mikroElektronika
Development tools - Books - Compilers
www.mikroelektronika.co.yu

User's manual

C Compiler for Sensory RSC-4x microcontrollers

RSC-4x mikroC

Making it simple

Develop your applications quickly and easily with the world's most intuitive C compiler for Sensory RSC-4x microcontrollers.

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

With useful implemented tools, many practical code examples, broad set of built-in routines, and a comprehensive Help, RSC-4x mikroC makes a fast and reliable tool, which can satisfy needs of experienced engineers and beginners alike.

Reader's note

This manual covers mikroC version 1.0.0.7 and the related topics. Newer versions may contain changes without prior notice.

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

- Your operating system
- Version of RSC-4x mikroC
- Code sample
- Description of a bug

CONTACT US:
mikroElektronika

Voice:	+ 381 (11) 30 66 377, + 381 (11) 30 66 378
Fax:	+ 381 (11) 30 66 379
Web:	www.mikroe.com
E-mail:	office@mikroe.com

RSC 4x mikroC User's manual

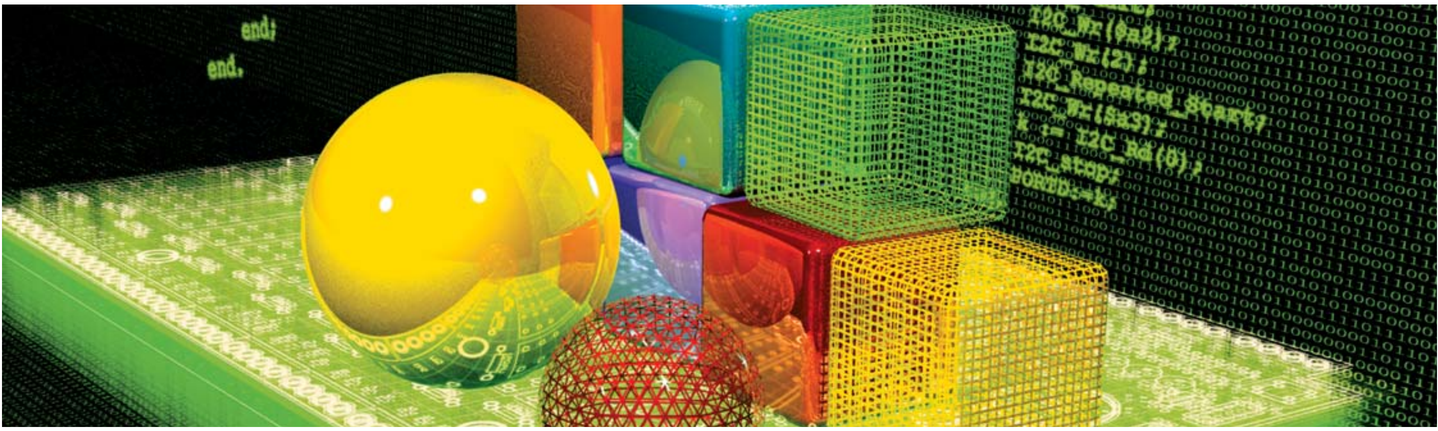
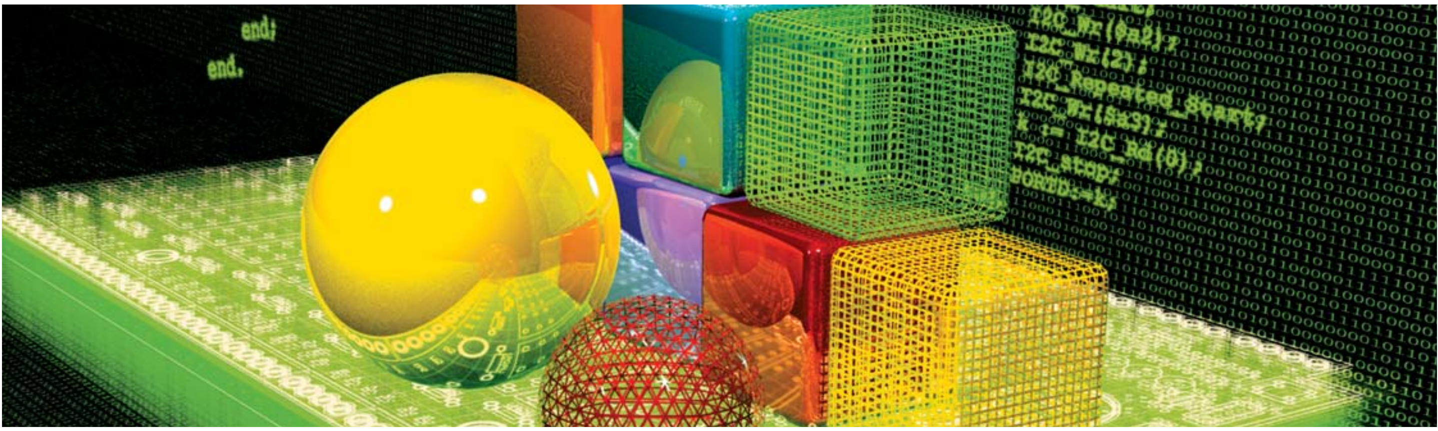


Table of Contents

CHAPTER 1

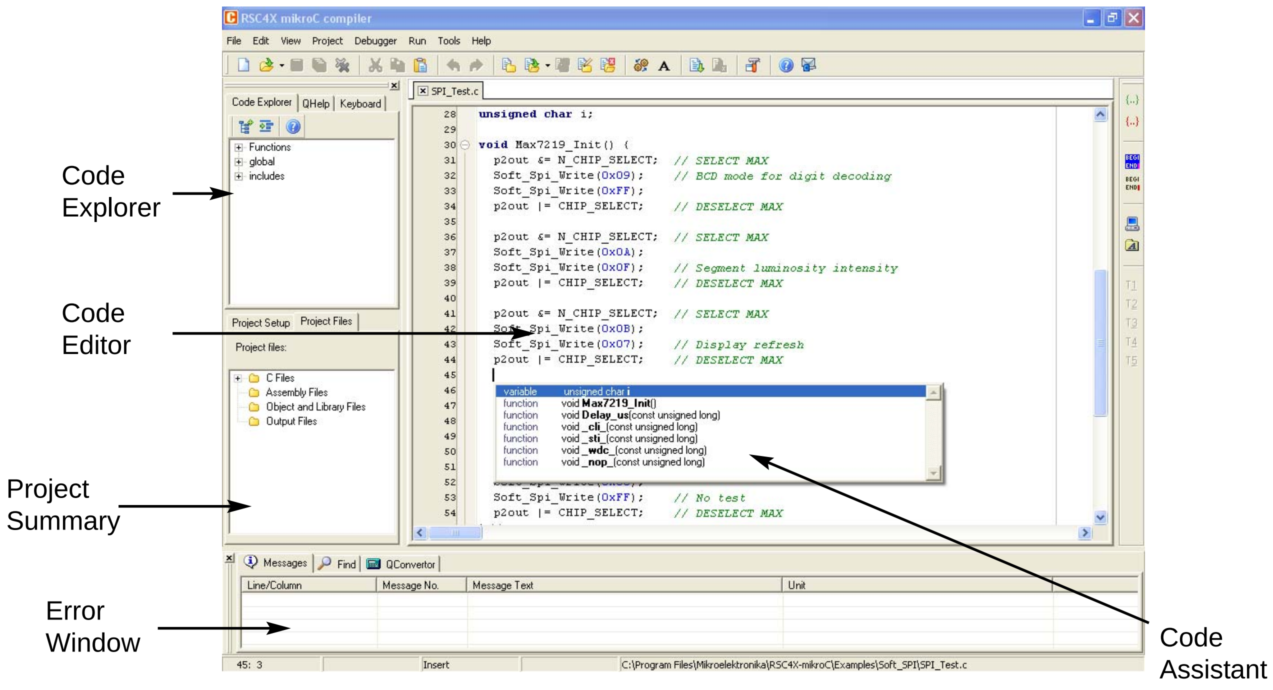


RSC-4x mikroC IDE

QUICK OVERVIEW

RSC-4x mikroC is a powerful, feature rich development tool for Sensory RSC-4x micros. It is designed to provide the customer with the easiest possible solution for developing applications for embedded systems, without compromising performance or control.

RSC-4x 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.



```
unsigned char i;

void Max7219_Init() {
  p2out &= N_CHIP_SELECT;  // SELECT MAX
  Soft_Spi_Write(0x09);    // BCD mode for digit decoding
  Soft_Spi_Write(0xFF);
  p2out |= CHIP_SELECT;    // DESELECT MAX

  p2out &= N_CHIP_SELECT;  // SELECT MAX
  Soft_Spi_Write(0x0A);
  Soft_Spi_Write(0x0F);    // Segment luminosity intensity
  p2out |= CHIP_SELECT;    // DESELECT MAX

  p2out &= N_CHIP_SELECT;  // SELECT MAX
  Soft_Spi_Write(0x0B);
  Soft_Spi_Write(0x07);    // Display refresh
  p2out |= CHIP_SELECT;    // DESELECT MAX

  Soft_Spi_Write(0xFF);    // No test
  p2out |= CHIP_SELECT;    // DESELECT MAX
```

```
variable    unsigned char i
function    void Max7219_Init()
function    void Delay_us(const unsigned long)
function    void _cli_(const unsigned long)
function    void _sti_(const unsigned long)
function    void _wdc_(const unsigned long)
function    void _nop_(const unsigned long)
```

RSC-4x mikroC allows you to quickly develop and deploy complex applications:

- Write your C source code using the highly advanced Code Editor

- Use the included RSC-4x 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.

- We have provided plenty of examples for you to expand, develop, and use as building bricks in your projects.

CODE EDITOR

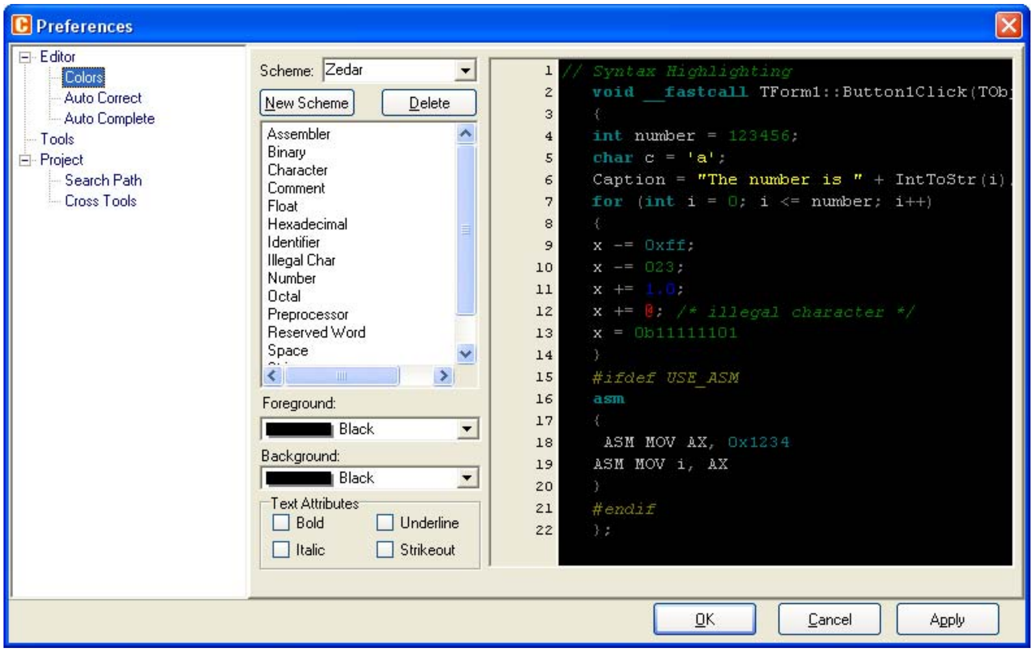
The Code Editor is an advanced text editor fashioned to satisfy the needs of professionals. General code editing is same as working with any standard text-editor, including familiar Copy, Paste, and Undo actions, common for Windows environment.

Advanced Editor features include:

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

Tools Icon.

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.



Scheme: Zedar

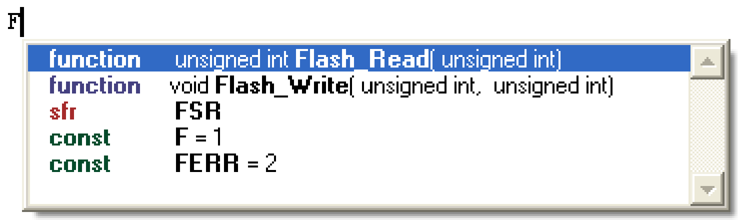
Foreground: Black

Background: Black

```
// Syntax Highlighting
void __fastcall TForm1::Button1Click(TOb
{
int number = 123456;
char c = 'a';
Caption = "The number is " + IntToStr(i)
for (int i = 0; i <= number; i++)
{
x -= 0xff;
x -= 023;
x += 1.0;
x += @; /* illegal character */
x = 0b11111101
}
#ifdef USE_ASM
asm
{
 ASM MOV AX, 0x1234
ASM MOV i, AX
}
#endif
};
```

Code Assistant [CTRL+SPACE]

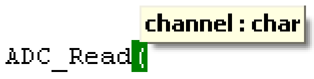
If you type a first few letter of a word and then press CTRL+SPACE, all the valid identifiers matching the letters you typed will be prompted in a floating panel (see the image). Now you can keep typing to narrow the choice, or you can select one from the list using the keyboard arrows and Enter.



```
F
function    unsigned int Flash_Read( unsigned int)
function    void Flash_Write( unsigned int,  unsigned int)
sfr         FSR
const       F = 1
const       FERR = 2
```

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.



```
channel : char
ADC_Read(
```

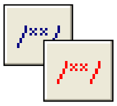
Code Template [CTRL+J]

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

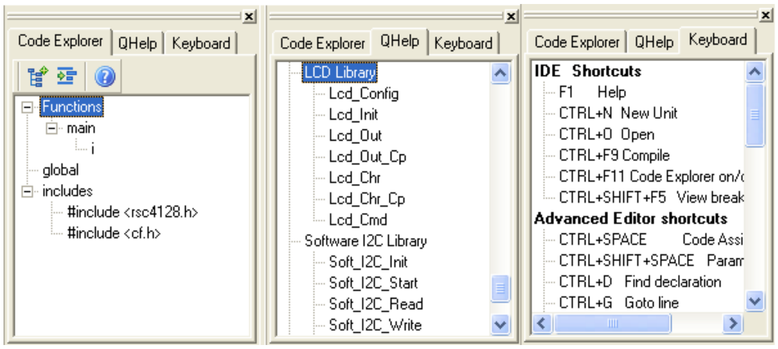
Goto Line option makes navigation through large code easier. Select Search > Goto Line from the drop-down menu, or use the shortcut CTRL+G.

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 RSC-4x mikroC.



```
Functions
  main
    i
global
includes
  #include <rsc4128.h>
  #include <cf.h>
```

```
LCD Library
  Lcd_Config
  Lcd_Init
  Lcd_Out
  Lcd_Out_Cp
  Lcd_Chr
  Lcd_Chr_Cp
  Lcd_Cmd
Software I2C Library
  Soft_I2C_Init
  Soft_I2C_Start
  Soft_I2C_Read
  Soft_I2C_Write
```

```
IDE  Shortcuts
  F1      Help
  CTRL+N  New Unit
  CTRL+O  Open
  CTRL+F9 Compile
  CTRL+F11 Code Explorer on/
  CTRL+SHIFT+F5  View break
Advanced Editor shortcuts
  CTRL+SPACE       Code Assi
  CTRL+SHIFT+SPACE  Param
  CTRL+D   Find declaration
  CTRL+G   Goto line
```

Simulator

Start Debugger

The source-level Debugger is not an integral component of RSC-4x mikroC development environment. RSC-4x mikroC compiler uses The PDS-SE simulator and PICE-SE emulator that have the same user interface and the same program developing and debugging capabilities.

