (`rem`).

ldiv

Prototype	`ldiv_t ldiv(long numer, long denom);`
Description	Function is similar to the `div` function, except that the arguments and the result structure members all have type `long`. Function computes the result of the division of the numerator `numer` by the denominator `denom`; function returns a structure of type `div_t` comprising quotient (`quot`) and remainder (`rem`).

labs

Prototype	`long labs(long num);`
Description	Function returns the absolute (i.e. positive) value of a long integer `num`.

max

Prototype	`int max(int a, int b);`
Description	Function returns greater of the two integers, `a` and `b`.

min

Prototype	`int min(int a, int b);`
Description	Function returns lower of the two integers, `a` and `b`.

rand

Prototype	`int rand(void);`
Description	Function returns a sequence of pseudo-random numbers between 0 and 32767. Function will always produce the same sequence of numbers unless `srand()` is called to seed the starting point.

srand

Prototype	`void srand(unsigned seed);`
Description	Function uses the seed as a starting point for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. No values are returned by this function.

xtoi

Prototype	`int xtoi(char *s);`
Description	Function converts the input string `s` consisting of hexadecimal digits into an integer value. Input parametes `s` should consist exclusively of hexadecimal digits, with an optional whitespace and a sign at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (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
strcat
strchr
strcmp
strcpy
strlen
strncat
strncpy
strspn
```

memcmp

Prototype	`int *memcmp(void *s1, void *s2, int n);`
Description	Function compares the first `n` characters of objects pointed to by `s1` and `s2`, and returns zero if the objects are equal, or returns a difference between the first differing characters (in a left-to-right evaluation). Accordingly, the result is greater than zero if the object pointed to by `s1` is greater than the object pointed to by `s2`, and vice versa.

memcmp

Prototype	`void *memcpy(void *s1, void *s2, int n);`
Description	Function copies n characters from the object pointed to by s2 into the object pointed to by s1. Objects may not overlap. Function returns the value of s1.

memmove

Prototype	`void *memmove(void *s1, void *s2, int n);`
Description	Function copies n characters from the object pointed to by s2 into the object pointed to by s1. Unlike with memcpy(), memory areas s1 and s2 may overlap. Function returns the value of s1.

memset

Prototype	`void *memset(void *s, int c, int n)`
Description	Function copies the value of character c (converted to char) into each of the first n characters of the object pointed by s. Function returns the value of s.

strcat

Prototype	`char *strcat(char *s1, char *s2);`
Description	Function appends the string s2 to the string s1, overwriting the null character at the end of s1. Then, a terminating null character is added to the result. Strings may not overlap, and s1 must have enough space to store the result. Function returns a resulting string s1.

strchr

Prototype	`char *strchr(char *s, char c);`
Description	Function locates the first occurrence of character `c` in the string `s`. Function returns a pointer to the `c`, or a null pointer if `c` does not occur in `s`. The terminating null character is considered to be a part of the string.

strcmp

Prototype	`char strcmp(char *s1, char *s2);`
Description	Function compares strings `s1` and `s2`, and returns zero if the strings are equal, or returns a difference between the first differing characters (in a left-to-right evaluation). Accordingly, the result is greater than zero if `s1` is greater than `s2`, and vice versa.

strcpy

Prototype	`char *strcpy(char *s1, char *s2);`
Description	Function copies the string `s2` into the string `s1`. If successful, function returns `s1`. The strings may not overlap.

strlen

Prototype	`unsigned strlen(char *s);`
Description	Function returns the length of the string `s` (the terminating null character does not count against string's length).

strncat

Prototype	`char *strncat(char *s1, char *s2, int n);`
Description	Function appends not more than n characters from the string s2 to s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result. Function returns s1.

strncpy

Prototype	`char *strncpy(char *s1, char *s2, int n);`
Description	Function copies not more than n characters from string s2 to s1. The strings may not overlap. If s2 is shorter than n characters, then s1 will be padded out with null characters to make up the difference. Function returns the resulting string s1.

strspn

Prototype	`int strspn(char *s1, char *s2);`
Description	Function returns the length of the maximum initial segment of s1 which consists entirely of characters from s2. The terminating null character character at the end of the string is not compared.

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	**void** ByteToStr(**unsigned short** number, **char** *output);
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	```unsigned short t = 24;```
	```char *txt;```
	```//...```
	```ByteToStr(t, txt); // txt is " 24" (one blank here)```

ShortToStr

Prototype	`void ShortToStr(short number, char *output);`
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	`short t = -24;`
	`char *txt;`
	`//...`
	`ByteToStr(t, txt);  // txt is " -24" (one blank here)`

WordToStr

Prototype	`void WordToStr(unsigned number, char *output);`
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	`unsigned t = 437;`
	`char *txt;`
	`//...`
	`WordToStr(t, txt);  // txt is "  437" (two blanks here)`

IntToStr

Prototype	`void IntToStr(int number, char *output);`
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	`int j = -4220;`
	`char *txt;`
	`//...`
	`IntToStr(j, txt);  // txt is " -4220" (one blank here)`

LongToStr

Prototype	`void LongToStr(long number, char *output);`
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	```long jj = -3700000;```
	 ```char *txt;```
	 ```//...```
	 ```LongToStr(jj, txt); // txt is " -3700000" (three blanks here)```

FloatToStr

Prototype	`void FloatToStr(float number, char *output);`
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.ddddd`; i.e. there will always be 5 digits following the dot. The `output` string must be at least 13 characters long.

Example	```float ff = -374.2;```
	 ```char *txt;```
	 ```//...```
	 ```FloatToStr(ff, txt); // txt is "-0.37420e3"```

Bcd2Dec

Prototype	`unsigned short Bcd2Dec(unsigned short bcdnum);`
Returns	Returns converted decimal value.
Description	Converts 8-bit BCD numeral `bcdnum` to its decimal equivalent.

Example	```unsigned short a;```
	 ```...```
	 ```a = Bcd2Dec(0x52); // equals 52```

Dec2Bcd

Prototype	`unsigned short Dec2Bcd(unsigned short decnum);`
Returns	Returns converted BCD value.
Description	Converts 8-bit decimal value `decnum` to BCD.

Example	`unsigned short a;`
	`...`
	`a = Dec2Bcd(52);  // equals 0x52`

Bcd2Dec16

Prototype	`unsigned Bcd2Dec16(unsigned bcdnum);`
Returns	Returns converted decimal value.
Description	Converts 16-bit BCD numeral `bcdnum` to its decimal equivalent.

Example	`unsigned a;`
	`...`
	`a = Bcd2Dec16(1234);    // equals 4660`

Dec2Bcd16

Prototype	`unsigned Dec2Bcd(unsigned decnum);`
Returns	Returns converted BCD value.
Description	Converts 16-bit decimal value `decnum` to BCD.

Example	`unsigned a;`
	`...`
	`a = Dec2Bcd16(4660);    // equals 1234`

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	`int SinE3(unsigned angle_deg);`
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 `angle_deg` 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	`res = SinE3(45);  // result is 707`

CosE3

Prototype	`int CosE3(unsigned angle_deg);`
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 `angle_deg` 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	`res = CosE3(196);  // result is -193`

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@mikroelektronika.co.yu
Website: www.mikroelektronika.co.yu