After you have successfully compiled your project, you can simulate and debug the code in the external PDS-SE Phyton Simulator. To select the Simulator, select **Debugger › Select Tool › External Simulator** from the drop-down menu.

Beside external simulator, you can emulate the code.To choose external emulator from **Debugger › Select Tool › External Emulator.**

For more info refer to The PDS-SE simulator and PICE-SE emulator help or go to http://www.phyton.com/

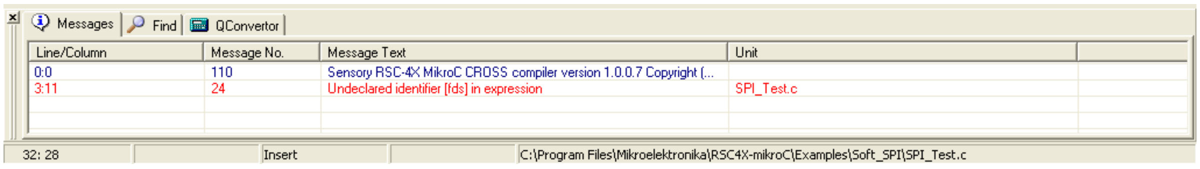
ERROR WINDOW

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.

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.



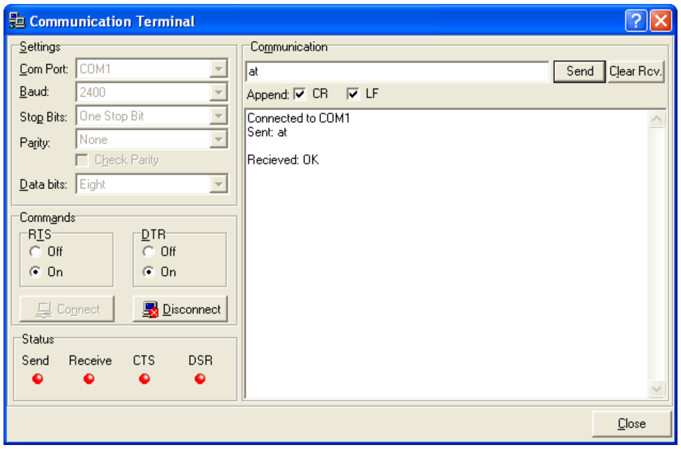
Line/Column	Message No.	Message Text	Unit
0:0	110	Sensory RSC-4X MikroC CROSS compiler version 1.0.0.7 Copyright (...	
3:11	24	Undeclared identifier [fds] in expression	SPI_Test.c

32: 28 Insert C:\Program Files\Mikroelektronika\RSC4X-mikroC\Examples\Soft_SPI\SPI_Test.c

INTEGRATED TOOLS

USART Terminal

RSC-4x 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.



Communication Terminal

Settings
Com Port: COM1
Baud: 2400
Stop Bits: One Stop Bit
Parity: None
Check Parity
Data bits: Eight

Commands
RTS: Off [✓ On]
DTR: Off [✓ On]

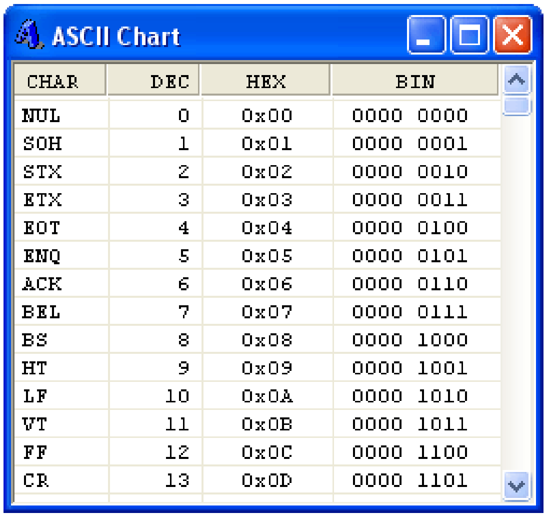
Communication
at
Append: [✓ CR] [✓ LF]

```
Connected to COM1
Sent: at

Recieved: OK
```

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.

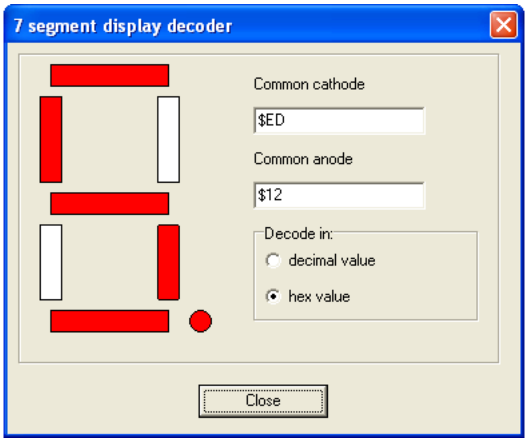


ASCII Chart

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

7 Segment Display Decoder

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



Common cathode: $ED

Common anode: $12

Decode in: ( ) decimal value [✓ hex value]

KEYBOARD SHORTCUTS

Below is the complete list of keyboard shortcuts available in RSC-4x 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+H	Replace
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

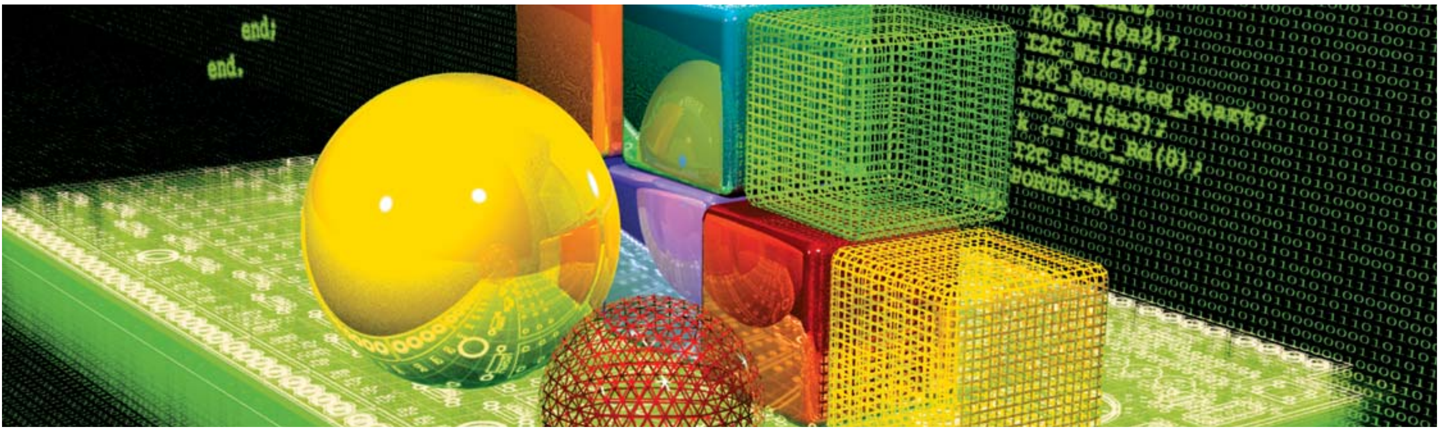
ALT+SELECT	Select columns

Debugger Shortcuts

F9	Debug

For other debugger shortcuts refer to the Phyton PDS-SE documentation.

CHAPTER 2



Building Applications

Creating applications in RSC-4x 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.

RSC-4x mikroC allows you to distribute your projects in as many files as you find appropriate. You can then share your RSC-4x mikroC compiled Libraries (`.mcl` files) with other developers without disclosing the source code.

PROJECTS

RSC-4x mikroC organizes applications into projects which consist of a single project file, with extension .psc, and one or more added files (C, assembler, and object/library files, with extensions .c, .mca, .mco/.mcl respectively). You can compile source files only if they are part of a project.

Before we start managing projects, we should introduce some terms that we'll be using throughout the documentation:

Source file

Source file is either C file with extension c or assembler file with the extension mca. Source files can be compiled/assembled only if they are part of a project. See Source Files for more information.

Project file

Project file is a file with extension psc which holds all the information regarding your project. The project file carries the following data:

- project name and optional description,
- project path,
- target device,
- memory model,
- device clock,
- wait states,
- memory address area settings,
- list of project files,
- search paths,
- include paths.

Project folder

Project folder is the folder in which the project file is stored. This folder is automatically added to the project's search path list.

Project

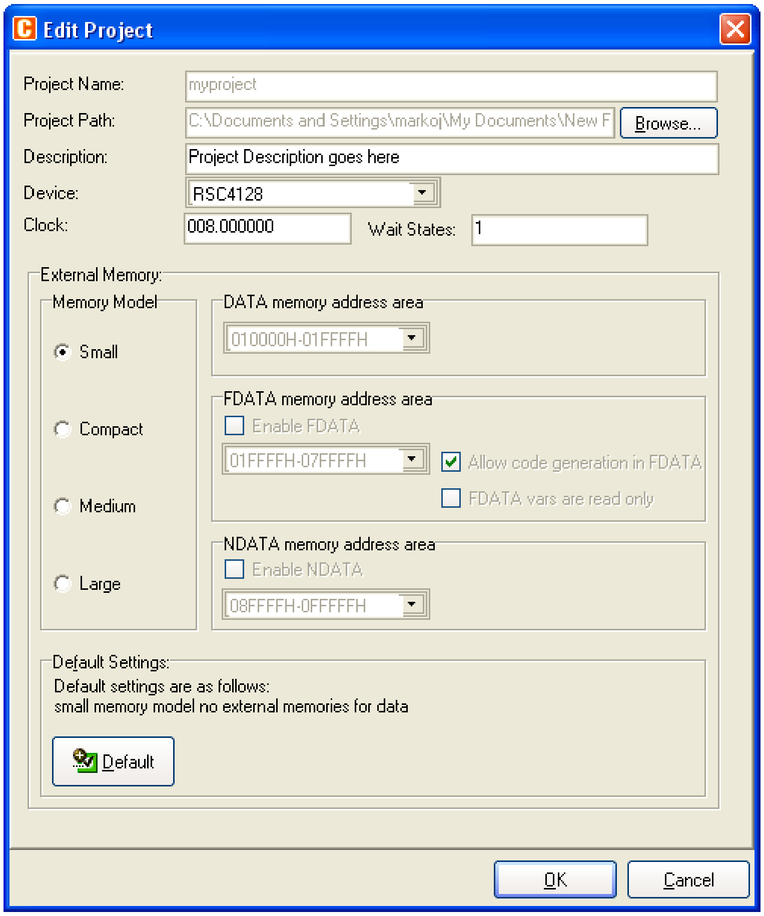
Project is a collection of all source files (C files, MCA files, header files) and object/library files (MCO/MCL files) relevant to the project. The project file binds them all together. Note that project added files have nothing to do with the preprocessor; see Add/Remove Files from Project below.

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 RSC-4x mikroC will create the appropriate project file. Also, an empty source file named after the project will be created by default.

In the following dialog, enter the project name and optional description.



Project Name: myproject
Project Path: C:\Documents and Settings\markoj\My Documents\New F

Description: Project Description goes here
Device: RSC4128
Clock: 008.000000
Wait States: 1

External Memory:
Memory Model
[✓ Small]
Compact
Medium
Large

DATA memory address area
010000H-01FFFFH

FDATA memory address area
☐ Enable FDATA
01FFFFH-07FFFFH
☑ Allow code generation in FDATA
☐ FDATA vars are read only

NDATA memory address area
☐ Enable NDATA
08FFFFH-0FFFFFH

Default Settings:
Default settings are as follows:
small memory model no external memories for data

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`) or or assembler files with extension .mca. The list of relevant source files is stored in the project file (extension `.psc`). 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`.

Search and Include Paths

Search paths are absolute or relative paths which tell the compiler where to look for files added to the project. Search paths are specific for each project and are stored in the project file.

Include paths are absolute or relative paths which tell the compiler where to look for header files included by the preprocessor. Include paths are specific for each project and are stored in the project file.

Do not confuse search paths and include paths. Although both lists are managed from the same window, they are totally separate categories. Search paths cover all files in the project, whereas include paths deal only with header files included by the preprocessor. There is a dropdown menu for choosing between absolute or relative paths for search and include paths.

Search Paths

Search paths (paths to source files) are specific for each project and are stored in the project file (.psc). To manage search paths, open **Tools › Options** from the drop-down menu and then select **Search Path**. The upper box in the dialog contains the list of project's search paths.

To add search path, click Add and browse to the location of the file that you wish to add. Click OK and you'll see your path added to the list. To remove the path, select it and click Remove.

The default search path is the folder in which the project is stored. Greyed out items represent invalid paths (i.e. non-existent folders). You can purge invalid paths with the Purge Invalid Paths button, which is useful when porting your projects to another computer.

The order of specified paths is important and determines the order in which the compiler looks for files. It will scan the list of locations from the top, and will stop as soon as the requested file is found.

Include Paths

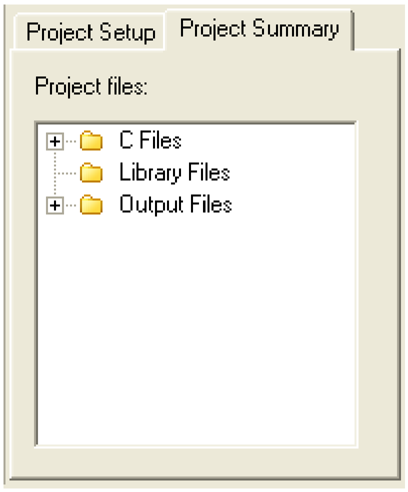
Header files are included by means of preprocessor directive `#include`. Do not use the preprocessor to include source files other than headers; instead, see Add/Remove Files from Project.

Include paths tell the compiler where to look for header files, when they are requested with the preprocessor directive `#include`. Include paths are specific for each project and are stored in the project file (.psc). To manage include paths, open **Tools › Options** from the drop-down menu and then select **Search Path**. The lower box in the dialog contains the list of project's include paths.

By default, the compiler looks for specified header files in RSC-4x mikroC installation folder › "include" folder. This is the default include path which cannot be changed. You can purge invalid paths with the **Purge Invalid Paths** button, which is useful when porting your projects to another computer.

Extended functionality of the Project Files tab

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



SOURCE FILES

In RSC-4x mikroC, source file is either C file with extension .c or assembler file with extension .mca. The list of source files relevant to your application is stored in the project file (extension .psc), along with other project information. A source file can only be compiled if it's a part of a project.

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

Managing Source Files

Creating a new source file

To create a new source file, do the following:

New File.

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. RSC-4x 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, RSC-4x mikroC will generate output files.

Upon successful compilation, RSC-4x mikroC will generate an assembler file for each C source file. These output files will be created in the approprate folder (folder which contains the original C source file).

View Assembly Icon.

After compiling your project in RSC-4x mikroC, you can click the View Assembler icon (**View › View Assembler**) to review the generated assembly code in a new tab. The option will display the assembler file (`.mca`) corresponding to the currently active C file in IDE. More information can be found in in the list file (`.lst`).

Then, the created assembler files and other assembler files you included in the project will be fed to the external assembler MCA-SE. Upon succesful assembling, an object file (extension `.mco`) will be created for each assembler file.

These object files together with object/library files you included in the project and default libraries will be fed to the external MCLINK linker, which will produce the hex file and the debug MCE file.

See Output Files for more information.

Compilation Options

You should check the compilation options before building your projects. Open Project › Build from the drop-down menu and then select Cross Tools.

Output Files

Output files are summarized in the table below:

Filetype	Filetype	Created By
.cp	File containing the preprocessor output. Preprocessor output files are helpful for debugging complex macros.	Preprocessor
.mca	Assembler source file generated from C source code. These files can be assembled with the MCA-SE assembler.	RSC-4x mikroC Compiler
.mco	Relocatable object file that contains relocatable object code. Relocatable object files may be linked to an absolute object file using the MCLINK linker.	MCA-SE Assembler
.lst	Listing file that contains the source text, generated assembler code, and all errors and warnings that were produced by the compiler. Listing file is optional, see Compilation Options.	MCA-SE Assembler
.mcl	Library file generated by the MCLIB Librarian from the relocatable object files.	MCLIB Librarian
.hex	Absolute Intel HEX file for CODE and CONST memory.	MCLINK linker
.dat	Absolute Intel HEX file for DATA memory.	MCLINK linker
.mce	Executable MCE-file in MicroCOSM-ST/Phyton format. This file contains executable code and debugging information and can be used for debugging with the Phyton IDE.	MCLINK linker

ERROR MESSAGES

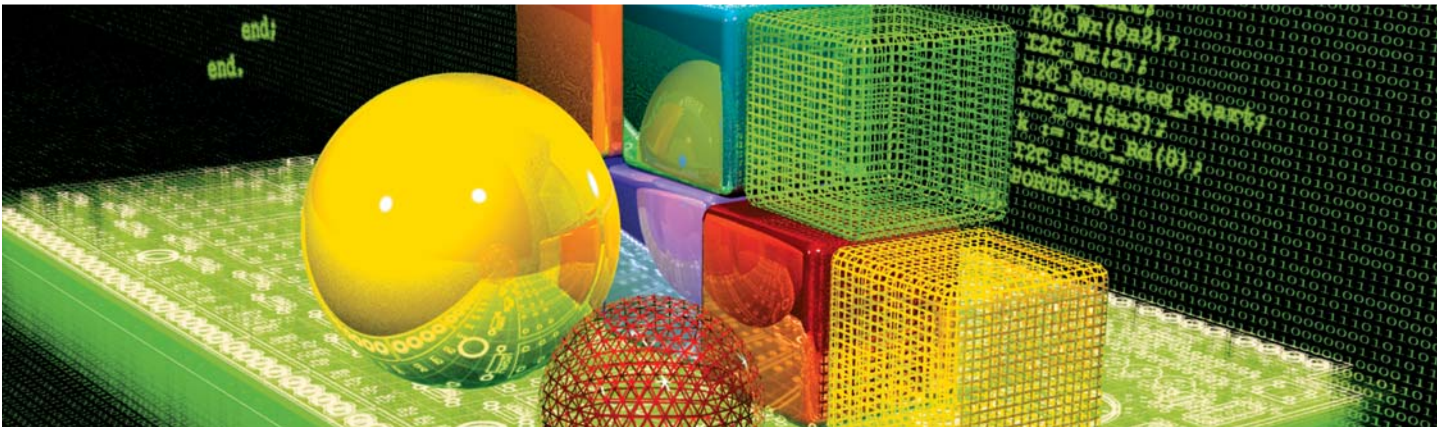
Error Messages

```
- 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 ineffecent code: func call in complex expression
- Ineffecent code: func call in complex expression
```

CHAPTER 3



RSC-4x mikroC Specifics

RSC-4x mikroC is a powerful, feature rich development tool for Sensory RSC-4x microcontrollers. It is designed to provide the programmer with the easiest possible solution for developing applications for embedded systems, without compromising performance or control.

Memory Types

RSC-4x mikroC compiler provides effective access to all RSC-4x chip memory types except the on-chip technology RAM:

On-chip SRAM. This is the register file consisting of the non-banked area and several memory banks. The 64 byte “window” is used to access the banked area. Only a part of the on-chip SRAM is available for the user’s program; the rest is reserved for the Sensory technology library.
ROM. Read-only or flash memory accessed with the RDR/WRC signals. On the logical level, ROM can contain executable code (code area) and constant data (data area). On the physical layer, ROM can be ether all internal or all external, but not both. The access methods for both external and internal ROM are the same.
External RAM. RSC-4x chips can be equipped with external memory of various physical types. Moreover, some applications have no external memory at all. The compiler distinguishes three types of external memory: non-persistent data memory (RAM), persistent writeable memory without predefined values (e.g. non-volatile RAM) and persistent writeable memory (e.g. flash).

On-chip SRAM

The following areas are available for the user’s C program:

0h – 0Bh (Global Register Area);
3Ah – 7Fh (Non-Banked Area);
80h – BFh (Banks 0h, 0Ah, 0Bh).
Other on-chip SRAM memory locations are reserved for using by the Sensory technology library.

The locations 3Ah – 7Fh and 80h – BFh (Bank 0h) are treated as a contiguous region used for allocation of C program data: static variables and statically placed in the compiled stack automatic variables, function parameters and temporaries.

Note that functions written in assembler should always clear the lower four bits of register BANK before passing control to a C function – either by call or return.

Banks 0Ah and 0Bh hold the dynamic stack. The dynamic stack is used as a call stack (i.e. holds return addresses of C functions); also it is used to hold the frames of previously invoked reentrant functions and contents of temporary registers during interrupt handling.

Global Register Area locations are used to pass the return address to the called C function, to return the function return value to the caller, and serve as temporaries for expression evaluation.

_tempArea	?R11 ?R10 ?R9 ?R8
	?R7
_returnAddress	?R6 ?R5 ?R4
_returnValue	?R3 ?R2 ?R1 ?R0

On-chip Technology RAM

On-chip technology RAM is used by Sensory technology library only. C program variables can not be allocated in this memory.

ROM

On the logical level, ROM can contain executable code (*code area*) and constant data (*data area*). On the physical layer, ROM can be ether all internal or all external, but not both. The access methods for both external and internal ROM are the same. Application may contain either one 64K ROM page (page 0), or several 64K pages up to 1M.

Static constant variables can be placed in ROM (read-only or flash memory) accessed with the RDR/WRC signals, i.e. CONST address space. The declaration of such variables should contain either memory type specifier `cdata` or qualifier `const` with the compilation option “const vars in CDATA area”.

Compiler distinguishes two types of `cdata` memory:

1. Located on page 0, i.e. in the range 0..0FFFFh
2. Located anywhere in the range 0..0FFFFFh

Memory type modifiers `near` and `far` can be used in `cdata` variable declarations. The allocation of a cdata variable declared without a memory type modifier will depend on the memory model. The size of a pointer to `near cdata` object is 2 bytes and the size of pointer to `far cdata` object is 3 bytes.

The `near cdata` objects are accessed by MOVC instruction. The `far cdata` objects are accessed by MOVX instruction with setting bits 0..3 of the ExtAddr register. The bit RW (bit 4 of ExtAddr) should be cleared when any C function is executed.

Note: The driver functions should restore the initial ExtAddr value upon return.

You can use the compilation option “constant CDATA variables” (Cross Tools) which will consider each variable declared with `cdata` memory type specifier to be `const` also (i.e. on any attempt to modify it the compiler will generate the error message).
If the CONST address space represents writeable flash memory, uncheck the option “constant CDATA variables” to place non-constant objects there.

External RAM

RSC-4x chips can be equipped with external memory of various physical types. Moreover, some applications have no external memory at all.

The compiler distinguishes three types of external memory:

- Non-persistent **data** memory (RAM) represented by address space DATA. Variables located in this memory are initialized during C program startup.

- Persistent writeable memory without predefined values (e.g. non-volatile RAM) represented by address space NDATA. Variables located in this memory are not initialized at startup; no initializers are allowed with **ndata** variable declarations.

- Persistent writeable memory with predefined values (e.g. flash) represented by address space FDATA. Variables located in this memory are not initialized at start-up, instead, the initial values are put directly into the memory locations for **fdata** variables. The linker creates a separate HEX file for address space FDATA.

To allocate a variable in the external memory the user should declare this variable with one of the memory type specifiers – **data, ndata** or **fdata**. Variables declared with such specifiers are put in the corresponding relocatable segments. Their address size is 3 bytes.

Since the compiler does not know what specific method should be implemented to **access data/ndata/fdata** external memory, three sets of special driver functions are used to access the external memory locations. Templates of the special driver functions written in assembler are included in the compiler distribution package. The user can modify their implementation, if necessary.

If **fdata** represents memory with no write access then the user should set the option “fdata variables are read only” in the Edit Project dialog (drop-down menu **Project › Edit Project**). In this case, the compiler will treat any **fdata** variable as if declared with the type qualifier **const**, and will generate the error message on any attempt to modify it. Individual **fdata** variables can be declared as **const** explicitly.

Memory Models

The memory model determines the following C program features:

- Maximum code size (and code generated for the function calls).
- Availability of external RAM.

Memory model	Code placed on page 0 only	cdata modifier by default	data / ndata / fdata allowed
SMALL	yes	near	no
COMPACT	yes	near	yes
MEDIUM	no	far	no
LARGE	no	far	yes

The memory model is specified in the Edit Project dialog (drop-down menu **Project › Edit Project**).

Address Spaces

Address space	Allocation	Description
CODE	code	Instructions
CONST	const	Constants, string literals and initializers
REG	reg	Static register variables, compiled stack
DATA	data	Static **data** variables
NDATA	ndata	Static **ndata** variables
FDATA	fdata	Static **fdata** variables

The address range of CODE address space for SMALL and COMPACT memory models is 0..0FFFFh, for MEDIUM and LARGE memory models is 0..1FFFFh.

By default, DATA address space range is 0..0FFFFFh. You can set the appropriate address range from the Edit Project dialog (drop-down menu **Project › Edit Project**).

If there is no FDATA (or NDATA) memory, this should be specified in the Edit Project dialog (drop-down menu **Project › Edit Project**). In this case, the compiler will check that there are no **fdata** (or **ndata**) variables in the C program.

Otherwise, the user should define the address space FDATA (with allocation **data**) and set the appropriate address range, again from the Edit Project dialog. The same must be done for NDATA when non-volatile RAM is used.

Absolute Memory Locations

There are built-in definitions for several function-like macros to access absolute memory locations.

Accessing Absolute Memory Locations

```
__BYTE_AT__
__WORD_AT__
__DWORD_AT__
```

These macros allow the developer to access individual bytes, words and double words in the on-chip SRAM and SFR's. The main purpose of using these macros is to provide access to the on-chip SRAM memory locations defined outside the C-written modules.

These built-in macros can be considered as if the following macros were defined in the program source text:

```
#define __BYTE_AT__(addr)      (*(volatile unsigned char *)(addr))
#define __WORD_AT__(addr)      (*(volatile unsigned int  *)(addr))
#define __DWORD_AT__(addr)     (*(volatile unsigned long *)(addr))
```

Note: The **volatile** keyword prevents the compiler from making any optimizations.

Example:

```
#define MyByte   (__BYTE_AT__(0x49))
#define MyWord   (__WORD_AT__(0x4A))
#define MyDword  (__DWORD_AT__(0x4C))

int a;

void main(void) {  /* some meaningless actions */
/* only for illustration */
  a = MyByte * 10;
  MyWord = a;
  MyDword = a + 1;
}
```

If the built-in macros `__BYTE_AT__`, `__WORD_AT__` and `__DWORD_AT__` are used inside inline assembler blocks, these macros are directly expanded as the values of their arguments (the typecast-containing expressions may not be used inside built-in assembler blocks).

Language Extensions

The following topics provide details of the RSC-4x mikroC language extensions:

ANSI Compliance
Types Specifics
Functions Specifics
Linker Directives
Inline Assembler
Pragma Directives

ANSI Compilance

RSC-4x mikroC compiler was built according to ISO/IEC 9899-1990 and ANSI/ISO 98991990 standards (except for the extensions and restrictions described below). These standards are practically identical, therefore the word “standard” will be used for referencing them both.

The standard defines two forms of conforming compiler implementations:

1. A *conforming hosted implementatio*n must correctly compile any program, which strictly conforms to the language standard.
2. A *conforming freestanding implementation* must correctly compile any strictly conforming program, in which the use of the standard C library features is confined to the contents of the standard headers: <float.h>, <limits.h>, <stdarg.h>, <stddef.h>.

It is impossible to create a strictly conforming library for single-chip microcontrollers. For example, it is not possible to implement all file access functions. Thus, the RSC-4x mikroC is not a conforming hosted implementation.

However, as the standard headers <float.h>, <limits.h>, <stdarg.h>, <stddef.h> and related library functions are fully implemented in accordance with the standard, the RSC-4x mikroC is a conforming freestanding implementation with the exception for a few hardware-specific restrictions.

Extensions

The standard extensions implemented in RSC-4x mikroC can be divided in two groups:

- Hardware-specific extensions. This includes memory type specifiers and modifiers.

- Common extensions, such as integer binary constants and C++ style comments.

Restrictions

RSC-4x mikroC has the following restrictions compared to the standard:

- **double** and **long** double types representation;

- Function reentrancy support. Functions are not considered as reentrant, by default. A function can be declared reentrant (called either by recursion or both from the main thread and interrupt handler) by the special `#pragma reentrant`. Some additional restrictions are applied to functionality of reentrant functions.

Types Specifics

Memory Type Specifiers and Modifiers

The user can select the memory type for placing a variable using a memory type specifier. Also, *memory type specifier* can be used in pointer type declaration if the pointer holds addresses in the specific memory.

Additionally, *memory type modifiers* can be used to specify the memory range where **cdata** variables are to be placed in.

The memory type specifiers and modifiers can be used in variable declarations, pointer type declarations of variables, pointers and type definitions (with typedef). Type constructed using a memory type specifier/modifier is called a **specialized type**.

Note: The memory type specifiers and modifiers are additional keywords.

Note: For standard type modifiers see Storage Classes and Type Qualifiers.

Specifiers

Specifier	Description
data	Variables in the external RAM initialized during startup. Special driver functions are used to access these variables.
ndata	Variables in the external RAM not initialized during startup. Special driver functions are used to access these variables.
fdata	Variables in the external FLASH not initialized during startup. Instead, the initial values are put in memory locations "directly" - the linker creates a separate HEX file with initial values. zero is used. Special driver functions are used to access these variables.
cdata	Constants placed in ROM. These constants are accessed by the MOVC or MOVX instructions in accordance with the given memory type modifier (see below). zero is used.

Modifiers

Modifier	Description
near	`A cdata variable that will be placed in the page 0 of ROM; it has a 16-bit address.`
far	`A cdata variable that will be placed in any ROM page; it has a 20-bit address.`

Variables Allocation

Memory type specifiers can be used in declaration of variables with static storage duration only. It is not allowed to use specifiers in declaration of automatic variables, functions, function parameters, structure and union members.

If a static or external variable is declared as **const** with no explicit memory type specifier and the compiler option “const vars in CDATA area” (Cross Tools) is checked, the variable is deemed to have the **cdata** specifier. Variables without explicit memory type specifiers are allocated in the on-chip SRAM. Therefore the **register** keyword in declaration of an automatic variable or a function parameter is unnecessary and is ignored.

Memory type modifiers can be used with **cdata** variables only. The allocation of a **cdata** variable declared with the **cdata** specifier and without any memory type modifiers will depend on the memory model.

Examples – Using the memory type specifiers:

```
data int n;    /* variable allocated in "data" memory */
ndata char c; /* variable allocated in "ndata" memory */
long k;        /* variable allocated in on-chip SRAM */
```

Pointer Types

Memory type specifier can be used in a pointer type declaration. In this case, the compiler will treat the pointer value as an address in the specific memory and will generate an appropriate instruction sequence when the pointer needs to be dereferenced. A pointer declared without a memory type specifier is treated as the pointer to the on-chip SRAM. The size of pointer type is determined by the memory type specifier (or absence of the specifier).

Note the difference between two pointer declarations:

```
/* variable allocated in the on-chip SRAM and points into "data" */
data int * p;

/* variable allocated in "data" and points into on-chip SRAM */
int * data q;
```

Qualifier `const`

If a static or external variable is declared as **const** with no explicit memory type specifier and the compiler option “const vars in CDATA area” is checked, the variable is deemed to have the **cdata** specifier. If this option is unchecked, such variables are allocated in the on-chip RAM (as variables without explicit memory type specifiers).

Pointer Specifics

This topic is an overview of specialized pointer types, constructed with memory type specifiers and modifiers. For more details on pointers syntax and usage see Pointers.

Data type	Bytes	Aligment	Comment
`*`	1	none	`Pointer to the on-chip SRAM`
`data *`	3	none	`Pointer to the external RAM. If dereferenced the special driver function call is generated.`
`fdata *`	3	none	`Pointer to the external RAM. If dereferenced the special driver function call is generated.`
`ndata *`	3	none	`Pointer to the external RAM. If dereferenced the special driver function call is generated.`
`near cdata *`	2	word	`Pointer to ROM page 0.`
`far cdata *`	3	word	`Pointer to ROM. The third byte contains 4 higher bits of the address.`
`pointer to function, SMALL/COMPACT memory model`	3	word	`Pointer to ROM. The third byte contains 4 higher bits of the address; the highest byte is not used.`
`pointer to function, MEDIUM/LARGE memory model`	4	word	`Pointer to ROM. The third byte contains 4 higher bits of the address; the highest byte is not used.`

Strings Specifics

By default, the compiler will store string constants (string literals) according to the destination's type. See Memory Type Specifiers and Modifiers and Memory Models.

If the option "store string literals in CDATA area" in Cross Tools (**Tools › Options**) is enabled, all string constants will be linked in the CDATA area.

Specialized Types Conversions

Two specialized types are compatible if their memory type specifiers (and modifiers) are equivalent and they are compatible in terms of ANSI standard.

The implicit conversion from a specialized type to a corresponding non-specialized type is always allowed. Backward conversion (from a non-specialized to a specialized version) is not allowed because only lvalue have the specialized type (the result of type conversion is rvalue).

Pointers to Specialized Types

The following explicit and implicit conversions of pointers to specialized types are allowed (if the corresponding conversion of pointers to non-specialized types is allowed by C standard):

Implicit conversion of pointers to specialized types:

Data Type	Result Type
fdata *	ndata *
ndata *	data *
near cdata *	far cdata *

Explicit conversion of pointers to specialized types:

Data Type	Result Type
fdata *	data *
fdata *	ndata *
data *	fdata *
ndata *	fdata *
far cdata *	near cdata *

Since any pointer declared without an explicit memory type specifier is considered to be a pointer into the on-chip SRAM, any explicit or implicit conversion of a pointer to specialized type into a pointer to non-specialized type is not allowed. All reverse conversions (from pointer to non-specialized type into pointer to specialized type) are not allowed as well.

If some implicit conversion is allowed, it can be written in the explicit form also.

The pointer to a function may not be converted to **void*** and back (neither explicitly, nor implicitly).

Functions Specifics

Following topics cover the implementation specifics of functions:

- Interrupt Handlers
- Reentrant Functions
- Indirect Function Calls
- Startup Functions
- Prologue/Epilogue Functions
- Monitor Functions

Interrupt Handlers

The following directive

```
#pragma interrupt <interrupt_number> <function_name>
```

lets the compiler know that the function *function_name* is an interrupt handler for the interrupt *with interrupt_number*.

The interrupt handler must be declared as void, have no parameters, and must be defined in the same source file where the corresponding *#pragma interrupt* directive appears.

The compiler uses a special mechanism to set the interrupt handler vector for the declared interrupt handler. Also, compiler adds a special prolog and epilog to the interrupt handler code (the prolog saves the necessary registers on the dynamic stack, the epilog restores them). See Interrupt Handling for the implementation details.

Note: Two or more interrupt handlers may not be assigned to one interrupt. However, a function may be used as a handler for several interrupts.

The function declared as an interrupt handler should not be called in the user's program, either directly or indirectly (by a pointer to function). The compiler generates the warning message if an interrupt handling function is called or its address is taken.

Example:

```
#pragma interrupt 00 timer1_isr
...
void timer1_isr(void) {
  /* user code */
}
```

Interrupt numbers and associated vectors:

Interrupt #	Address	Interrupt source
0	04h	Timer 1 overflow
1	08h	Timer 2 overflow
2	0Ch	Positive edge of Filter End Marker
3	10h	Positive edge of P00
4	14h	Timer 3 overflow
5	18h	Block End
6	1Ch	Positive edge of P02

Note: SFR definition files `RSC4xxx.h` contain definitions for SFR bits. Thus, the developer can access particular bits in SFR. For example, to disable Interrupt #1 (Timer 2 overflow) you can write the following:

```
IMR1 = 0; /* produces code: AND IMR, #0FDh */
```

You should not clear the bits in the IRQ register using bit names defined in RSC4xxx.h (the IRQ bits should be cleared by an explicit MOV only rather than AND). For example, in order to clear the Interrupt #1 request bit, you should write the following:

```
IRQ = 0xFD; /* produces code: MOV IRQ, #0FDh */
```

Reentrant Functions

The function is *reentrant* if it may be called either recursively or both from the main thread and from an interrupt handler. Functions are not considered reentrant, by default. The function can be declared reentrant with the special `#pragma` directive:

```
#pragma reentrant <func_name>
```

If the function is reentrant, the corresponding `#pragma` must be placed in all source modules where this function is implemented, declared or called (the easiest is to place `#pragma reentrant` in the header file containing the function declaration).

The reentrant function call requires additional overhead. Moreover, the support for reentrant functions results in the following restrictions that violate the C standard:

- Total size of reentrant function parameters must not exceed 4 bytes, since the _tempArea field of GlobalRegisterArea is used for passing the reentrant function parameters.
- Reentrant functions with ellipsis (variable number of arguments) are not supported.

A reentrant function return value cannot have a structure type, i.e. reentrant function cannot return a structure (but can return a pointer to structure).

- Local objects (automatic variables and actual function parameters) of the previous invocations of reentrant functions are inaccessible from the current reentrant function invocation (because they can be overwritten by the current reentrant function frame, see Compiled Stack for details). Also, only local objects of the last invocation of reentrant function can be accessed from non-reentrant functions.

Example:

```
#pragma reentrant func
...
void func(char counter) {
   ...
   if (++counter < 10)
   func(counter);
   ...
}
```

Note: Local objects of all non-reentrant functions' previous invocations can be accessed from both reentrant and non-reentrant function invocations.

The linker checks if a function may possibly be activated two or more times simultaneously in the user's program. If it encounters such a function with a non-empty frame (the function has automatic variables or parameters) and the function is not declared as reentrant, the linker generates the error message.

Indirect Function Calls

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

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

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

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

Startup Functions

The user is enabled to execute several custom initialization functions at startup. These functions are executed after running the service startup code and before calling the `main()` function. The order of startup function invocation is based on the user-defined priorities.

The following special directive:

```
#pragma startup <priority> <function_name>
```

can be used to inform the RSC-4x mikroC compiler that the function `function_name` must be executed during startup with the specified priority.

The function `function_name` should return no value and have no parameters. Declaration of the function should be placed before the `#pragma startup` directive for that function. External function declaration is allowed.

Priority must be a constant expression ranging from 0 to 255. Smaller values correspond to higher priorities and sooner execution. The value 0 sets the highest priority level (the earliest execution), the value 255 sets the lowest priority (the latest execution).

Note: If the same priority value is used in several `#pragma startup` directives, the corresponding functions will be executed in an arbitrary order.

Priority levels from 0 to 15 and from 241 to 255 are reserved for using in the RSC-4x mikroC libraries, so it is not recommended to use them.

Example:

```
extern void ExtFunc(void);   /* an external function declaration */

static void LocalFunc(void);/* a local function prototype declara-
tion */


/* then #pragma startup can be used */
#pragma startup 20 ExtFunc
#pragma startup 21 LocalFunc


/* the local function definition */
static void LocalFunc(void) {
  /* function body */
}
```

Monitor Functions

The following directive:

```
#pragma monitor <function_name>
```

declares the function `function_name` as monitor function. All interrupts are disabled when this function is executed.

Prologue/Epilogue Functions

The `#pragma` directives

```
#pragma prolog
#pragma epilog
```

enable the user to customize prologs and epilogs of the functions in a translation unit. This is useful primarily for debugging.

`#pragma prolog` informs RSC-4x mikroC that the function prolog_function should be called from the prologs of all functions in the file (except any prolog and epilog functions). Call is performed immediately after saving the return address on the dynamic stack.

`#pragma epilog` informs the compiler that the function epilog_function should be called from the epilogs of all functions in the file (except any prolog and epilog functions). Call is performed immediately before restoring the return address from the dynamic stack.

`#pragma prolog` can appear more than once in a given file. In this case, the order of the prolog function calls is determined by the arrangement of the `#pragma prolog` directives in source file.

Function `prolog_function` should return no value and have no parameters. The declaration of the function should be placed before the `#pragma prolog` directive for that function. External function declaration is allowed.

The same rules apply to `epilog_function` and `#pragma epilog`.

Example:

```
extern void ExtPrologFunc(void);
#pragma prolog ExtPrologFunc;

int MyFunction(int a) {
  /* call to ExtPrologFunc here    */
  return a + 1;
} /* call to LocalEpilogFunc here */

static void LocalEpilogFunc(void) {
  /* function body */
}
#pragma epilog LocalEpilogFunc
```

Inline Assembler

According to standard, C should allow embedding assembler in the source code by means of **asm** declaration. In addition, RSC-4x mikroC also supports declarations **_asm** and **__asm** which have the same meaning.

Group assembler instructions by the asm keyword (or **_asm** or **__asm**):

```
asm {
  block of assembler instructions
}
```

Assembler one-line comments starting with semicolon are allowed in the embedded assembly code; C/C++ style comments are also allowed.

Embedding assembler with pragma

For the sake of backward compatibility, RSC-4x mikroC compiler supports another method for embedding assembler in C source code.

Note: Preferred method is using the asm declaration as described above.

```
/* Single-line format: */
#pragma asm <machine_instruction0> [; <machine_instruction1> ] ...
```

or:

```
/* Block format: */
#pragma asm
<machine_instruction0> [; <machine_instruction1> ] ...
...
#pragma endasm
```

In the single-line format, instructions should follow the `#pragma asm` directive in the same line. You can specify more than one instruction in one directive. Instructions should be separated by semicolons.

In the block format, the `#pragma asm` directive indicates the beginning of assembler block, and `#pragma endasm` directive indicates the end of the block. You can write one or more instructions in each line of the block. Multiple instructions in one line should be separated by semicolons.

You can also use function-like macros in the assembly language blocks and lines. Inside an assembler block, the preprocessor acts in a regular way and processes the assembler block properly. Therefore, in the built-in assembler you can use C-style macros and other preprocessing directives. The `#` and `##` operators may not be used.

Pragma Directives

Data Type	Result Type
#pragma asm/endasm	Built-in assembler
#pragma funcall	Enumerate indirectly called functions
#pragma interrupt	Writing interrupt service routines in C
#pragma monitor	Declaration of a monitor function
#pragma prolog/epilog	Customizing prolog and epilog
#pragma reentrant	Declare a reentrant fuction
#pragma startup	Execution of a function at startup

Implementation Details

Compiled Stack

The RSC-4x architecture has no indexed addressing mode. For this reason, allocation of C-function arguments and local variables at fixed absolute memory locations is much more effective than simulation of a runtime stack.

The linker can overlay function parameters and local variables for functions that cannot be activated simultaneously to extend the amount of available REG address space. This scheme is called compiled stack.

The size of function frame that consists of the local variable and parameter sections is evaluated by the compiler. The linker analyzes all direct and indirect function calls and builds the call graph. Then the linker places the function frames at fixed addresses in such a way that frames of functions which cannot be activated simultaneously overlay.

The linker places the frames of non-reentrant functions in the non-banked area and Bank 0 of on-chip SRAM as it was described above. The frames of reentrant function are placed by the linker beginning from the same start address in the non-banked area of on-chip SRAM.

Reentrant function, in its prolog, pushes on the dynamic stack the contents of memory locations to be used as the frame of that function (locations which hold locals from the previous invocation of another or the same reentrant function). Before returning, the reentrant function epilog restores the contents of memory locations saved in the prolog.

Because all reentrant function frames are allocated starting from the same address and overlap, the local objects of the previous reentrant functions calls are inaccessible in the current reentrant function invocation.

Note: The RSC-4x mikroC compiler does not support automatic variables and function parameters allocation in ROM or external RAM (see also memory type specifiers).

In the SMALL and COMPACT memory models, the called function returns control to the caller by:

```
JMPR @?R6
```

in the MEDIUM and LARGE memory models – through a jump to the `__RETURN` label:

```
JMP .LWRD(__RETURN)
```

where the CB1 bit value of ExtAdd is restored and control returns to the caller by JMPR instruction in the same way as for the SMALL and COMPACT models. That code is duplicated in both program memory banks.

Passing Return Value

Value returned by a function is loaded into the `_returnValue` field of the GlobalRegisterArea. The number of bytes occupied by the return value depends on its type.

If a called function returns a value of structure (or union) type, the caller reserves a memory location in its frame for the structure and passes the pointer to that location to the called function as the first parameter (before all explicit parameters). The called function stores the structure to be returned in the reserved memory location and returns the pointer to the structure through the `_returnValue` field.

Note: Reentrant function may not return values of structure types.

Interrupt Handling

Runtime library contains six library modules with code for default interrupt vectors. If there is a C-written interrupt handler for interrupt #N, the compiler will generate code for such interrupt vector (jump to interrupt handler code). If there is no C-written interrupt handler for an interrupt then a corresponding default interrupt vector will be linked.

In its prolog, the function declared as an interrupt handler copies the state of the register `ExtAdd` to the special location `?ExtAddSav`, pushes on the dynamic stack:

contents of GlobalRegisterArea,
variable `?CurRegBank`,
variable `?ExtAddSav`,
value of register `BANK`,
variable `?ExtAdd` (only for MEDIUM and LARGE memory models);
and then sets the value of `?CurRegBank` to 0Bh.

The interrupt handler epilog pops the saved values from the dynamic stack.

MEDIUM and LARGE Memory Models

In the MEDIUM and LARGE memory models the code for functions declared as interrupt handlers is always placed in bank 0 of the program memory. Thus, if there is at least one C-written interrupt handler, the interrupt handling dispatcher will be linked. The dispatcher copies `ExtAdd` to `?ExtAddSav`, sets `ExtAdd` to 0 (bank 0 of program memory) and transfers control to C-written interrupt handler. C-written interrupt handler returns control to the dispatcher by `JMP` instruction, the dispatcher restores `ExtAdd` from `?ExtAddSav` and returns control by `IRET`. If there are no C-written interrupt handlers, the dispatcher module will not be linked.

In the MEDIUM and LARGE memory models, interrupt vectors and interrupt handling dispatcher will be placed in both code pages.

Note: Although the dispatcher contains entry points for all interrupts, if there is no C-written interrupt handler for an interrupt then the dispatcher will not be used for that interrupt because the default interrupt vector will be linked.

Runtime Library

This chapter gives an overview of the runtime library functions implementation details.

The runtime library contains the following code that is implicitly called in code generated by the compiler:

- Driver functions for external data memory access
- Functions for integer multiplication/division operations
- Floating point library functions
- Code for calling support in MEDIUM and LARGE memory models (see Calling Convention)
- Functions for dynamic stack operations (return address, GlobalRegisterArea and reentrant function frame save/restore functions)
- Interrupt handling support

In SMALL and COMPACT memory models, the `CALL` instruction is used to pass control to all the runtime library code. In MEDIUM and LARGE memory models, the control is transferred to driver functions, integer and floating point functions through the resident code; the code for other functions is placed in both code pages. The resident code copies the `ExtAdd` register value to the `?ExtAdd` variable and sets/clears CB bit of `ExtAdd` to the number of runtime library code page. By default, the runtime library code is placed in page 0. Change the page number in startup module to relocate the runtime library code to the other page.

The arguments and return value are passed through the `_tempArea` and `_returnValue` fields of GlobalRegisterArea respectively. The runtime library code does not modify the 3-byte field `_returnAddress` of `GlobalRegisterArea` (except for the function that restore the return address from the dynamic stack).

Some of the runtime library functions (floating point library, integer multiplication/division) use up to 8 bytes of temporary variables. Because each of these functions can be called both from the main thread and an interrupt handler, there are two copies of temporary registers allocated in banks 0Ah and 0Bh of the on-chip SRAM.

The area in Bank 0Ah is used for main thread, the area in Bank 0Bh – for interrupt handlers. The 1-byte variable `?CurRegBank` is allocated in non-banked area of the on-chip SRAM, and each of the system functions that use temporary variables sets the lowest 4 bits of BANK register from `?CurRegBank` before executing and clears the bits before returning. `?CurRegBank` is initialized with value 0Ah at startup. Interrupt handler writes the value 0Bh to `?CurRegBank` in its prologue and restores the original value of `?CurRegBank` in its epilogue.

Note: if an interrupt handler invokes a system function that uses temporary variables (e.g., a floating point function) and then it is interrupted by another interrupt handler, which, in turn, invokes a system function that also uses temporary variables, the behavior is undefined.

CSTARTUP and SYSLIB Modules

These modules contain segment declarations (and segment allocation directives for linker), interrupt vector declarations, special register declarations (GlobalRegisterArea, service registers for integer and floating point system libraries, dynamic stack support) and startup code. The C startup code is invoked when the user application is started. This code includes the following operations:

- Initialization of: dynamic stack, special registers, external memory access
- Initialization of the static variables allocated in the on-chip SRAM and external RAM
- Invocation of the startup functions in the order based on the priorities;
- Calling the user `main()` function.

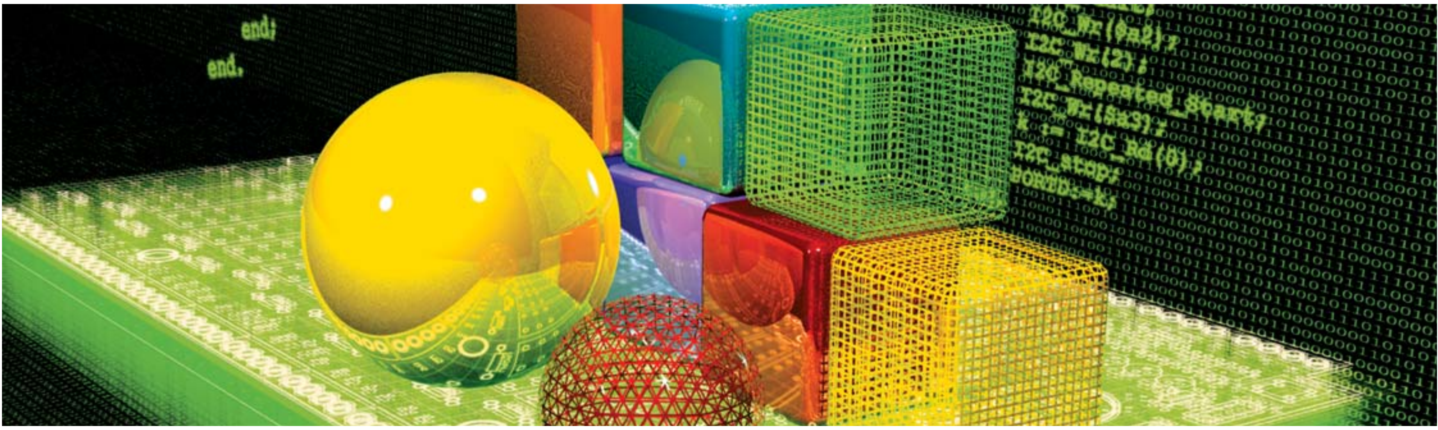
Usage of Modules Written in Assembler

The functions in modules written in assembler and then linked to a C program can be called from C functions. These assembler functions must comply with the C function calling convention. To declare an assembler function that will be included into the compiled stack, special assembler directives and operators can be used (check the MCASE documentation).

Note: When an assembler function returns control, the `BANK` register must contain the value of Bank 0 and the `ExtAdd` register value must be retained (`ExtAdd` must contain the original value after return from an assembler function).

In assembler modules, you can use the file "`include\tmpreg_mikro.inc`" which contains definitions for all symbolic names of GlobalRegisterArea registers.

CHAPTER 4



RSC-4x Language Reference

C offers unmatched power and flexibility in programming microcontrollers. RSC-4x mikroC adds even more power with an array of libraries, specialized for RSC-4x modules and communications. This chapter should help you learn or recollect C syntax, along with the specifics of programming RSC-4x modules. If you are experienced in C programming, you will probably want to consult mikroC Specifics first.

Lexical Elements Overview

These topics provide a formal definition of the RSC-4x 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 RSC-4x mikroC are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor.

C program starts as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the RSC-4x mikroC Code Editor). The basic program unit in RSC-4x 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:

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

Whitespace in Strings

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

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

parses to seven tokens, including the single string literal token:

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

Line Splicing with Backslash (\)

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

```
"mikroC \

Compiler"
```

parses as "`mikroC Compiler`". Refer to String Constants for more information.

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 RSC-4x mikroC.

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

Standard C comments

A 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 sequence may span multiple lines. After macro expansion, the entire sequence, including the four comment-delimiter symbols, is replaced by one space.

The following code

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

parses as:

```
int i;
```

Note that RSC-4x mikroC does not support the nonportable token pasting strategy using `/**/`. For more on token pasting, refer to Preprocessor Operators.

C++ style comments

The RSC-4x mikroC compiler allows C++ style comments – a single-line comment starting with two adjacent slashes (//). The comment can start at any position, and extends until the beginning of new line.

The following code

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

parses as:

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

Nested comments

RSC-4x mikroC as ANSI C doesn’t allow nested comments. The attempt to nest a comment like this

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

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

```
i ; */
```

which would generate a syntax error.

TokenToken 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.

RSC-4x mikroC recognizes following kinds of tokens:

keywords
identifiers
constants
operators
punctuators (also known as separators)
Tokens can be concatenated (pasted) by means of preprocessor operator `##`. See Preprocessor Operators for details.

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

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

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

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

Constants

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

RSC-4x 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 the `U` suffix, its data type will be the first of the following that can accommodate its value: `unsigned short, unsigned int, unsigned long int`.
- If the constant has the `L` suffix, its data type will be the first of the following that can accommodate its value: `long int, unsigned long int`.
- If the constant has both `L` and `U` suffixes, (`LU` or `UL`), its data type will be `unsigned long int`.

Decimal

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

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

In the absence of any overriding suffixes, the data type of a decimal constant is derived from its value, as shown on a next page.

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

Hexadecimal
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
The RSC-4x mikroC compiler allows integer binary constants to be used.
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
All constants with an initial zero are taken to be octal. If an octal constant contains illegal digits 8 or 9, the compiler will report an error. 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.

RSC-4x mikroC limits floating-point constants to range ±1.17549435082 * 10-38 .. ±6.80564774407 * 1038.

Here are some examples:

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

RSC-4x 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 RSC-4x mikroC). Larger numbers will generate the compiler error “Out of range”.

For example, the octal number \777 is larger than the maximum value allowed (\377) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

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

The following table shows the available escape sequences:

Sequence	Value	Char	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

Disambiguation

There are situations when ambiguities might arise when using escape sequences.

Let’s have an example:

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

This is intended to be interpreted as `\x09` and "`1.0 Intro`". However, RSC-4x mikroC compiles it as the hexadecimal number `\x091` and the literal string "`.0 Intro`". To avoid such problems, we could rewrite the code like this:

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

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

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

```
"\118"
```

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

String Constants

Note: This topic discusses the syntax of string constants according to ANSI C Standard; for the implemenatation specifics and storage see Strings Specifics.

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. This code, for example:

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

prints like this:

```
        "Name"\       Address
```

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

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

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

is equivalent to

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

Line Continuation with Backslash

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

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

Enumeration Constants

Enumeration constants are identifiers defined in `enum` type declarations. The identifiers are usually chosen as mnemonics to 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.

Consider the following examples:

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

The following assignments are legal:

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

The following assignments are illegal:

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

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

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.

Concepts

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

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

Objects
An object is a specific region of memory that can hold a fixed or variable value (or set of values). 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 RSC-4x 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**: 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**: File scope identifiers, also known as *globals*, are declared outside of all blocks; their scope is from the point of declaration to the end of the source file.

- **Function**: The only identifiers having function scope are statement labels. Label names can be used with goto statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing `label_name`: followed by a statement. Label names must be unique within a function.

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

Visibility

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

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

Technically, visibility cannot exceed scope, but scope *can* exceed visibility.

Name Spaces

Name space is the scope within which an identifier must be unique. C uses four distinct categories of identifiers:

1. `goto` label names. These must be unique within the function in which they are declared.
2. 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.
3. 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.
4. 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.
5. Duplicate names are legal for different name spaces regardless of scope rules.

For example:

```
int blue = 73;

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

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

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

Duration

Duration, closely related to 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 RSC-4x mikroC, static duration objects are not initialized to zero (or null) in the absence of any explicit initializer.

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

Local Duration

Local duration objects are also known as *automatic* objects. They are created on the stack (or in a register) when 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. For example:

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

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

Types

Note: This topic discusses the concept of types and covers standard C types. For specialized types and implemenation details see Types Specifics.

C is a 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 access.
- in many type-checking situations, to ensure that illegal assignments are trapped.

RSC-4x 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 RSC-4x mikroC standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that RSC-4x mikroC can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Type Categories

Common way to categorize types is to divide them into:

- fundamental
- derived

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. For more information on fundamental types, refer to the topic Fundamental Types.

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. For more information on derived types, refer to the topic Derived 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.

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.

Note: The integral type values are stored in the little-endian byte order.

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

Floating-point Types

All floating types (`float, double, long double`) use 32-bit IEEE real format (ANSI/IEEE 754-1985) of single-precision normalized numbers. The three types are of the same size and entirely compatible (casting would not produce any code), though their types are different.

Here is the overview of the floating-point types:

Type	Size in bytes	Range
`float`	4	±1.17549435 * 10-38 .. ±3.40282347 * 10^38
`double`	4	±1.17549435 * 10-38 .. ±3.40282347 * 10^38
`long double`	4	±1.17549435 * 10-38 .. ±3.40282347 * 10^38

Some features of the `double` and `long double` types do not comply with the standard:

Macro defined in <float.h>	Value in this implementation	ANSI C Standard requirements
`DBL_DIG`	6	>=10
`LDBL_DIG`	6	>=10
`DBL_EPSILON`	1.19209290 * 10-7	<=10-9
`LDBL_EPSILON`	1.19209290 * 10-7	<=10-9

Enumerations

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

Enumeration Declaration

Enumeration is declared like this:

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

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

```
int blue = 73;

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

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

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

Void Type

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

Void Functions

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

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

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

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

Generic Pointers

Pointers can be declared as `void`, 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. The derived types include:

- arrays
- pointers
- structures
- unions

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

Array Initialization

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:

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

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

If you specify both the length and starting values, the number of starting values must not exceed the specified length. 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

Note: This topic discusses the concept and syntax of pointers according to ANSI C Standard; for the implemenatation specifics and storage see Pointers Specifics.

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.

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*.

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` 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 ) { ... }
```

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

```
void *vp;
```

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

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

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]

/* .. but: */
y = *pa + 3;         // y = a[4] + 3
```

Also, you need to be careful with operator precedence:

```
*pa++;              // 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) {... /* won't be executed as 4 is not equal 2 */ }
if (pa > pb)  {... /* 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 type int */
int *pa = &a[0];  /* pa is pointer to int, pointing to a[0] */
*(pa + 3) = 6;
/* pa+3 is a pointer pointing to a[3], so a[3] now equals 6 */
pa++;
/* pa now points to the next element of array a: a[1] */
```

There is no such element as “one past the last element”, of course, but 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 type some_type */

  int i;
  some_type *p=&a[0];

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

Pointer Subtraction

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

Also, you may subtract two pointers. Difference will equal the distance between the two pointed addresses, in bytes.

For example:

```
int a[10];
int *pi1 = &a[0];
int *pi2 = &a[4];
i = pi2 - pi1;      /* i equals 8 */
pi2 -= (i >> 1);    /* pi2 = pi2 - 4: pi2 now points to 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 to be single objects. The RSC-4x mikroC structure type lets you handle complex data structures almost as easily as single variables.

Note: RSC-4x 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 {
  float r;
  struct Dot center;
} o1, o2;
/* declare variables o1 and o2 of Circle */
```

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. For example:

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

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

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

Incomplete Declarations

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

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

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

Untagged Structures and Typedefs

If you omit the structure tag, you get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited `member-declarator-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:

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

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

Usually, you don't need both `tag` and `typedef`: either can be used in structure type declarations.

Untagged structure and union members are ignored during initialization.

Note: See also Working with structures.

Working with Structures

Structures represent user-defined types. Set of rules governing the application of structures is strictly defined.

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

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

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

Structure Member Access

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

. (period)
-> (right arrow)

The operator . is called the direct member selector and it is used to directly access one of the structure's members. Suppose that the object s is of 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 the object `s` is of struct type `S`, and 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[21];
  double d;
} s, *sptr = &s;

//...

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

The expression `s.m` is 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 aa; double d;
} s, *sptr;

//...

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

Structure Uniqueness

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

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

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

the objects `aa` and `aaa` are both of 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: RSC-4x 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;
```

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. The identifier `pm` is a pointer to union `myunion`.

Size of Union

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

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

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

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

RSC-4x mikroC supports using bitfields in structures. The length of each bitfield cannot exceed 16 bits. Only unsigned bitfields are supported.

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.

Function Calls

The compiler arranges bitfields within 16-bit words. Bitfields are arranged starting from the lower bits in the word. Consequent bitfields are placed in the following higher bits right after the used bits. When there is not enough free bits remaining in the current word to accommodate next bitfield, the compiler places that bitfield in the next following word, thus leaving an unused gap between the words. If you specify an unnamed bitfield, the corresponding bits will also remain unused, simply serving as a gap between the bitfields.

Here's an example:

```
struct {                  /* the structure occupies two words */
  unsigned type   : 7;/* occupies [0...6] bits of the first word */
  unsigned         : 4;/* bits [7...10] of the first word are free*/
  unsigned index : 4;/* occupies [10...13] bits of the first word
*/
                      /* [14...15] bits of the first word are free  */
  unsigned value : 7;/* occupies [0...6] bits of the second word
*/
  unsigned exp    : 5;   /* occupies [7...11] bits of the second
word   */
  unsigned flag   : 1;   /* occupies bit 12 of the second word
*/
                      /* [13...15] bits of the second word are free */
} S;
```

Bit Fields Access

Bit fields can be accessed in same way as the structure members. Use direct and indirect member selector (`.` and `->`).

Performance

Bitfield operations are much more time-consuming for processor than operations with object of base types (**`int`**, **`char`**). Moreover, bitfield operations result in enlarged code size, therefore and it is not recommended to use bitfields unless RAM size is not enough to accommodate modifiable objects.

Types Conversion

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. For more information, refer to Standard Conversions.

Conversion is required in following situations:

- if statement requires an expression of particular type (according to language defi nition), and we use an expression of different type,
- if operator requires an operand of particular type, and we use an operand of dif ferent 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 func tion 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.

See also Specialized Types Conversions.

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, RSC-4x 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.

When a value of floating type is converted to integral type, the fractional part is discarded, in accordance with the standard. If the value of the integral part cannot be represented by the integral type, the result is the maximum valid value of the required integral type.

When a value of integral type is converted to a floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is the nearest lower value, also in accordance with the standard.

In details:

Here are the steps RSC-4x mikroC uses to convert the operands in an arithmetic expression:

First, any small integral types are converted according to the following rules:

1. `unsigned 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 RSC-4x 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`, then the other operand is converted to `long`.
4. Otherwise, if either operand is `unsigned`, then the other operand is converted to `unsigned`.
5. Otherwise, both operands are `int`.

The result of the expression is the same type as that of the two operands.

Here are several examples of implicit conversion:

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

Pointer 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
```

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

For example:

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

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

/* Following line will coerce a to double,
   then coerce b to double automatically,
   resulting in double type value: */

(double) a + b;     // equivalent to ((double) a) + b;
```

Declarations

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.

For example:

```
/* Here is a nondefining declaration of function max; */
/* it merely informs compiler that max is a function */
int max();
/* Here is a definition of function max: */
int max(int x, int y) {
  return (x >= y) ? x : y;
}
/* Definition of variable i: */
int i;
/* Following line is an error, i is already defined! */
int i;
```

Declarations and Declarators

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.

For example:

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

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

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

Linkage

An executable program is usually created by compiling several independent *translation units*, then linking the resulting object files with preexisting libraries. 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 Rules

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

The storage class specifiers `auto` and `register` cannot appear in an external declaration. 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.

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 RSC-4x 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 RSC-4x 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, RSC-4x mikroC stores variables within internal microcontroller memory. Thus, modifier `register` technically has no special meaning. RSC-4x 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, RSC-4x 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".

Inline Assembler

According to standard, C should allow embedding assembler in the source code by means of **`asm`** declaration. In addition, RSC-4x mikroC also supports declarations **`_asm`** and **`__asm`** which have the same meaning.

Group assembler instructions by the asm keyword (or **`_asm`** or **`__asm`**):

```
asm {
  block of assembler instructions
}
```

Assembler one-line comments starting with semicolon are allowed in the embedded assembly code; C/C++ style comments are also allowed.

Embedding assembler with pragma

For the sake of backward compatibility, RSC-4x mikroC compiler supports another method for embedding assembler in C source code.

Note: Preferred method is using the **`asm`** declaration as described above.

```
/* Single-line format: */
#pragma asm <machine_instruction0> [; <machine_instruction1> ] ...

or:

/* Block format: */
#pragma asm
<machine_instruction0> [; <machine_instruction1> ] ...
...
#pragma endasm
```

In the single-line format, instructions should follow the #pragma asm directive in the same line. You can specify more than one instruction in one directive. Instructions should be separated by semicolons.

In the block format, the `#pragma asm` directive indicates the beginning of assembler block, and `#pragma endasm` directive indicates the end of the block. You can write one or more instructions in each line of the block. Multiple instructions in one line should be separated by semicolons.
You can also use function-like macros in the assembly language blocks and lines. Inside an assembler block, the preprocessor acts in a regular way and processes the assembler block properly. Therefore, in the built-in assembler you can use C-style macros and other preprocessing directives. The `#` and `##` operators may not be used.

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 example:

```
struct dot {int x; int y; } m = {30, 40};
```

For more information, refer to Structures and Unions.

Also, you can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For more information, refer to Arrays.

Automatic Initialization

RSC-4x mikroC does not provide automatic initialization for objects. Uninitialized globals and objects with static duration will take random values from memory.

Functions

Note: This topic discusses the syntax and usage of functions. For specialized functions and implemenation details see Functions Specifics.

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 to be an expression like any other.

C allows a function to create results other than its return value, referred to as *side effects*. Often, 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 ANSI Compliance section for more info on functions' limitations on RSC-4x.

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 an array or another function.

Within parentheses, `parameter-declarator-list` is a list of formal arguments that function takes. These declarators specify the type of each function parameter. The compiler uses this information to check 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 (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 and Argument Conversions

Function Calls

A function is called with actual arguments placed in the same sequence as their matching formal parameters. Use the function-call operator ():

```
function_name(expression_1, ... , expression_n)
```

Each `expression` in the function call is an *actual argument*. Number and types of actual arguments should match those of formal function parameters. If types 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.

```
// For example, Lcd_Init takes the address of PORT,
// so it can change the value of an actual argument:
Lcd_Init(&PORTB);

// This would be wrong; you would pass the value
// of PORT to the function:
Lcd_Init(PORTB);
```

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.

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	`()` `[]` `.` `->`	->
14	1	`!` `~` `++` `--` `+` `-` `*` `&` `(type)` **`sizeof`**	<-
13	2	`*` `/` `%`	->
12	2	`+` `-`	->
11	2	`<<` `>>`	->
10	2	`<` `<=` `>` `>=`	->
9	2	`==` `!=`	->
8	2	`&`	->
7	2	`^`	->
6	2	`|`	->
5	2	`&&`	->
4	2	`||`	->
3	3	`?:`	<-
2	2	`=` `*=` `/=` `%=` `+=` `-=` `&=` `^=` `|=` `<<=` `>>=`	<-
1	2	`,`	->

Arithmetic Operators

Arithmetic operators are used to perform mathematical computations. They have numerical operands and return numerical results. 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
Binary Operators		
+	addition	12
-	subtraction	12
*	multiplication	13
/	division	13
%	modulus operator returns the remainder of integer division (cannot be used with floating points)	13
Unary Operators		
+	unary plus does not affect the operand	14
-	unary minus changes the sign of operand	14
++	increment adds one to the value of the operand. Postincrement adds one to the value of the operand after it evaluates; while preincrement adds one before it evaluates	14
--	decrement subtracts one from the value of the operand. Postdecrement subtracts one from the value of the operand after it evaluates; while predecrement subtracts one before it evaluates	14

Note: Operator * is context sensitive and can also represent the pointer reference operator.

Binary Arithmetic Operators

Division of two integers returns an integer, while remainder is simply truncated:

```
/* for example: */
7 / 4;           /* equals 1 */
7 * 3 / 4;       /* equals 5 */

/* but: */
7. * 3. / 4.; /* equals 5.25 because we are working with floats */
```

Remainder operand % works only with integers; 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 `--` (postincrement and postdecrement) 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 */

but:

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

Relational Operators

Use relational operators to test equality or inequality of expressions. If the expression evaluates to be true, it returns 1; otherwise it returns 0.

All relational operators associate from left to right.

Relational Operators Overview

Operator	Operation	Precedence
`==`	equal	9
`!=`	not equal	9
`>`	greater then	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   /* -> (a + 5) >= (c - (1.0 / e)) */
```

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

```
/* ok: */
5 > 7                   /* returns 0 */
10 <= 20                /* returns 1 */

/* this can be tricky: */
8 == 13 > 5 /* returns 0, as: 8 == (13 > 5)  ->  8 == 1  ->  0 */
14 > 5 < 3  /* returns 1, as: (14 > 5) < 3  ->  1 < 3  ->  1 */
a < b < 5  /* returns 1, as: (a < b) < 5  ->  (0 or 1) < 5 -> 1*/
```

Bitwise Operators

Use the bitwise operators to modify 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.

Operator	Operation	Precedence
`&`	`bitwise AND; compares pairs of bits and returns 1 if both bits are 1, otherwise returns 0`	8
`|`	`bitwise (inclusive) OR; compares pairs of bits and returns 1 if either or both bits are 1, otherwise returns 0`	6
`^`	`bitwise exclusive OR (XOR); compares pairs of bits and returns 1 if the bits are complementary, otherwise returns 0`	7
`~`	`bitwise complement (unary); inverts each bit`	14
`<<`	`bitwise shift left; moves the bits to the left, discards the far left bit and assigns 0 to the right most bit.`	11
`>>`	`bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends`	11

Logical Operations on Bit Level

&	**0**	**1**
0	0	0
1	0	1

|	**0**	**1**
0	0	1
1	1	1

^	**0**	**1**
0	0	1
1	1	0

~	**0**	**1**
	1	0

Bitwise operators `&`, `|`, and `^` perform logical operations on appropriate pairs of bits of their operands. Operator `~` complements each bit of its operand.

For example:

```
0x1234 & 0x5678          /* equals 0x1230 */

/* because ..

0x1234 : 0001 0010 0011 0100
0x5678 : 0101 0110 0111 1000
----------------------------
  &     : 0001 0010 0011 0000

.. that is, 0x1230 */

/* Similarly: */

0x1234 | 0x5678;         /* equals 0x567C */
0x1234 ^ 0x5678;         /* equals 0x444C */
~ 0x1234;                 /* equals 0xEDCB */
```

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

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 the 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 the 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

&&	0	x
0	0	0
x	0	1

||	0	x
0	0	1
x	1	1

!	0	x
	0	1

Precedence of logical, relational, and arithmetic operators was designated 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
```

The `expression1` is evaluated first. If its value is true, then `expression2` evaluates and `expression3` is ignored. If expression1 evaluates to false, then `expression3` evaluates and `expression2` is ignored. The result will be 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 operation (represented by an operator) rather than instruction.

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

```
expression1 = expression2
```

The `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.

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 */
sizeof(float)             /* returns 4 */
```

When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type):

```
int i, j, a[10];
//...
j = sizeof(a[1]);   /* j = sizeof(int) = 2 */
i = sizeof(a);      /* i = 10*sizeof(int) = 20 */
/* To get the number of elements in an array: */

int num_elem = i/j;
```

If the operand is a parameter declared as array type or function type, `sizeof` gives the size of the pointer. When applied to `structures` and `unions`, `sizeof` gives the total number of bytes, including any padding. 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 RSC-4x 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;
```

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:

- The label identifier serves as a target for the unconditional goto statement,

- 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;
```

RSC-4x 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 C 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:

- if
- switch

If Statement

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

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

When `expression` evaluates to true, `statement1` executes. If `expression` is false, `statement2` executes. The `expression` must evaluate to an integral value; otherwise, the condition is ill-formed. Parentheses around the `expression` are mandatory.

The else keyword is optional, but no statements can come between the if and the else.

Nested If statements

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

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

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

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.

Parentheses 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.

Here is a simple example with `switch`. Let's assume we have a variable `phase` with only 3 different states (0, 1, or 2) and a corresponding function (event) for each of these states. This is how we could switch the code to the appopriate routine:

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

Nested switch

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

Iteration Statements

Iteration statements let you loop a set of statements. There are three forms of iteration statements in RSC-4x mikroC:

- while
- do
- 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. Note that parentheses around `expression` are mandatory.

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

```
int s = 0, i = 0;
while (i < n) {
  s += a[i] * b[i];
  i++;
}
```

Note that body of 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-expression]; [condition-expression]; [increment-expres-
sion]) statement
```

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

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

All the expressions are optional. If `condition-expression` is left out, it is assumed to be always true. Thus, "empty" for statement is commonly used to create an endless loop in C:

```
for ( ; ; ) statement
```

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

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

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 RSC-4x mikroC:

- break
- continue
- goto
- return

Break and Continue Statements

Break Statement

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

Specifically, continue statement within the loop will jump to the marked position as you can see below:

```
while (..) {
.
.
.
// continue jumps here

}do {
.
.
.
// continue jumps here

while (..);

for (..;..;..;) {
.
.
.
// continue jumps here
}
```

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 (see File Inclusion),
- replacing specific lexical symbols with other symbols (see Macros),
- conditional compiling which conditionally includes or omits parts of code (see Conditional Compilation).

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

Preprocessor Directives

Any line in 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 white-space (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 RSC-4x 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.

Here is one commonly used directive:

```
#include <math.h>
```

For more information on including files with `#include` directive, refer to File Inclusion.

RSC-4x mikroC supports standard preprocessor directives:

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

For the list of pragma directives see Pragma Directives.

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 the 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 Add/Remove Files from Project 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. RSC-4x mikroC installation folder › “include” folder
2. your custom include paths

The "header_name" version specifies a user-supplied include file; RSC-4x mikroC will look for the header file in following locations, in this particular order:

1. the project folder (folder which contains the project file .psc)
2. RSC-4x mikroC installation folder › “include” folder
3. your custom include paths

Explicit Path

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

```
#include "C:\headers\my.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));
```

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)` would expand to identifier `cnt_2`.

Note
RSC-4x 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. RSC-4x 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

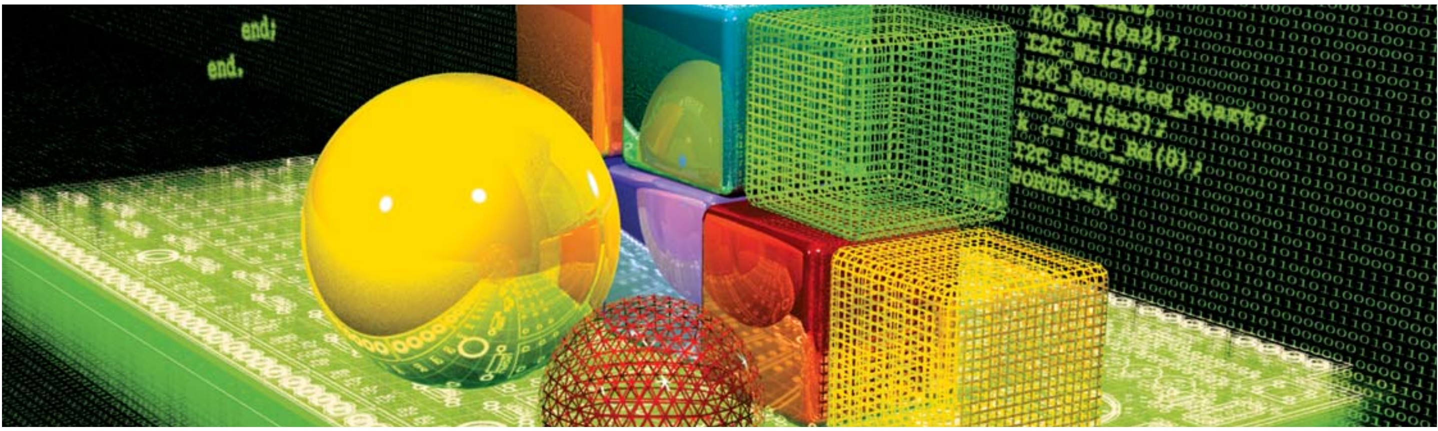
```
#ifdef identifier
```

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.

CHAPTER 5



RSC-4x mikroC Libraries

RSC-4x mikroC provides a number of built-in and library routines which help you develop your application faster and easier. Libraries for Character Handling Mathematics, Input/Output, General Utilities, String Handling, Compact Flash Library, LCD Library, Software I2C Library, Software SPI Library and many other are included along with practical, ready-to-use code examples.

RSC-4x mikroC Libraries

To use any of the following libraries include the appropriate header file in your source code.

C Standard Library

Character Handling (ctype.h)
Mathematics (math.h)
Input/Output (stdio.h)
General Utilities (stdlib.h)
String Handling

Additional Libraries

Compact Flash Library (cf.h)
LCD Library (lcd.h)
Software I2C Library (soft_i2c.h)
Software SPI Library (soft_spi.h)
See also Built-in Routines.

Note: All library functions have been compiled with the option “store const vars in CDATA area” *disabled*; i.e., library functions take only pointers to SRAM, which means that you should pay additional attention if enabling this option. (The only exception is the function sprintf.)

C Ctype Library

Header file ctype.h contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set.

Library Routines

```
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
ispunct
isspace
isupper
isxdigit
toupper
tolower
```

isalnum

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

isalpha

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

iscntrl

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

isgraph

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

islower

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

isprint

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

ispunct

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

isupper

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

isxdigit

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

toupper

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

C Math Library

Header file math.h contains declarations of common mathematical functions.

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, double y);`
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 *n);`
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 `exponent` is stored in the object pointed to by `n`.

ldexp

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

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.

tanh

Prototype	`double tanh(double x);`
Description	Function returns the hyperbolic tangent of `x`, defined mathematically as `sinh(x)/cosh(x)`.

C stdio.h Library

Header file stdio.h provides basic input and output capabilities of RSC-4x mikroC.

Library Routines

sprintf
sscanf

sprintf

Description: Function formats a series of strings and numeric values and stores the resulting string in *buffer*.

Note: format string must be in the CDATA area because `sprintf` parameter is of `cdata` type.

The *fmtstr* argument is a format string and may be composed of characters, escape sequences, and format specifications. Ordinary characters and escape sequences are copied to the *buffer* in the order in which they are interpreted. Format specifications always begin with a percent sign (%) and require additional arguments to be included in the function call.

The format string is read from left to right. The first format specification encountered references the first argument after fmtstr and converts and outputs it using the format specification. The second format specification accesses the second argument after fmtstr, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications. Format specifications have the following format:

% [flags] [width] [.precision] [{ h | l | L }] conversion_type

Each field in the format specification can be a single character or a number which specifies a particular format option. The *conversion_type* field is where a single character specifies that the argument is interpreted as a character, a string, a number, or a pointer, as shown in the following table.

Conversion Type	Argument Type	Output Format
d	int	Signed decimal number
u	unsigned int	Unsigned decimal number
o	unsigned int	Unsigned octal number
x	unsigned int	Unsigned hexadecimal number using 0123456789ABCEDF
X	double	Floating-point number using the format [-]dddd.dddd
e	double	Floating-point number using the format [-]d.dddde[-]dd
E	double	Floating-point number using the format [-]d.ddddE[-]dd
g	double	Floating-point number using either e or f format, whichever is more compact for the specified value and precision
c	int	The int is converted to an unsigned char, and the resulting character is written
s	char *	String with a terminating null character
p	void *	Pointer value, the X format is used
%	none	A % is written. No argument is converted. The complete conversion specification shall be %%.

The flags field is where a single character is used to justify the output and to print +/- signs and blanks, decimal points, and octal and hexadecimal prefixes, as shown in the following table.

Flags	Meaning
-	Left justify the output in the specified field width.
+	Prefix the output value with a + or - sign if the output is a signed type.
space (' ')	Prefix the output value with a blank if it is a signed positive value. Otherwise, no blank is prefixed
#	Prefixes a non-zero output value with 0, 0x, or 0X when used with o, x, and X field types, respectively. When used with the e, E, f, g, and G field types, the # flag forces the output value to include a decimal point. The # flag is ignored in all other cases.
*	Ignore format specifier.

The *width* field is a non-negative number that specifies the minimum number of characters printed. If the number of characters in the output value is less than width, blanks are added on the left or right (when the - flag is specified) to pad to the minimum width. If width is prefixed with a 0, zeros are padded instead of blanks. The width field never truncates a field. If the length of the output value exceeds the specified *width*, all characters are output.

The precision field is a non-negative number that specifies the number of characters to print, the number of significant digits, or the number of decimal places. The precision field can cause truncation or rounding of the output value in the case of a floating-point number as specified in the following table.

Flags	Meaning of precision field
d, u, o, x, X	The precision field is where you specify the minimum number of digits that will be included in the output value. Digits are not truncated if the number of digits in the argument exceeds that defined in the precision field. If the number of digits in the argument is less than the precision field, the output value is padded on the left with zeros.
f	The precision field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
e, E	The precision field is where you specify the number of digits to the right of the decimal point. The last digit is rounded.
g	The precision field is where you specify the maximum number of significant digits in the output value.
c, C	The precision field has no effect on these field types.
s	The precision field is where you specify the maximum number of characters in the output value. Excess characters are not output.

The optional characters h and l or L may immediately precede the *conversion_type* to respectively specify short or long versions of the integer types *d*, *i*, *u*, *o*, *x*, and *X*.

You must ensure that the argument type matches that of the format specification. You can use type casts to ensure that the proper type is passed to *sprintf*.

Prototype:

```
int sprintf (
  char *buffer,                  /* storage buffer */
  const cdata char *fmtstr,    /* format string */
  ... );                          /* additional arguments */
```

sscanf

Description: Function returns the number of input fields that were successfully converted. An EOF is returned if an error is encountered before any conversion.

Function reads data from the string `buffer`. Data input are stored in the locations specified by argument according to the format string `fmtstr`. Each argument must be a pointer to a variable that corresponds to the type defined in `fmtstr` which controls the interpretation of the input data.

The `fmtstr` argument is composed of one or more whitespace characters, non-whitespace characters, and format specifications as defined below.

Whitespace characters cause `sscanf` to skip whitespace characters in the `buffer`. Whitespace characters are such ones for which the `isspace` function returns non-zero value. Note that a single whitespace character in the format string matches 0 or more whitespace characters in the `buffer`.

Non-whitespace characters, with the exception of the percent sign (`%`), cause `sscanf` to read but not store a matching character from the `buffer`. The `sscanf` function terminates if the next character in the buffer does not match the specified non-whitespace character.

Format specifications begin with a percent sign (`%`) and cause `sscanf` to read and convert characters from the `buffer` to the specified type values. The converted value is stored to an argument in the parameter list. Characters following a percent sign that are not recognized as a format specification are treated as an ordinary character. For example, `%%` matches a single percent sign in the `buffer`.

The format string is read from left to right. Characters that are not part of the format specifications must match characters in the `buffer`. These characters are read from the `buffer` but are discarded and not stored. If a character in the `buffer` conflicts with the format string, `sscanf` terminates. Any conflicting characters remain in the `buffer`.

The first format specification encountered in the format string references the first argument after fmtstr and converts input characters and stores the value using the format specification. The second format specification accesses the second argument after fmtstr, and so on. If there are more arguments than format specifications, the extra arguments are ignored. Results are unpredictable if there are not enough arguments for the format specifications.

Values in the `buffer` are called input fields and are delimited by whitespace characters. When converting input fields, `sscanf` ends a conversion for an argument when a whitespace character is encountered. Additionally, any unrecognized character for the current format specification ends a field conversion. Format specifications have the following format:

```
%  *  width  { b | h | l }  type
```

Each field in the format specification can be a single character or a number which specifies a particular format option.

The type field is where a single character specifies whether input characters are interpreted as a character, string, or number. This field can be any one of the characters in the following table.

Character	Argument Type	Input Format
d	int *	Signed decimal number
i	int *	Signed decimal, hexadecimal, or octal integer
u	unsigned int *	Unsigned decimal number
o	unsigned int *	Unsigned octal number
x, X	unsigned int *	Unsigned hex number
c	char *	A single character
s	char *	A string of characters terminated by whitespace

An asterisk (*) as the first character of a format specification causes the input field to be scanned but not stored. The asterisk suppresses assignment of the format specification.

The `width` field is a non-negative number that specifies the maximum number of characters read from the `buffer`. No more than width characters will be read from the `buffer` and converted for the corresponding argument. However, fewer than width characters may be read if a whitespace character or an unrecognized character is encountered first.

The optional characters `h` and `l` or `L` may immediately precede the type character to respectively specify short, or long versions of the integer types `d, i, u, o,` and `x`.

C Stdlib Library

Header file `stdlib.h` contains functions for performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, and sorting.

Library Routines

```
abs
atof
atoi
atol
div
ldiv
labs
max
min
rand
srand
strtod
strtol
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 whitespace at the beginning. The string will be processed one character at a time, until the function reaches a character which it doesn't recognize (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.

strtod

Prototype	`double strtod (`
	`  const char * s,    /* string to be converted */`
	`  char ** sret);     /* ptr to final string */`

Returns	Function returns the converted value, if any. If no conversion performed, zero is returned. If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of value).
Description	Function converts the initial portion of the string pointed to by `s` to double representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, a subject sequence resembling floating point constant; and a final string of one or more unrecognized characters. Then, it attempts to convert the subject sequence to a floating point number, and returns the result. A pointer to the final string is stored in the object pointed to by `sret`, provided that `sret` is not a null pointer.

strtol

Prototype	`long int strtol (` `const char * s,    /* string to be converted   */` `char ** sret,      /* ptr to the final string */` `int base);         /* radix base */`
Returns	Function returns the converted value, if any. If no conversion performed, zero is returned. If the correct value is outside the range of representable values, `LONG_MAX` or `LONG_MIN` is returned (according to the sign of value).
Description	Function converts the initial portion of the string pointed to by s to **`long int`** representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters, a subject sequence resembling an integer represented in some radix determined by the value of `base`, and a final string of one or more unrecognized characters. Then, it attempts to convert the subject sequence to an integer, and returns the result. A pointer to the final string is stored in the object pointed to by `sret`, provided that `sret` is not a null pointer.

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).

C String Library

Header file `string.h` contains various functions for string handling.

Library Routines

```
memchr
memcmp
memcpy
memmove
memset
strcat
strchr
strcmp
strcpy
strcspn
strlen
strncat
strncmp
strncpy
strpbrk
strrchr
strspn
strstr
strtok
```

memchr

Prototype	`void *memchr(`
	`  const void *s,    /* buffer to search */`
	`  int c,             /* byte to find */`
	`  size_t n);        /* maximum buffer length */`

Returns	Function returns a pointer to the located character or a null pointer if the character was not found.
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	`int memcmp (`
	`  const void *buf1,       /* first buffer */`
	`  const void *buf2,       /* second buffer */`
	`  size_t len);`

Returns	Function returns a positive, negative, or zero value indicating the relationship of first `len` bytes of `buf1` and `buf2`.

Description	The memcmp function compares two objects `buf1` and `buf2` for len bytes and returns a value indicating their relationship as follows:
	
	`Value     Meaning`
	`< 0       buf1 less than buf2`
	`= 0       buf1 equal to buf2`
	`> 0       buf1 greater than buf2`
	
	In other words, the sign of nonzero value returned by function is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as **`unsigned char`**) that differ in the objects being compared.

memcpy

Prototype	`void *memcpy (`
	`  void *dest,          /* destination buffer */`
	`  const void *src,     /* source buffer */`
	`  size_t len);        /* bytes to copy */`

Returns	Function returns `dest`.
Description	Function copies len bytes from `src` to `dest`. If these memory buffers overlap, the `memcpy` function cannot guarantee that bytes in `src` are copied to `dest` before being overwritten. If these buffers do overlap, use the memmove function.

memmove

Prototype	`void *memmove (`
	`  void *dest,          /* destination buffer */`
	`  const void *src,     /* source buffer */`
	`  int len);           /* maximum bytes to move */`

Returns	Function returns `dest`.
Description	Function copies len bytes from `src` to `dest`. If these memory buffers overlap, the memmove function ensures that bytes in `src` are copied to `dest` before being overwritten.

memset

Prototype	`void *memset  (`
	`  void *s,             /* buffer to initialize */`
	`  int c,               /* byte value to set */`
	`  size_t len);        /* buffer length */`

Returns	Function returns the value of s.
Description	Function sets the first len bytes in object pointed to by `s` to `c` (converted to an **`unsigned char`**).

strcat

Prototype	`char *strcat (`
	`  char *dest,           /* destination string */`
	`  const char *src);    /* source string */`

Returns	Function returns `dest`.
Description	Function concatenates or appends `src` to `dest` and terminates dest with a null character.

strcat

Prototype	`char *strcat (`
	`  char *dest,           /* destination string */`
	`  const char *src);     /* source string */`

Returns	Function returns `dest`.
Description	Function concatenates or appends `src` to `dest` and terminates `dest` with a null character.

strchr

Prototype	`char *strchr (`
	`  const char *string, /* string to search */`
	`  char c);`

Returns	Function returns a pointer to the character `c` found in `string` or a null pointer if no matching character was found.
Description	Function searches `string` for the first occurrence of `c`. The null character terminating `string` is included in the search.

strcmp

Prototype	`char strcmp (`
	`  char *string1,      /* first string */`
	`  char *string2);     /* second string */`

Returns	Function returns the following values to indicate the relationship of string1 to string2:
	
	`Value    Meaning`
	`< 0      string1 less than string2`
	`= 0      string1 equal to string2`
	`> 0      string1 greater than string2`

Description	Function lexicographically compares the contents of string1 and string2 and returns a value indicating their relationship.

strcpy

Prototype	`char *strcpy (`
	`  char *dest,          /* destination string */`
	`  const char *src);    /* source string */`

Returns	Function returns `dest`.
Description	PFunction copies `src` to `dest` and appends a null character to the end of `dest`.

strcspn

Prototype	`size_t strcspn (`
	`  const char *src,     /* source string */`
	`  const char *set);    /* characters to find */`

Returns	Function returns the index of the first character located in `src` that matches any character in `set`. If the first character in `src` matches a character in `set`, a value of 0 is returned. If there are no matching characters in `src`, the length of the string is returned (not including the terminating null character).
Description	Function searches the `src` string for any of the characters in the `set` string.

strlen

Prototype	`int strlen (`
	`  char *src);       /* source string */`

Returns	Function returns the length of `src`.
Description	Function calculates the length, in bytes, of `src`. This calculation does not include the null terminating character.

strncat

Prototype	
	```
	char strncmp (
	  const char *string1,    /* *first string* */
	  const char *string2,    /* *second string* */
	  size_t len);               /* *maximum characters to compare* */
	```

Returns	Function returns `dest`.
Description	Function appends at most len characters from `src` to `dest` and terminates dest with a null character. If `src` is shorter than len characters, `src` is copied up to and including the null terminating character.

strncmp

Prototype	
	```
	char strncmp (
	  const char *string1,    /* *first string* */
	  const char *string2,    /* *second string* */
	  size_t len);               /* *maximum characters to compare* */
	```

Returns	Function returns the following values to indicate the relationship of `string1` to `string2`:
	```
	Value    Meaning
	< 0      string1 less than string2
	= 0      string1 equal to string2
	> 0      string1 greater than string2
	```

Description	Function lexicographically compares the first `len` bytes of `string1` and `string2` and returns a value indicating their relationship.

strncat

Prototype	
	```
	char *strncpy (
	  char *dest,          /* *destination string* */
	  const char *src,    /* *source string* */
	  size_t len);          /* *maximum characters to copy* */
	```

Returns	Function returns `dest`.
Description	Function copies at most `len` characters from `src` to `dest`. If `src` contains fewer characters than `len`, `dest` is padded out with null characters to `len` characters.

strpbrk

Prototype	`char *strpbrk (`
	`  const char *string,   /* string to search */`
	`  const char *set);     /* characters to find */`

Returns	Function returns a pointer to the matching character in `string`. If string contains no characters from set, a null pointer is returned.
Description	Function searches `string` for the first occurrence of any character from set. The null terminator is not included in the search.

strrchr

Prototype	`char *strrchr (`
	`  const char *string,     /* string to search */`
	`  int c);                  /* character to find */`

Returns	Function returns a pointer to the last character c found in `string` or a null pointer if no matching character was found.
Description	Function searches `string` for the last occurrence of `c`. The null character terminating `string` is included in the search.

strspn

Prototype	`size_t strspn (`
	`  const char *string,     /* string to search            */`
	`  const char *set);       /* characters that not allowed */`

Returns	Function returns the index of first character located in `string` that does not match a character in `set`. If the first character in `string` does not match a character in `set`, a value of 0 is returned. If all characters in `string` are found in `set`, the length of `string` is returned (not including the terminating null character).
Description	Function searches `string` for characters not found in the `set` string.

strstr

Prototype	`char * strstr (` `const char * s, /* string to search */` `const char * find); /* string to be found */`
Returns	Function returns a pointer to the located string, or a null pointer if the string is not found. If `find` points to a string with zero length, the function returns s.
Description	Function locates the first occurrence in the string pointed to by `s` of the sequence of characters (excluding the terminating null character) in the string pointed to by `find`.

strstr

Prototype	`char * strtok (` `char * s, /* string to be broken into tokens */` `const char * delim); /* separator-string */`
Returns	Function returns a pointer to the first character of token, or a null pointer if there is no token.
Description	A sequence of calls to the `strtok` function breaks the string pointed to by s into a sequence of tokens, each of which is delimited by a character from the string pointed to by `delim`. The first call in the sequence has s as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `delim` may be different from call to call. The first call in the sequence searches the string pointed to by `s` for the first character that is not contained in the current separator string pointed to by `delim`. If no such character is found, then there are no tokens in the string pointed to by `s` and the `strtok` function returns a null pointer. The `strtok` function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by `s`, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The `strtok` function saves a pointer to the following character, from which the next search for token will start. Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

Built-in Routines

The RSC-4x 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.

```
Delay_us
Delay_ms

Clock_Khz
Clock_Mhz

_cli_
_sti_
_nop_
_wdc_
```

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. This is an "inline" routine; code is generated in the place of the call.
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. This is an "inline" routine; code is generated in the place of the call.
Example	`Delay_ms(1000);  /* One second 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. This is an "in-line" routine; code is generated in the place of the call.
Example	`clk = Clock_Khz();`

Clock_Mhz

Prototype	`unsigned Clock_Mhz(void);`
Returns	Device clock in MHz, rounded to the nearest integer.
Description	Returns device clock in MHz, rounded to the nearest integer. This is an "in-line" routine; code is generated in the place of the call.
Example	`clk = Clock_Mhz();`

cli

Prototype	`void _cli_ (void);`
Returns	Nothing.
Description	Function executes the CLI instruction. This is an "in-line" routine; code is generated in the place of the call.

nop

Prototype	`void _nop_ (void);`
Returns	Nothing.
Description	Function executes the NOP instruction. This is an “in-line” routine; code is generated in the place of the call.

wdc

Prototype	`void _wdc_ (void);`
Returns	Nothing.
Description	Function executes the WDC instruction. This is an “in-line” routine; code is generated in the place of the call.

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. These routines can be used for CF with the FAT16 and the FAT32 file system. Note that routines for file handling can be used only with the FAT16 file system.

To use the CF Library, include the header file cf.h in your source code.

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

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

Library Routines

```
Cf_Init
Cf_Detect
Cf_Read_Init
Cf_Read_Byte
Cf_Write_Init
Cf_Write_Byte
```

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`. The function uses all 8 pins on dataport and 5 pins on `ctrlport`, as given in the CF library header file (*Cf_Lib.h*).
Example	`Cf_Init(&p2out, &p1out);`

Cf_Detect

Prototype	`unsigned short Cf_Detect(void);`
Returns	Returns 1 if CF is present, otherwise returns 0.
Requires	Control port (`ctrlport`) must be initialized. See Cf_Init.
Description	Checks for presence of CF card on `ctrlport`.

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

Cf_Read_Init

Prototype	`void Cf_Read_Init(unsigned long address, unsigned short 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	`unsigned short Cf_Read_Byte(void);`
Returns	Returns byte from CF.
Description	Reads one byte from CF.
Requires	Ports must be initialized. See Cf_Init. CF must be initialized for read operation. See Cf_Read_Init.

Example	`Read byte and display it on port0:`
	
	`p0out = Cf_Read_Byte();`

Cf_Write_Init

Prototype	`void Cf_Write_Init(unsigned long address, unsigned short sectc-nt);`
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(unsigned short 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`. CF must be initialized for write operation. See Cf_Write_Init.

Example	`int i = 20;`
	`Cf_Write_Byte(100);`
	`Cf_Write_Byte(i);`

Library Example

The following example writes 512 bytes at sector no.595, and then reads the data and prints on port0 for a visual check.

```
#include <rsc4128.h>
#include <cf.h>

void main() {
  unsigned i;

  //--- init rscxxx - all pins digital I/O
  cmpCtl |= 0x07;
  p0ctla = 255;
  p0ctlb = 255;
  //--- init CF
  Cf_Init(&p2out, &p1out);
  //--- wait until CF card is inserted
  while (!Cf_Detect()) ;
  //--- stabilize
  Delay_us(500);
  //--- init CF for write
  Cf_Write_Init(595, 1);
  //--- write 512 bytes to sector (595)
  for (i = 0; i < 512; i++)
    Cf_Write_Byte(i);
  //--- init CF for read
  Cf_Read_Init(595, 1);
  //--- read 512 bytes from sector (595)
  for (i = 0; i < 512; i++) {
    p0out = Cf_Read_Byte();
    Delay_ms(500);
  }

}//~!
```

LCD Library (4-bit interface)

RSC-4x mikroC provides a library for communicating with commonly used LCD (4-bit interface).

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 * data_port, char db3, char db2, char db1, char db0, char * ctrl_port, char rs, char rw, char enable);`
Description	Initializes LCD with data lines at `data_port` and control lines at `ctrl_port`, with pin settings you specify: parameters `rs, enable, rw, db3 .. db0` need to be a combination of values 0–7 (e.g. 3,6,0,7,2,1,4).
Example	`Lcd_Config(&p0out,4,5,6, &p1out,4,5,6,7);`

Lcd_Init

Prototype	`void Lcd_Init(unsigned short * data_port, unsigned short * ctrl_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(&p1out, &p0out);`

Lcd_Out

Prototype	`void Lcd_Out(unsigned short row, unsigned short 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(unsigned short row, unsigned short 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(unsigned short 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

Here's a simple test for LCD Library; the example should print a text message on the LCD connected to `port1` (data port) and `port0` (ctrl port).

```
#include <rsc4128.h>
#include <lcd.h>

char *text = "mikroElektronika";

void main() {
  //--- init rscxxx - all pins digital I/O
  cmpCtl |= 0x07;
  p0ctla = 255;
  p0ctlb = 255;
  p1ctla = 255;
  p1ctlb = 255;
  //--- Configure LCD connections
  Lcd_Config(&p1out, 7, 6, 5, 4, &p0out, 4, 5, 6);
  Lcd_Cmd(LCD_CURSOR_OFF);              // Turn off cursor
  Lcd_Out(1, 1, text);                   // Print text at LCD
}//~!
```

Software I2C Library

RSC-4x 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(unsigned short *portOut, char SDI, char SD0, 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(&p0Out, 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	`unsigned short Soft_I2C_Read(unsigned short ack);`
Returns	Returns one byte from the slave.
Description	Reads one byte from the slave, and sends not *acknowledge* signal if parameter ack is 0, otherwise it sends *acknowledge*.
Requires	START signal needs to be issued in order to use this function. See `Soft_I2C_Start`.

Example	Read data and send not *acknowledge* signal:
	`short take;`
	`...`
	`take = Soft_I2C_Read(0);`

Soft_I2C_Write

Prototype	`unsigned short Soft_I2C_Write(unsigned short 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

The following example is a simple demonstration how to read date and time from PCF8583 RTC (real-time clock). Date and time are read from the RTC every second and printed on LCD.

```
#include <rsc4128.h>
#include <soft_i2c.h>
#include <lcd.h>

void Delay_ms(unsigned int time_ms);

unsigned char sec, mnt, hr, day, mn, year;
char *txt, tnum[4];

char *R_Trim(char *str1){
  while (*str1 == ' ')
    str1++;
  return str1;
}//~

void Zero_Fill(char *value) {     // fill text repesentation
  if (value[1] == 0) {             //       with leading zero
    value[1] = value[0];
    value[0] = 48;
    value[2] = 0;
  }
}//~

//---------------------- Reads time and date information from RTC (PCF8583)
void Read_Time(char *sec, char *mnt, char *hr, char *day, char *mn, char *year)
{
  Soft_I2C_Start();
  Soft_I2C_Write(0xA0);
  Soft_I2C_Write(2);
  Soft_I2C_Start();
  Soft_I2C_Write(0xA1);
  *sec = Soft_I2C_Read(1);
  *mnt = Soft_I2C_Read(1);
  *hr = Soft_I2C_Read(1);
  *day = Soft_I2C_Read(1);
  *mn = Soft_I2C_Read(0);
  Soft_I2C_Stop();
}//~

//continues ...
```

```
//-------------------- Formats date and time
void Transform_Time(char  *sec, char *mnt, char *hr, char *day,
char *mn, char *year) {
  *sec  =  ((*sec & 0xF0) >> 4)*10 + (*sec & 0x0F);
  *mnt  =  ((*mnt & 0xF0) >> 4)*10 + (*mnt & 0x0F);
  *hr   =  ((*hr & 0xF0) >> 4)*10 + (*hr & 0x0F);
  *year =  (*day & 0xC0) >> 6;
  *day  =  ((*day & 0x30) >> 4)*10 + (*day & 0x0F);
  *mn   =  ((*mn & 0x10) >> 4)*10 + (*mn & 0x0F);
}//~

//-------------- Converts unsigned short number to string
char * ByteToStr(unsigned short inByte, char *outStr) {
  char *strCnv;
  unsigned short tmpV;

  strCnv = outStr + 4;
  *strCnv-- = 0;  //termination
  tmpV = inByte;

  while (tmpV != 0) {
    *strCnv-- = (tmpV % 10) + 48;      //ASCII value
    tmpV /= 10;
  }

  while (strCnv >= outStr)
    *strCnv-- = 32;                //SPACE

  return outStr;
}//~

//continues ...
```

```
//--------------------- Output values to LCD
void Display_Time(char sec, char mnt, char hr, char day, char mn,
char year) {
  char *tc;

   ByteToStr(day, tnum);                 // day
   tc = R_Trim(tnum);
   Zero_Fill(tc);
   Lcd_Out(1,6, tc);
   ByteToStr(mn, tnum);                  // month
   tc = R_Trim(tnum);
   Zero_Fill(tc);
   Lcd_Out(1,9, tc);
   Lcd_Chr(1,15,52+year);               // year
   ByteToStr(hr,tnum);                  // hour
   tc = R_Trim(tnum);
   Zero_Fill(tc);
   Lcd_Out(2,6,tc);
   ByteToStr(mnt,tnum);                 // minute
   tc = R_Trim(tnum);
   Zero_Fill(tc);
   Lcd_Out(2,9,tc);
   ByteToStr(sec,tnum);                 // second
   tc = R_Trim(tnum);
   Zero_Fill(tc);
   Lcd_Out(2,12,tc);
}//~

//------------------ Performs project-wide init
void Init_Main() {
  //--- init rscxxx - all pins digital I/O
  cmpCtl |= 0x07;
  p0ctla = 255;
  p0ctlb = 255;
  Lcd_Config(&p1out,7,6,5,4, &p0out,4,5,6);
  Soft_I2C_Init(&p0out, 2, 7);      // Initialize I2C
  txt = "Date:";            // Prepare and output static text on LCD
  Lcd_Out(1,1,txt);
  Lcd_Chr(1,8,':');
  Lcd_Chr(1,11,':');
  txt = "Time:";
  Lcd_Out(2,1,txt);
  Lcd_Chr(2,8,':');
  Lcd_Chr(2,11,':');
  txt = "200";
  Lcd_Out(1,12,txt);
  Lcd_Cmd(LCD_CURSOR_OFF);          // Cursor off
}//~
//continues ...
```

```
//------------------ Main function
void main() {
  Init_Main();                             // Perform initialization
  while (1) {
    Read_Time(&sec,&mnt,&hr,&day,&mn,&year);
    // Read time from RTC(PCF8583)

    Transform_Time(&sec,&mnt,&hr,&day,&mn,&year);
    // Format date and time

    Display_Time(sec, mnt, hr, day, mn, year);
    // Prepare and display on LCD

    Delay_ms(1000);
    // Wait 1s
  }
}//~!
```

Software SPI Library

RSC-4x mikroC provides library which implement software SPI. These routines are hardware independent and can be used with any MCU. You can easily communicate with other devices via SPI: A/D converters, D/A converters, MAX7219, LTC1290, etc.

The library configures SPI to master mode, clock = 50kHz, data sampled at the middle of interval, clock idle state low and data transmitted at low to high edge.

Note: These functions implement time-based activities, so interrupts need to be disabled when using the library.

Library Routines

```
Soft_Spi_Config
Soft_Spi_Read
Soft_Spi_Write
```

Soft_Spi_Config

Prototype	**void** Soft_Spi_Init(**unsigned short** *portOut, **const unsigned short** rx, **const unsigned short** tx, **const unsigned short** clk);
Description	Configures and initializes software SPI. Parameter `portOut` specifies port of MCU on which SDI (rx), SDO (tx), and SCK (clk) 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_Init` 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 (rx) pin is p0.5, SDO (tx) pin is p0.6 and SCK (clk) pin is p0.4: `Soft_Spi_Init(&p0Out, 5,6,4);`

Soft_Spi_Read

Prototype	`unsigned short Soft_Spi_Read(unsigned short 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	`short take, buffer;`
	`...`
	`take = Soft_Spi_Read(buffer);`

Soft_Spi_Write

Prototype	`void Soft_Spi_Write(unsigned short 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 code demonstrates using library routines for SPI communication. Also, this example demonstrates working with max7219. Eight 7 segment displays are connected to MAX7219. MAX7219 is connected to port2 and SDO, SDI, SCK pins are connected accordingly.

```
#define CHIP_SELECT 2
#define N_CHIP_SELECT 253

#include <rsc4128.h>
#include <soft_spi.h>

unsigned char i;

void Max7219_Init() {
  p2out &= N_CHIP_SELECT;   // SELECT MAX
  Soft_Spi_Write(0x09);     // BCD mode for digit decoding
  Soft_Spi_Write(0xFF);
  p2out |= CHIP_SELECT;     // DESELECT MAX
  p2out &= N_CHIP_SELECT;   // SELECT MAX
  Soft_Spi_Write(0x0A);
  Soft_Spi_Write(0x0F);     // Segment luminosity intensity
  p2out |= CHIP_SELECT;     // DESELECT MAX
  p2out &= N_CHIP_SELECT;   // SELECT MAX
  Soft_Spi_Write(0x0B);
  Soft_Spi_Write(0x07);     // Display refresh
  p2out |= CHIP_SELECT;     // DESELECT MAX
  p2out &= N_CHIP_SELECT;   // SELECT MAX
  Soft_Spi_Write(0x0C);
  Soft_Spi_Write(0x01);     // Turn on the display
  p2out |= CHIP_SELECT;     // DESELECT MAX
  p2out &= N_CHIP_SELECT;   // SELECT MAX
  Soft_Spi_Write(0x00);
  Soft_Spi_Write(0xFF);     // No test
  p2out |= CHIP_SELECT;     // DESELECT MAX
}//~
void main() {
  //--- init rscxxx - all pins digital I/O
  cmpCtl |= 0x07;
  p2ctla = 255;
  p2ctlb = 255;
  Soft_Spi_Init(&p2out,4,5,3);   // Configure SPI
  Max7219_Init();                 // Initialize  max7219
  for (i = 1; i <= 8u; i++) {
    p2out &= N_CHIP_SELECT;      // Select max7219
    Soft_Spi_Write(i);           // Send i to max7219 (digit place)
    Soft_Spi_Write(10-i);        // Send i to max7219 (digit)
    p2out |= CHIP_SELECT;        // Deselect max7219 }}//~!
```

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 RSC-4x mikroC for purely educational purposes, please contact us.

Problems with transport or delivery

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

Would you like to become mikroElektronika's distributor?

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

Other

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

mikroElektronika
Admirala Geprata 1B
11000 Belgrade
EUROPE

Phone: + 381 (11) 30 66 377, + 381 (11) 30 66 378
Fax: + 381 (11) 30 66 379
E-mail: office@mikroe.com
Website: www.mikroe.com